Università degli Studi di Padova

Dipartimento di Scienze Statistiche

Corso di Laurea Magistrale in

Scienze Statistiche


# Auto-Encoding Variational Bayes

**Relatore:** Prof. Mauro Bernardi

Dipartimento di Scienze Statistiche

**Laureando:** Mattia Bruno

**Matricola:** 2020313

Anno Accademico 2021/2022

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Introduction

Directed probabilistic models have become an important component of machine learning. Very often, data can be modeled with the help of latent variables, of which we are interested in computing the posterior distribution using Bayes' theorem. However, for most interesting models, this computation is intractable: we need to resort to approximate inference techniques. In this vein, we will focus primarily on a method called *variational inference* (VI), which we will equivalently refer to as *variational Bayes* (VB).

Variational Bayes is to optimally approximate the posterior density $p(\cdot)$ with a manageable density belonging to the $q(\cdot)$ family. This is done by minimizing the Kullback-Leibler divergence $\mathrm{KL}(q \,||\, p)$ (Kullback & Leibler, 1951), or more easily by maximizing a lower bound on the log likelihood, which provides a surrogate target more amenable to optimization. Compared with Markov chain Monte Carlo (MCMC) sampling methods, which are the leading paradigm of Bayesian statistics, variational inference provides an approach that is better suited to large data and complex models.

From a historical perspective, variational inference originated from ideas of neural networks (Hinton & Van Camp, 1993; Peterson, 1987) and statistical mechanics (Parisi, 1988). After that, there was a flurry of works that derived variational inference for models conditionally in the exponential family, as reviewed by (Jordan et al., 1999; Wainwright et al., 2008). Modern research on variational inference focuses on developing fast, automated algorithms for applying VI to massive data and nonconjugate models (Kucukelbir et al., 2017; Ranganath et al., 2014). Particular attention has been paid to the so-called *deep generative models* (Kingma & Welling, 2014; Rezende et al., 2014), which exploit hierarchical architectures to simulate

from the data generative process.

These models benefit from recent advances in *deep learning*, which is discovering layered representations of high-dimensional data with multiple levels of abstraction (Bengio et al., 2013). In this context, we distinguish methods such as *convolutional neural networks*, which are extremely successful in computer vision tasks, and *autoencoders*, which allow to learn useful features about data.

The aim of this thesis is to show how variational inference can be combined with the representational power of deep learning to arrive at *variational autoencoders* (VAEs) (Kingma & Welling, 2014), models that learn to generate new realistic data. VAEs are powerful but difficult-to-master methods, since they blend knowledge from variational Bayes and neural networks. We bridge this gap by untangling all the components inside the "black-box".

The work is organized as follows. Chapter 1 presents the basic ideas of variational inference with the Gaussian mixture model (GMM) example. With reference to this, we compare the Bayesian estimation with the maximum likelihood EM (expectation-maximization) algorithm, and then we apply VB for clustering natural images according to their color profiles.

In Chapter 2 we see how the Bayesian GMM example is a special case of VB applied to conditionally conjugate exponential family models, and we compare it with a corrispondent collapsed Gibbs sampler. Next we show how, within this family, stochastic variational inference (SVI) scales to large data. Finally, black box variational inference (BBVI) methods are discussed, which allow VI to be applied to a broad range of nonconjugate models.

Chapter 3 describes the theory and practicalities behind neural networks, ranging from their structure, to optimization methods and up to regularization strategies. Convolutional neural networks are presented, along with autoencoders, which represent a nonlinear extension of PCA (principal component analysis).

Lastly, Chapter 4 illustrates the example of variational autoencoders, which combine ideas from all the previous chapters. VAEs are applied to images of handwritten digits and celebrity faces.

# Chapter 1

# Variational Inference

Approximating difficult-to-compute probability densities is a central problem of modern statistics. This is particularly relevant within the Bayesian framework, where inference is founded on evaluation of the posterior distribution, which is intractable for many models. Let $\mathbf{x} = (x_1, \ldots, x_N)$ be a set of observed variables and $\mathbf{z} = (z_1, \ldots, z_K)$ be a set of latent variables. The inference problem amounts to computing the *posterior* density $p(\mathbf{z}|\mathbf{x})$, which can be obtained by applying the Bayes' theorem

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})}{p(\mathbf{x})}\,, \tag{1.1}$$

where $p(\mathbf{x}|\mathbf{z})$ is the *likelihood* of the data, $p(\mathbf{z})$ is the *prior* density of $\mathbf{z}$ and the denominator is calculated by marginalizing out the latent variables as

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, d\mathbf{z}. \tag{1.2}$$

The latter is known as *marginal likelihood* and it is usually called *evidence* in the computer science literature.

In many situations of practical interest, the evaluation of $p(\mathbf{x})$ is either not possible through closed-form analytical solutions or it requires exponential time. Therefore, in such situations, we need to rely on approximation schemes, which fall broadly into two classes, according to whether they are stochastic or deterministic. As for stochastic techniques, Markov chain

Monte Carlo (MCMC) sampling methods have revolutionized the Bayesian approach, in which they still remain the leading paradigm. MCMC methods first construct an ergodic Markov chain on $\mathbf{z}$ whose stationary distribution is the posterior density $p(\mathbf{z}|\mathbf{x})$. Then, it is possible to sample from the chain, approximating the posterior with an empirical estimate constructed from the collected samples. Given infinite computational resources, MCMC techniques provide guarantees of producing (asymptotically) exact samples from the target density (Robert & Casella, 2004); the approximation arises from the use of a finite amount of processor time.

The downside of these methods is that they tend to be computationally demanding in problems where datasets are large or models are very complex. In these settings, *variational approximations* are a better alternative to MCMC, since they provide an inferential approach that is faster and easier to scale up to massive data. Specifically, variational approximations constitute a deterministic approach based on analytical approximations to the posterior distribution. However, these techniques are limited in their approximation accuracy, as opposed to MCMC, which can be made arbitrarily accurate by increasing the Monte Carlo sample size. Summing up, the strengths and weaknesses of variational approximations are complementary to those of sampling methods.

From now on, we will focus on a family of variational approximation techniques called *variational inference* or *variational Bayes*, which has its roots in *variational calculus*, a branch of mathematics which studies small changes in functionals. The latter are a mapping from an input function to the corresponding outcome, e.g. the entropy $\mathbb{H}(p)$, which takes a probability distribution $p(x)$ as the input and returns the quantity $\mathbb{H}(p) = -\int p(x) \log p(x)\, dx$. Variational calculus is concerned with the problem of optimizing a functional over a class of functions on which that functional depends. Although there is nothing instrinsically approximate about this method, variational inference lends itself to finding approximate solutions, restricting the class of functions in a way that we present in the next section.

## 1.1 The density transform approach

The *density transform* approach (Ormerod & Wand, 2010) involves approximation of posterior densities $p(\mathbf{z}|\mathbf{x})$ by other distributions $q(\mathbf{z})$ for which inference is more tractable. The key idea is to solve this problem with optimization, restricting $q(\cdot)$ to belong to a family of tractable densities. Formally, let $\mathcal{Q}$ be a family of densities over the latent variables $\mathbf{z}$, whose single member is denoted with $q(\mathbf{z})$ and called *variational density*. The goal of variational Bayes is to find the density $q^*(\mathbf{z})$ that minimizes a particular functional, the Kullback-Leibler (KL) divergence between the variational density $q(\mathbf{z})$ and the true posterior density $p(\mathbf{z}|\mathbf{x})$:

$$\mathrm{KL}\left(q(\mathbf{z})\,||\,p(\mathbf{z}|\mathbf{x})\right) = \int q(\mathbf{z}) \log\left\{\frac{q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\right\} d\mathbf{z} \tag{1.3}$$

$$q^*(\mathbf{z}) = \underset{q(\mathbf{z})\in\mathcal{Q}}{\arg\min}\ \mathrm{KL}\left(q(\mathbf{z})\,||\,p(\mathbf{z}|\mathbf{x})\right). \tag{1.4}$$

The KL divergence is an information-theoretical measure of proximity between two densities. It is asymmetric, that is $\mathrm{KL}\left(q\,||\,p\right) \neq \mathrm{KL}\left(p\,||\,q\right)$, nonnegative and it is minimized when $q(\cdot) = p(\cdot)$.

Variational inference thus turns the inference problem into an optimization problem, the complexity of which depends on the family of approximate densities $\mathcal{Q}$ chosen. However, the objective in (1.3) is not computable because it requires the evaluation of the true posterior $p(\mathbf{z}|\mathbf{x})$. Let us see how the optimization problem can be rewritten in an alternative feasible form.

### 1.1.1 The evidence lower bound

The logarithm of the marginal likelihood satisfies

$$\begin{aligned}
\log p(\mathbf{x}) &= \log p(\mathbf{x}) \int q(\mathbf{z})\, d\mathbf{z} = \int q(\mathbf{z}) \log p(\mathbf{x})\, d\mathbf{z} \\
&= \int q(\mathbf{z}) \log\left\{\frac{p(\mathbf{x},\mathbf{z})/q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})/q(\mathbf{z})}\right\} d\mathbf{z} \\
&= \int q(\mathbf{z}) \log\left\{\frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})}\right\} d\mathbf{z} + \int q(\mathbf{z}) \log\left\{\frac{q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\right\} d\mathbf{z}.
\end{aligned} \tag{1.5}$$

So far we have assumed that $\mathbf{z}$ is a continuous random variable; the discrete case has a similar treatment, but with summations rather than integrals. Nevertheless, defining the *evidence lower bound* (ELBO) as

$$\text{ELBO}(q) = \int q(\mathbf{z}) \log \left\{ \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \right\} d\mathbf{z}\,, \tag{1.6}$$

we can rewrite equation (1.5) in the following alternative form

$$\log p(\mathbf{x}) = \text{ELBO}(q) + \text{KL}\left(q(\mathbf{z}) \,||\, p(\mathbf{z}|\mathbf{x})\right). \tag{1.7}$$

Since $\text{KL}(\cdot) \geq 0$, from equation (1.7) it follows immediately that, consistent with the name, the ELBO lower-bounds the log-marginal likelihood, that is $\log p(\mathbf{x}) \geq \text{ELBO}(q)$ for any $q(\mathbf{z})$.

Therefore, according to result (1.7), it is clear that maximizing the lower bound $\text{ELBO}(q)$ is equivalent to minimizing the Kullback-Leibler divergence $\text{KL}\left(q \,||\, p\right)$, which was the optimization problem of interest, as equation (1.4) shows. The essence of the density transform variational approach is thus approximation of the posterior density $p(\mathbf{z}|\mathbf{x})$ by a $q(\mathbf{z})$ for which $\text{ELBO}(q)$ is more tractable than $p(\mathbf{x})$. In order to achieve tractability, we restrict $q(\mathbf{z})$ to a more convenient class of densities and then maximize $\text{ELBO}(q)$ over that class: the complexity of the family $\mathcal{Q}$ determines the complexity of the optimization. Therefore, the goal of variational inference is to choose $\mathcal{Q}$ to be flexible enough to capture a density close to the true posterior, but simple enough for efficient optimization.

Two common restrictions for the $q$ density are:

(a) $q(\mathbf{z}) = \prod_{i=1}^{M} q_i(\mathbf{z}_i)$, for some partition $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$ of $\mathbf{z}$;

(b) $q(\cdot)$ is a member of a parametric family of distributions.

In the case of (a), the factorization is the only assumption being made and in principle each factor can take on any distribution. Thus (a) is a nonparametric restriction, known as *mean field variational Bayes* (MFVB), since it has roots in the framework of physics called *mean field theory* (Parisi, 1988). Instead restriction (b), when $q(\cdot)$ is assumed to be multivariate Gaussian, yields the *Gaussian variational Bayes* (GVB).

### 1.1.2 Mean field variational Bayes

Focusing on the mean field factorization, it is possible to rewrite the lower bound as

$$\begin{aligned}\text{ELBO}(q) &= \int \prod_{i=1}^{M} q_i(\mathbf{z}_i) \left\{ \log p(\mathbf{x}, \mathbf{z}) - \sum_{i=1}^{M} \log q_i(\mathbf{z}_i) \right\} d\,\mathbf{z}_1 \dots d\,\mathbf{z}_M \\ &= \int q_1(\mathbf{z}_1) \left\{ \int \left( \log p(\mathbf{x}, \mathbf{z})\, q_2(\mathbf{z}_2) \dots q_M(\mathbf{z}_M) \right) d\,\mathbf{z}_2 \dots d\,\mathbf{z}_M \right\} d\,\mathbf{z}_1 \\ &\quad - \int q_1(\mathbf{z}_1) \log q_1(\mathbf{z}_1)\, d\,\mathbf{z}_1 + \text{const}\,,\end{aligned} \tag{1.8}$$

where *const* contains terms not involving $q_1(\mathbf{z}_1)$. Define the new joint density $\tilde{p}(\mathbf{x}, \mathbf{z}_1)$ by the relation

$$\log \tilde{p}(\mathbf{x}, \mathbf{z}_1) = \int \log p(\mathbf{x}, \mathbf{z})\, q_2(\mathbf{z}_2) \dots q_M(\mathbf{z}_M)\; d\,\mathbf{z}_2 \dots d\,\mathbf{z}_M \; + \text{const.} \tag{1.9}$$

Then the lower bound is equal to

$$\text{ELBO}(q) = \int q_1(\mathbf{z}_1) \log \left\{ \frac{\tilde{p}(\mathbf{x}, \mathbf{z}_1)}{q_1(\mathbf{z}_1)} \right\} d\,\mathbf{z}_1 + \text{const.} \tag{1.10}$$

Recognizing that equation (1.10) is the negative Kullback-Leibler divergence between $q_1(\mathbf{z}_1)$ and $\tilde{p}(\mathbf{x}, \mathbf{z}_1)$, which is maximized when $q_1(\mathbf{z}_1) = \tilde{p}(\mathbf{x}, \mathbf{z}_1)$, we have that

$$\begin{aligned}q_1^*(\mathbf{z}_1) &= \underset{q_1(\mathbf{z}_1) \in \mathcal{Q}}{\arg\max}\ \text{ELBO}(q) = \tilde{p}(\mathbf{z}_1 | \mathbf{x}) \equiv \frac{\tilde{p}(\mathbf{x}, \mathbf{z}_1)}{\int \tilde{p}(\mathbf{x}, \mathbf{z}_1)\, d\,\mathbf{z}_1} \\ &\propto \exp\left\{ \int \log p(\mathbf{x}, \mathbf{z})\, q_2(\mathbf{z}_2) \dots q_M(\mathbf{z}_M)\; d\,\mathbf{z}_2 \dots d\,\mathbf{z}_M \right\} \\ &= \exp\left\{ \mathbb{E}_{-\mathbf{z}_1} \left[ \log p(\mathbf{x}, \mathbf{z}) \right] \right\},\end{aligned} \tag{1.11}$$

where $\mathbb{E}_{-\mathbf{z}_1}$ denotes expectation with respect to the density $\prod_{j \neq 1} q_j(\mathbf{z}_j)$. Repeating the same argument leads to the general expression for the optimal densities

$$q_i^*(\mathbf{z}_i) \propto \exp\left\{ \mathbb{E}_{-\mathbf{z}_i} \left[ \log p(\mathbf{x}, \mathbf{z}) \right] \right\}, \quad i = 1, \dots, M. \tag{1.12}$$

It is easily shown that a equivalent expression for the optimal $q_i^*(\mathbf{z}_i)$ is

$$q_i^*(\mathbf{z}_i) \propto \exp\left\{ \mathbb{E}_{-\mathbf{z}_i} \left[ \log p\left( \mathbf{z}_i \,|\, \mathbf{z}_{-i}, \mathbf{x} \right) \right] \right\}, \quad i = 1, \dots, M. \tag{1.13}$$

**Algorithm 1:** Coordinate ascent variational inference (CAVI)

**Initialize:** Variational factors $q_i^*(\mathbf{z}_i)$

**while** ELBO *has not converged* **do**

  **for** $i \in \{1, \dots, M\}$ **do**

    $q_i^*(\mathbf{z}_i) \leftarrow \frac{\exp\left\{\mathbb{E}_{-\mathbf{z}_i}[\log p(\mathbf{z}_i \mid \mathbf{z}_{-i}, \mathbf{x})]\right\}}{\int \exp\left\{\mathbb{E}_{-\mathbf{z}_i}[\log p(\mathbf{z}_i \mid \mathbf{z}_{-i}, \mathbf{x})]\right\} d\,\mathbf{z}_i}$

  **end**

  Compute ELBO$(q^*)$

**end**

**return** $q^*(\mathbf{z}) = \prod_{i=1}^{M} q_i^*(\mathbf{z}_i)$

Result (1.13) suggests an iterative procedure (see Algorithm 1) to solve for the optimal variational densities $q_i^*(\mathbf{z}_i)$ that is known as *coordinate ascent variational inference* (CAVI) (Ghahramani & Beal, 2001). CAVI cycles through the variational factors and replaces each of them in turn, climbing the ELBO: it can be shown that convergence to at least local optima is guaranteed (see e.g. Titterington & Wang, 2006).

We will see that if conditionally conjugate priors are used, then the $q_i^*$ belong to recognizable density families and the $q_i^*$ updates reduce to updating parameters in the $q_i^*$ family. Moreover, equation (1.13) reveals that CAVI is closely related to Gibbs sampling (Geman & Geman, 1984). Indeed, the distributions $p(\mathbf{z}_i \mid \mathbf{z}_{-i}, \mathbf{x})$ are known as *full conditionals* in the MCMC literature: the Gibbs sampler maintains a realization of the latent variables and iteratively samples from the full conditional of each variable. CAVI only requires the exponentiated expectation of the log-full conditionals.

Finally, it should be emphasized that the choice of the specific factorization is a crucial aspect of mean field variational Bayes.

**Example 1.** Suppose that the density $p(\mathbf{z}|\mathbf{x}) = p(z_1, z_2, z_3 \mid \mathbf{x})$ has to be approximated. Possible factorizations for $q(\mathbf{z})$ are

$$q(z_1)\, q(z_2)\, q(z_3), \quad q(z_1, z_2)\, q(z_3), \quad q(z_1)\, q(z_2, z_3), \quad q(z_1, z_3)\, q(z_2).$$

If there is strong correlation between any of the components of $\mathbf{z}$, choosing the first mean field full factorization, will result in a weak approximation.

Useful insights about conditional independence properties of the distribution of interest can be highlighted by expressing it as a particular type of probabilistic graphical model, the *Directed Acyclic Graph* (DAG). These kind of graphs are also known as *Bayesian networks* (Jordan, 1999), since they are especially helpful tools in Bayesian inference. A graph comprises *nodes* connected by *links*; in a DAG each node represents a random variable (or group of random variables) and the links express directed probabilistic relationships between these variables.

**Example 2.** Let $\mathbf{z} = (z_1, z_2, z_3, z_4)$ be latent variables. Consider a joint density $p(\mathbf{x}, \mathbf{z})$ with the following decomposition

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|z_2, z_3, z_4)\, p(z_4|z_3)\, p(z_2|z_1)\, p(z_1)\, p(z_3). \tag{1.14}$$

The DAG provides a simple visualization of the probabilistic structure:

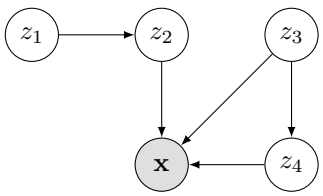



**Figure 1.1:** DAG describing the joint density (1.14).

Note that the observed variable $\mathbf{x}$ is denoted by shading the corresponding node and that the conditional distributions are depicted as directed links starting from the nodes of the conditioning variables. We say for example that node $\mathbf{x}$ is the *child* of node $z_3$ and symmetrically $z_3$ is the *parent* of $\mathbf{x}$, with $z_2$ and $z_4$ called the *co-parents* of $z_3$.

That being said, representations such as the one in Figure 1.1 can be exploited to define the notion of *Markov blankets*. In a DAG, the Markov blanket of the node $z_i$ is the set of its parents, children and co-parents; this concept is strictly connected to the properties of mean field variational Bayes approximations. Indeed, CAVI can be seen as a *message passing* algorithm, which means that during updates, considering the full conditional of a variable $z_i$, it is possible to drop all the variables that are not in its Markov blanket. See (Winn & Bishop, 2005) for further discussion.

## 1.2 A complete example: mixture of Gaussians

We now illustrate a full example involving the Gaussian mixture model, which provides many insights into the practical application of variational methods. First, we present the *expectation-maximization* (EM) algorithm to find (local) maximum likelihood estimates for this model. Then, we show how, adopting the variation inference framework, a Bayesian treatment elegantly solves some of the difficulties associated with the maximum likelihood approach.

The Gaussian mixture model is a linear superposition of Gaussian components, whose aim is to provide a richer class of densities models than the single Gaussian. Indeed a Gaussian mixture model is a universal approximator of densities, in the sense that almost any smooth density can be approximated to arbitrary accuracy by a mixture of Gaussians with enough components (Plataniotis & Hatzinakos, 2000).

Hence, consider a data set $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, where the generic observation $\mathbf{x}_n$ is a $D$-dimensional vector. Following (Bishop, 2006), the Gaussian mixture density can be written in the form

$$p(\mathbf{x}_n) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\,, \tag{1.15}$$

where each (multivariate) Gaussian $\mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ constitutes a *component* of the mixture and has its own mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$. The parameters $\pi_k$ are known as *mixing coefficients* and satisfies the requirements to be probabilities, namely $0 \leq \pi_k \leq 1$ together with $\sum_{k=1}^{K} \pi_k = 1$.

The Gaussian mixture can be expressed in terms of discrete latent variables, which are interpreted as defining assignments of data points to specific components of the mixture. Let us denote the (unknown) latent variables by $\mathbf{Z} = \{\mathbf{z}_1, \dots, \mathbf{z}_N\}$, where each generic $\mathbf{z}_n$ is a $K$-dimensional random variable, that defines which is the true component that generated the corresponding observation $\mathbf{x}_n$. Thus, $\mathbf{z}_n$ has a 1-of-$K$ representation in which a particular element $z_{nk}$ is equal to 1 and all other elements are equal to 0, satisfying $z_{nk} \in \{0, 1\}$ and $\sum_{k=1}^{K} z_{nk} = 1$.

We have that the marginal distribution over $\mathbf{z}_n$ is specified in terms of the mixing coefficients $\pi_k$ , as

$$p(z_{nk} = 1) = \pi_k. \tag{1.16}$$

Since $\mathbf{Z}$ is an array of 1-of-$K$ representations, the marginal distribution over $\mathbf{Z}$ is a categorical (Appendix A) and can be written down in the form

$$p(\mathbf{Z}) = \prod_{n=1}^{N} \mathrm{Cat}(\mathbf{z}_n|\boldsymbol{\pi}) = \prod_{n=1}^{N}\prod_{k=1}^{K} \pi_k^{z_{nk}}. \tag{1.17}$$

Similarly, the conditional distribution of $\mathbf{x}_n$ given a particular value for $\mathbf{z}_n$ is a Gaussian

$$p(\mathbf{x}_n|z_{nk} = 1) = \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{1.18}$$

and the conditional distribution of $\mathbf{X}$ given $\mathbf{Z}$ can be thus written as

$$p(\mathbf{X}|\mathbf{Z}) = \prod_{n=1}^{N}\prod_{k=1}^{K} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_{nk}}. \tag{1.19}$$

A quantity that will play an important role is the conditional probability of $\mathbf{z}_n$ given $\mathbf{x}_n$, which can be defined using Bayes' theorem

$$\begin{aligned} \gamma(z_{nk}) \equiv p(z_{nk} = 1|\mathbf{x}_n) &= \frac{p(z_{nk} = 1)\, p(\mathbf{x}_n|z_{nk} = 1)}{\sum_{j=1}^{K} p(z_{nj} = 1)\, p(\mathbf{x}_n|z_{nj} = 1)} \\ &= \frac{\pi_k\, \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j\, \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}. \end{aligned} \tag{1.20}$$

The quantity $\gamma(z_{nk})$ is viewed as the *responsibility* that component $k$ takes for explaining the observation $\mathbf{x}_n$. We shall view $\pi_k$ as the prior probability of $z_{nk} = 1$ and $\gamma(z_{nk})$ as the corresponding posterior probabilty once we have observed $\mathbf{x}_n$.

Finally, we can express the Gaussian mixture model as the directed acyclic graph shown in Figure 1.2. The surrounding box labeled with $N$ is called *plate* and indicates that there are $N$ i.i.d observations (nodes) with corresponding latent variables. This example provides a good illustration of the distinction between latent variables and parameters. Variables like $\mathbf{z}_n$, which are inside the plate, are regarded as latent variables since the number

of such variables grows with the size of the dataset. Contrarily, variables like $\boldsymbol{\mu}$ are considered parameters, because they are fixed in number independently of the size of the dataset. This distinction is important within the maximum likelihood approach, whereas from a Bayesian perspective there is really no fundamental difference between them.

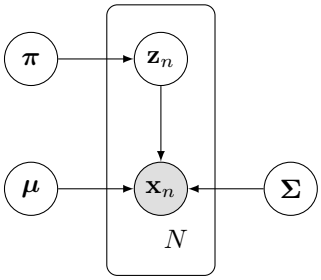



**Figure 1.2:** Graphical representation of the Gaussian mixture model for a set of $N$ i.i.d observations $\mathbf{x}_n$ with corresponding latent variables $\mathbf{z}_n$.

From formula (1.15) it follows that the log likelihood of the model is:

$$\log p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \log \left\{ \sum_{j=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{1.21}$$

The maximization of this function turns out to be problematic because of the presence of the summation inside the logarithm, which no longer acts directly on the Gaussian. As a result, setting the derivative of the log likelihood to zero, we do not obtain a closed form solution. In the next section, we try to solve this problem by replacing this difficult maximization with a series of easier maximizations, which is the key idea behind the EM algorithm.

### 1.2.1 Maximum likelihood EM algorithm

In general terms, consider a joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\vartheta})$ over observed variables $\mathbf{X}$ and latent variables $\mathbf{Z}$, governed by parameters $\boldsymbol{\vartheta}$. The goal of the *expectation-maximization* (EM) algorithm (Dempster et al., 1977) is to maximize the likelihood function $p(\mathbf{X}|\boldsymbol{\vartheta})$ with respect to $\boldsymbol{\vartheta}$.[1] The algorithm

[1] In general the EM algorithm can also be used to find MAP (maximum a posteriori) solutions for Bayesian models in which a prior $p(\boldsymbol{\vartheta})$ is defined over the parameters. However, in this chapter we refer only to the maximum likelihood approach.

involves the following steps

1. Choose an initial setting for the parameters $\widehat{\boldsymbol{\vartheta}}_{(0)}$.
2. **E step**: compute $\mathcal{Q}\left(\boldsymbol{\vartheta}, \widehat{\boldsymbol{\vartheta}}_{(j)}; \mathbf{X}\right) = \mathbb{E}_{\widehat{\boldsymbol{\vartheta}}_{(j)}}\left[\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\vartheta})\right]$, where the expectation is taken with respect to the distribution $p(\mathbf{Z}|\mathbf{X}, \widehat{\boldsymbol{\vartheta}}_{(j)})$.
3. **M step**: obtain $\widehat{\boldsymbol{\vartheta}}_{(j+1)} = \arg\max_{\boldsymbol{\vartheta}} \mathcal{Q}\left(\boldsymbol{\vartheta}, \widehat{\boldsymbol{\vartheta}}_{(j)}; \mathbf{X}\right)$.
4. Check for convergence of either the log likelihood or the parameter values. If the convergence criterion is not satisfied, return to step 2.

Therefore, EM algorithm iteratevely alternates between expectation and maximization steps: it can be proved (Neal & Hinton, 2000) that each cycle increase the log likelihood function (unless it is already at a local maximum).

Returning to the Gaussian mixture model, we can now suppose that the latent variables $\mathbf{Z}$ are observed. Augmenting the data, from (1.17) and (1.19) the log likelihood for the complete dataset $\{\mathbf{X}, \mathbf{Z}\}$ takes the form

$$\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\left\{\log \pi_k + \log \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\}. \tag{1.22}$$

Comparison with the *incomplete* log likelihood (1.21) shows that now the logarithm acts directly on the Gaussian distribution, which leads to a simpler maximization. However, in practice the values for the latent variables are unknown. Thus, as seen in the general EM algorithm, we need to consider the expectation of the complete log likelihood, with respect to the posterior distribution of the latent variables. Using (1.17), (1.19) and the Bayes' theorem, this posterior distribution takes the form

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) \propto \prod_{n=1}^{N}\prod_{k=1}^{K}\left[\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right]^{z_{nk}}. \tag{1.23}$$

The expected value of the indicator variable $z_{nk}$ under this posterior distribution is then given by

$$\begin{aligned}\mathbb{E}\left[z_{nk}\right] &= \frac{\sum_{z_{nk}} z_{nk}\left[\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right]^{z_{nk}}}{\sum_{z_{nj}}\left[\pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)\right]^{z_{nj}}} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K}\pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} = \gamma(z_{nk}),\end{aligned} \tag{1.24}$$

which is just (1.20), the responsibility of component $k$ for observation $\mathbf{x}_n$. Using this last result, it is therefore possible to obtain the expected value of the complete log likelihood (E step) as

$$\mathbb{E}_{\mathbf{Z}}\left[\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})\right] = \sum_{n=1}^{N}\sum_{k=1}^{K}\gamma(z_{nk})\left\{\log \pi_k + \log \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\}. \tag{1.25}$$

The M step then involves the maximization of the quantity (1.25) with respect to the parameters $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$ and $\pi_k$, given the responsibilities' values $\gamma(z_{nk})$. Starting by maximizing with respect to $\boldsymbol{\mu}_k$, we obtain the equation

$$0 = -\sum_{n=1}^{N}\underbrace{\frac{\pi_k\,\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j\,\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}}_{\gamma(z_{nk})}\boldsymbol{\Sigma}_k^{-1}\left(\mathbf{x}_n - \boldsymbol{\mu}_k\right). \tag{1.26}$$

Multiplying by $\boldsymbol{\Sigma}_k$ (which we assume to be nonsingular) and rearranging we obtain

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})\,\mathbf{x}_n\,, \tag{1.27}$$

where we have defined

$$N_k = \sum_{n=1}^{N}\gamma(z_{nk})\,, \tag{1.28}$$

which can be viewed as the effective number of points assigned to cluster $k$. Repeating a similar reasoning with respect to $\boldsymbol{\Sigma}_k$ gives

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})\left(\mathbf{x}_n - \boldsymbol{\mu}_k\right)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T. \tag{1.29}$$

Note that, for the $k$-th Gaussian component, both the mean $\boldsymbol{\mu}_k$ and the covariance $\boldsymbol{\Sigma}_k$ are obtained by taking a weighted mean (covariance) of all points in the dataset, in which the weighting factor for observation $\mathbf{x}_n$ is the responsibilty $\gamma(z_{nk})$.

Finally, we can maximize (1.25) with respect to $\pi_k$ using the Lagrange multiplier

$$\mathbb{E}_{\mathbf{Z}}\left[\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})\right] + \lambda\left(\sum_{k=1}^{K}\pi_k - 1\right), \tag{1.30}$$

whose maximization gives the equation

$$0 = \sum_{n=1}^{N}\frac{\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j\,\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} + \lambda. \tag{1.31}$$

**Algorithm 2:** EM for the Gaussian mixture model

**Initialize:** $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, $\pi_k$

**while** *log likelihood or parameters have not converged* **do**

**E step**: $\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)}$

**M step**: $N_k = \sum_{n=1}^{N} \gamma(z_{nk})$

$\boldsymbol{\mu}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})\, \mathbf{x}_n$

$\boldsymbol{\Sigma}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})\, (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T$

$\pi_k \leftarrow \frac{N_k}{N}$

Evaluate log $p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})$

**end**

**return** $\gamma(z_{nk})$, $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, $\pi_k$

Multiplying (1.30) by $\pi_k$ and summing over $k$ using $\sum_k \pi_k = 1$, we obtain $\lambda = -N$. Using this to eliminate $\lambda$ and rearranging, we find

$$\pi_k = \frac{N_k}{N}, \tag{1.32}$$

which means that the $k$-th mixing coefficient can be interpreted as the effective proportion of data points assigned to cluster $k$.

Thus, the results we derived suggest an iterative scheme for the EM algorithm, in order to find (local) maximum likelihood solutions. As Algorithm 2 shows, the E step evaluate the posterior probabilities (responsibilities) by result (1.20); while the M step re-estimates the means, covariances and mixing coefficients using the results (1.27), (1.29) and (1.32).

However, there is a notable problem associated with the maximum likelihood approach applied to Gaussian mixture models, due to the presence of singularities. Without loss of generality, consider the case where the covariances matrices of the components take the simplified form $\boldsymbol{\Sigma}_k = \sigma_k^2 \mathbf{I}$, with $\mathbf{I}$ identity matrix. Now suppose that the $j$-th component of the mixture has its mean $\boldsymbol{\mu}_j$ exactly equal to one observation, that is $\boldsymbol{\mu}_j = \mathbf{x}_n$ for some $n$. Then this observation will contribute to the likelihood with the term

$$\mathcal{N}(\mathbf{x}_n|\mathbf{x}_n, \sigma_k^2 \mathbf{I}) = \frac{1}{(2\pi)^{1/2}\, \sigma_j}. \tag{1.33}$$

Considering the limit $\sigma_j \to 0$, we observe that the term (1.33) and the corresponding likelihood go to infinity. That arises whenever one of the Gaussian components 'collapses' onto a specific observation and in the general case, it leads to a singular covariance matrix, hence the name.

An illustration is provided in Figure 1.3, which shows that one component of the mixture has finite variance and assigns probability to all the observations, while the 'spiky' component shrinks onto one specific data point. This is an example of the overfitting that can occur in a maximum likelihood framework.

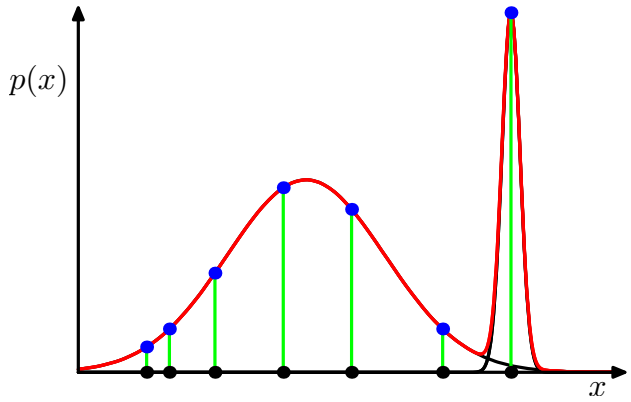



**Figure 1.3:** Singularity issue in a Gaussian mixture model. The red curve is the mixture density fitted through the maximum likelihood approach, while the black points are the true observations drawn from a single Gaussian. Source: (Bishop, 2006)

### 1.2.2 Variational Bayesian estimation

We now add a Bayesian flavor to the Gaussian mixture model by applying the variational inference machinery developed in Section 1.1. It will be shown how this Bayesian treatment elegantly solves some difficulties related to the maximum likelihood approach.

Consider again the $N \times D$ observed data set $\mathbf{X}$ and the corresponding $N \times K$ matrix of latent variables $\mathbf{Z}$. The specification of the mixture model is the same as discussed above, but with the addition of prior information about the parameters. Therefore, the conditional distribution of $\mathbf{Z}$ given the

mixing coefficients $\boldsymbol{\pi}$ is a categorical, that is

$$p(\mathbf{Z}|\boldsymbol{\pi}) = \prod_{n=1}^{N} \text{Cat}(\mathbf{z}_n|\boldsymbol{\pi}) = \prod_{n=1}^{N}\prod_{k=1}^{K} \pi_k^{z_{nk}}. \tag{1.34}$$

In the same way, the conditional distribution of the observations, given the latent variables and the component parameters, takes the Gaussian form

$$p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = \prod_{n=1}^{N}\prod_{k=1}^{K} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1})^{z_{nk}}, \tag{1.35}$$

where the model is parametrized with the precision matrices $\{\boldsymbol{\Lambda}_k\}$ because this will simplify the mathematics.

According to (Bishop, 2006), priors over the parameters $\boldsymbol{\mu}$, $\boldsymbol{\Lambda}$ and $\boldsymbol{\pi}$ are chosen to be conditionally conjugate distributions (Gelman et al., 2013). Specifically, the prior on the mixing weights $\boldsymbol{\pi}$ is a Dirichlet distribution

$$p(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}_0) = C(\boldsymbol{\alpha}_0)\prod_{k=1}^{K} \pi_k^{\alpha_0 - 1}, \tag{1.36}$$

where $C(\boldsymbol{\alpha}_0)$ is the Dirichlet normalization costant defined by (A.6) in Appendix A. The hyperparameter $\boldsymbol{\alpha}_0$ can be interpreted as the prior number of observations associated with each component of the mixture. By symmetry, the same $\alpha_0$ has been chosen for each of the components.

As for the other two parameters, the conditionally conjugate prior is an independent Gaussian-Wishart distribution, which governs the unknown mean and precision of each Gaussian component, such that

$$\begin{aligned} p(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= p(\boldsymbol{\mu}|\boldsymbol{\Lambda})\, p(\boldsymbol{\Lambda}) \\ &= \prod_{k=1}^{K} \mathcal{N}\left(\boldsymbol{\mu}_k|\mathbf{m}_0, (\beta_0\, \boldsymbol{\Lambda}_k)^{-1}\right)\, \mathcal{W}\left(\boldsymbol{\Lambda}_k|\mathbf{W}_0, \nu_0\right), \end{aligned} \tag{1.37}$$

where $\mathcal{W}(\cdot)$ stands for the Wishart distribution, defined in Appendix A.

At this point, it is possible to express the complete model through the following decomposition of the joint distribution

$$p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda})\, p(\mathbf{Z}|\boldsymbol{\pi})\, p(\boldsymbol{\pi})\, p(\boldsymbol{\mu}|\boldsymbol{\Lambda})\, p(\boldsymbol{\Lambda}), \tag{1.38}$$

which corresponds to the graphical representation provided in Figure 1.4. Observe that there is a link from $\boldsymbol{\Lambda}$ to $\boldsymbol{\mu}$ because, according to (1.37), the variance of the prior over $\boldsymbol{\mu}$ depends on $\boldsymbol{\Lambda}$.

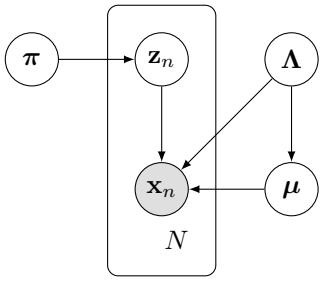



**Figure 1.4:** Directed acyclic graph representation of the Bayesian Gaussian mixture model for a set of $N$ i.i.d observations $\mathbf{x}_n$ with corresponding latent variables $\mathbf{z}_n$.

Following (Bishop, 2006), consider now an approximation to the intractable posterior $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})$, given by the variational distribution that factorizes between the latent variables and the parameters

$$q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = q(\mathbf{Z})\, q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}). \tag{1.39}$$

It should be emphasized that this is the only assumption we make in order to achieve a tractable solution; the functional form of the factors will then specified automatically by the optimization. Using the general result (1.12) and absorbing into the constant all the terms of the joint density (1.38) that do not depend on $\mathbf{Z}$, the optimal $q(\mathbf{Z})$ has the form

$$\log q^*(\mathbf{Z}) = \mathbb{E}_{\boldsymbol{\pi}} \left[\log p(\mathbf{Z}|\boldsymbol{\pi})\right] + \mathbb{E}_{\boldsymbol{\mu},\boldsymbol{\Lambda}} \left[\log p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda})\right] + \text{const.} \tag{1.40}$$

Substituting for the conditional distributions (1.34) and (1.35), we obtain

$$\log q^*(\mathbf{Z}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \log \rho_{nk} + \text{const}, \tag{1.41}$$

where we have defined

$$\begin{aligned} \log \rho_{nk} = \mathbb{E}\left[\log \pi_k\right] - \frac{D}{2} \log 2\pi + \frac{1}{2} \mathbb{E}\left[\log |\boldsymbol{\Lambda}_k|\right] \\ - \frac{1}{2} \mathbb{E}_{\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k}[(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Lambda}_k (\mathbf{x}_n - \boldsymbol{\mu}_k)]. \end{aligned} \tag{1.42}$$

Exponentiating both sides of (1.41) gives

$$q^*(\mathbf{Z}) \propto \prod_{n=1}^{N} \prod_{k=1}^{K} \rho_{nk}^{z_{nk}}, \tag{1.43}$$

which can be normalized in the following way:

$$q^*(\mathbf{Z}) = \prod_{n=1}^{N}\prod_{k=1}^{K} r_{nk}^{z_{nk}}, \qquad r_{nk} = \frac{\rho_{nk}}{\sum_{j=1}^{K}\rho_{nj}}. \tag{1.44}$$

Recognize that $q^*(\mathbf{Z})$ takes the same distribution as $p(\mathbf{Z}|\boldsymbol{\pi})$, that is categorical, with $\mathbb{E}[z_{nk}] = r_{nk}$ , where by construction $r_{nk} \geq 0$ and $\sum_{k=1}^{K} r_{nk} = 1$. Thus, $r_{nk}$ is the responsibility that component $k$ takes for explaining observation $\mathbf{x}_n$. It is convenient to define the following three statistics of the dataset calculated with respect to the responsibilities:

$$N_k = \sum_{n=1}^{N} r_{nk} \tag{1.45}$$

$$\overline{\mathbf{x}}_k = \frac{1}{N_k}\sum_{n=1}^{N} r_{nk}\,\mathbf{x}_n \tag{1.46}$$

$$\mathbf{S}_k = \frac{1}{N_k}\sum_{n=1}^{N} r_{nk}\,(\mathbf{x}_n - \overline{\mathbf{x}}_k)(\mathbf{x}_n - \overline{\mathbf{x}}_k)^T, \tag{1.47}$$

analogous to quantities (1.27) , (1.28) and (1.29) seen for the EM algorithm.

Consider now the other factor $q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})$. Again using the general result (1.12) and the joint density (1.38), the optimal solution is given by:

$$\begin{aligned}\log q^*(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = \log p(\boldsymbol{\pi}) + \sum_{k=1}^{K} \log p(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) + \mathbb{E}_{\mathbf{Z}}[\log p(\mathbf{Z})] \\ + \sum_{k=1}^{K}\sum_{n=1}^{N} \mathbb{E}[z_{nk}]\, \log \mathcal{N}\left(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}\right) + \text{const.}\end{aligned} \tag{1.48}$$

Note that the right-hand side comprises terms involving only $\boldsymbol{\pi}$ together with a sum over $k$ of terms involving only $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$, which leads to the further factorization

$$q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = q(\boldsymbol{\pi}) \prod_{k=1}^{K} q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k). \tag{1.49}$$

It should be emphasized that the latter is known as *induced factorizaton*, since it arises from the interaction between the assumed factorization and the conditional indipendence properties of the true distribution shown in Figure 1.4.

Focusing on the terms of (1.48) that depend on $\boldsymbol{\pi}$, it follows that

$$\log q^*(\boldsymbol{\pi}) = (\alpha_0 - 1) \sum_{k=1}^{K} \log \pi_k + \sum_{k=1}^{K} \sum_{n=1}^{N} r_{nk} \log \pi_k + \text{const}, \tag{1.50}$$

where we used the fact that $\mathbb{E}[z_{nk}] = r_{nk}$. We recognize the distribution to be the same of the conjugate prior, that is a Dirichlet

$$q^*(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}), \tag{1.51}$$

where $\boldsymbol{\alpha}$ has components $\alpha_k$ given by

$$\alpha_k = \alpha_0 + N_k. \tag{1.52}$$

Finally, we can obtain the optimal variational posterior distribution for $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ using the chain rule $q^*(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = q^*(\boldsymbol{\mu}_k|\boldsymbol{\Lambda}_k)\, q^*(\boldsymbol{\Lambda}_k)$. As expected from looking at the conjugate prior (1.37), the result is a Gaussian-Wishart distribution

$$q^*(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = \mathcal{N}\left(\boldsymbol{\mu}_k|\mathbf{m}_k, (\beta_k\, \boldsymbol{\Lambda}_k)^{-1}\right)\, \mathcal{W}\left(\boldsymbol{\Lambda}_k|\mathbf{W}_k, \nu_k\right), \tag{1.53}$$

where posterior hyperparameters are defined by the conjugacy property (A.24) such that

$$\beta_k = \beta_0 + N_k \tag{1.54}$$

$$\mathbf{m}_k = \frac{1}{\beta_k}\left(\beta_0 \mathbf{m}_0 + N_k \overline{\mathbf{x}}_k\right) \tag{1.55}$$

$$\mathbf{W}_k^{-1} = \mathbf{W}_0^{-1} + N_k \mathbf{S}_k + \frac{\beta_0 N_k}{\beta_0 + N_k}\left(\overline{\mathbf{x}}_k - \mathbf{m}_0\right)\left(\overline{\mathbf{x}}_k - \mathbf{m}_0\right)^T \tag{1.56}$$

$$\nu_k = \nu_0 + N_k. \tag{1.57}$$

Note that the optimal variational densities (1.51) and (1.53) depend on the statistics $N_k$ , $\overline{\mathbf{x}}_k$ and $\mathbf{S}_k$, which requires the expectations of the latent variables $\mathbb{E}[z_{nk}] = r_{nk}$ , that are the responsibilities. These are obtained by normalizing the quantities $\rho_{nk}$ in (1.42), which in turn involve expectations with respect to the variational distributions of the parameters. The evaluation of these three expectations gives

$$\mathbb{E}_{\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k}\left[(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Lambda}_k (\mathbf{x}_n - \boldsymbol{\mu}_k)\right] = D\beta_k^{-1} + \nu_k (\mathbf{x}_n - \mathbf{m}_k)^T \mathbf{W}_k (\mathbf{x}_n - \mathbf{m}_k) \tag{1.58}$$

$$\log \widetilde{\pi}_k \equiv \mathbb{E}[\log \pi_k] = \psi(\alpha_k) - \psi(\widehat{\alpha}) \tag{1.59}$$

$$\log \widetilde{\mathbf{\Lambda}}_k \equiv \mathbb{E}\left[\log|\mathbf{\Lambda}_k|\right] = \sum_{i=1}^{D} \psi\left(\frac{\nu_k + 1 - i}{2}\right) + D\log 2 + \log|\mathbf{W}_k|, \tag{1.60}$$

where we have introduced definitions of $\log \widetilde{\pi}_k$ and $\log \widetilde{\mathbf{\Lambda}}_k$. Result (1.58) follows from (A.20), while results (1.59) and (1.60) are properties of the Dirichlet and Wishart distributions respectively listed in (A.10) and (A.32). Substituting the three previous expectations into (1.42) and normalizing, the responsibilities are such that

$$r_{nk} \propto \widetilde{\pi}_k \widetilde{\mathbf{\Lambda}}_k \exp\left\{ -\frac{D}{2\beta_k} - \frac{\nu_k}{2}(\mathbf{x}_n - \mathbf{m}_k)^T \mathbf{W}_k (\mathbf{x}_n - \mathbf{m}_k) \right\}. \tag{1.61}$$

Therefore, it is clear that the optimal solutions for $q^*(\mathbf{Z})$ and $q^*(\boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})$ are coupled and must be determined iteratively. The fact that we are dealing with a model in which each full conditional is in the exponential family, ensures that the variational factors take the same distributions as the corresponding conjugate priors. For this reason, the variational updates reduce to updating parameters in the distributions of interest.

In this way, CAVI implicates cycling between two stages analogous to the E and M steps of the maximum likelihood EM algorithm, as shown in Algorithm 3. In the variational equivalent of the E step, we evaluate the responsibilities (1.61) by means of the expectations (1.58), (1.59), (1.60). Then in the ensuing variational equivalent of the M step, fixed these responsibilities, we first compute the statistics (1.45), (1.46), (1.47) and then update the variational parameters (1.52), (1.54), (1.55), (1.56), (1.57).

It is also possible to evaluate the expected values of the mixing coefficients in the posterior distribution that, using the Dirichlet mean, are given by

$$\mathbb{E}[\pi_k|\mathbf{X}] = \frac{\alpha_0 + N_k}{K\alpha_0 + N}. \tag{1.62}$$

Consider a component $k$ that takes practically no responsibility for explaining the data points, which means $r_{nk} \simeq 0$ and thus $N_k \simeq 0$. Then, if the prior over the mixing coefficients is broad so that $\alpha_0 \to 0$, we have $\mathbb{E}[\pi_k|\mathbf{X}] \to 0$ and that component plays no role in the model. If on the other hand the prior imposes a tight constraint so that $\alpha_0 \to \infty$, then $\mathbb{E}[\pi_k|\mathbf{X}] \to 1/K$.

**Algorithm 3:** CAVI for the Bayesian Gaussian mixture model

**Initialize:** $\alpha_k, \beta_k, \mathbf{m}_k, \mathbf{W}_k^{-1}, \nu_k$

**while** ELBO *has not converged* **do**

Optimize $q(\mathbf{Z})$:

$$r_{nk} \propto \widetilde{\pi}_k \, \widetilde{\mathbf{\Lambda}}_k \, \exp\left\{ -\frac{D}{2\beta_k} - \frac{\nu_k}{2}(\mathbf{x}_n - \mathbf{m}_k)^T \mathbf{W}_k \, (\mathbf{x}_n - \mathbf{m}_k) \right\}$$

Optimize $q(\boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})$:

$$\alpha_k \leftarrow \alpha_0 + N_k$$
$$\beta_k \leftarrow \beta_0 + N_k$$
$$\mathbf{m}_k \leftarrow \frac{1}{\beta_k}\left(\beta_0 \mathbf{m}_0 + N_k \overline{\mathbf{x}}_k\right)$$
$$\mathbf{W}_k^{-1} \leftarrow \mathbf{W}_0^{-1} + N_k \mathbf{S}_k + \frac{\beta_0 N_k}{\beta_0 + N_k}\left(\overline{\mathbf{x}}_k - \mathbf{m}_0\right)\left(\overline{\mathbf{x}}_k - \mathbf{m}_0\right)^T$$
$$\nu_k \leftarrow \nu_0 + N_k$$

Evaluate ELBO$(q)$

**end**

**return** $r_{nk}, \alpha_k, \beta_k, \mathbf{m}_k, \mathbf{W}_k^{-1}, \nu_k$

At this point, in order to carry out Algorithm 3, we only need the evaluation of the lower bound. From definition (1.6) it can be written as

$$\begin{aligned} \text{ELBO}(q) &= \sum_{\mathbf{Z}} \iiint q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda}) \log\left\{ \frac{p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})}{q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})} \right\} d\boldsymbol{\pi} \, d\boldsymbol{\mu} \, d\mathbf{\Lambda} \\ &= \mathbb{E}[\log p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})] + \mathbb{E}[\log p(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})] - \mathbb{E}[\log q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})] \\ &= \mathbb{E}[\log p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})] - \text{KL}\left(q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda}) \,||\, p(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})\right), \end{aligned} \tag{1.63}$$

where all expectations are taken with respect to $q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \mathbf{\Lambda})$. The first term of (1.63) is an expected likelihood, while the second term is the negative divergence between the variational density and the prior, which acts as a regularizer. In this way, the variational objective mirrors the balance in a Bayesian model between fit and complexity, in which complexity of the model arises from components whose variational parameters are pushed away from their prior values.

The lower bound can be further developed to give

$$\begin{aligned}\text{ELBO}(q) = \mathbb{E}[\log p(\mathbf{X}|\mathbf{Z},\boldsymbol{\mu},\boldsymbol{\Lambda})] + \mathbb{E}[\log p(\mathbf{Z}|\boldsymbol{\pi})] + \mathbb{E}[\log p(\boldsymbol{\pi})] + \mathbb{E}[\log p(\boldsymbol{\mu},\boldsymbol{\Lambda})] \\ - \mathbb{E}[\log q(\mathbf{Z})] - \mathbb{E}[\log q(\boldsymbol{\pi})] - \mathbb{E}[\log q(\boldsymbol{\mu},\boldsymbol{\Lambda})].\end{aligned} \tag{1.64}$$

These terms are calculated to give

$$\begin{aligned}\mathbb{E}[\log p(\mathbf{X}|\mathbf{Z},\boldsymbol{\mu},\boldsymbol{\Lambda})] &= \sum_{n=1}^{N}\sum_{k=1}^{K}\mathbb{E}[z_{nk}]\;\mathbb{E}[\log\mathcal{N}\left(\mathbf{x}_n|\boldsymbol{\mu}_k,\boldsymbol{\Lambda}_k^{-1}\right)] \\ &= \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\big\{-D\log 2\pi + \log\widetilde{\boldsymbol{\Lambda}}_k - D\beta_k^{-1} - \nu_k(\mathbf{x}_n-\mathbf{m}_k)^T\mathbf{W}_k(\mathbf{x}_n-\mathbf{m}_k)\big\} \\ &\overset{(A.18)}{=} \frac{1}{2}\sum_{k=1}^{K} N_k\Big\{\log\widetilde{\boldsymbol{\Lambda}}_k - D\beta_k^{-1} - \nu_k\,\mathrm{Tr}\left(\mathbf{S}_k\mathbf{W}_k\right) \\ &\qquad - \nu_k(\overline{\mathbf{x}}_k-\mathbf{m}_k)^T\mathbf{W}_k(\overline{\mathbf{x}}_k-\mathbf{m}_k) - D\log 2\pi\Big\}\end{aligned} \tag{1.65}$$

$$\mathbb{E}[\log p(\mathbf{Z}|\boldsymbol{\pi})] = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\log\widetilde{\pi}_k \tag{1.66}$$

$$\mathbb{E}[\log p(\boldsymbol{\pi})] = \log C(\boldsymbol{\alpha}_0) + (\alpha_0-1)\sum_{k=1}^{K}\log\widetilde{\pi}_k \tag{1.67}$$

$$\begin{aligned}\mathbb{E}[\log p(\boldsymbol{\mu},\boldsymbol{\Lambda})] = \frac{1}{2}\sum_{k=1}^{K}\Big\{D\log\left(\frac{\beta_0}{2\pi}\right) + \log\widetilde{\boldsymbol{\Lambda}}_k - \frac{D\beta_0}{\beta_k} \\ - \beta_0\,\nu_k\,(\mathbf{m}_k-\mathbf{m}_0)^T\mathbf{W}_k(\mathbf{m}_k-\mathbf{m}_0)\Big\} + K\log B(\mathbf{W}_0,\nu_0) \\ + \frac{(\nu_0-D-1)}{2}\sum_{k=1}^{K}\log\widetilde{\boldsymbol{\Lambda}}_k - \frac{1}{2}\sum_{k=1}^{K}\nu_k\,\mathrm{Tr}\left(\mathbf{W}_0^{-1}\mathbf{W}_k\right)\end{aligned} \tag{1.68}$$

$$\mathbb{E}[\log q(\mathbf{Z})] = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\log\,r_{nk} \tag{1.69}$$

$$\mathbb{E}[\log q(\boldsymbol{\pi})] = \sum_{k=1}^{K}(\alpha_k-1)\log\widetilde{\pi}_k + \log C(\boldsymbol{\alpha}) \tag{1.70}$$

$$\mathbb{E}[\log q(\boldsymbol{\mu},\boldsymbol{\Lambda})] = \sum_{k=1}^{K}\Big\{\frac{1}{2}\log\widetilde{\boldsymbol{\Lambda}}_k + \frac{D}{2}\log\left(\frac{\beta_k}{2\pi}\right) - \frac{D}{2} - \mathbb{H}[\boldsymbol{\Lambda}_k]\Big\}, \tag{1.71}$$

where $C(\boldsymbol{\alpha})$ and $B(\mathbf{W},\nu)$ are normalization constants of the Dirichlet and Wishart distributions respectively defined in (A.6) and (A.30), while $\mathbb{H}[\boldsymbol{\Lambda}_k]$ is the entropy of the Wishart given by (A.33).

In many circumstances it is also useful to evaluate the posterior *predictive density* for a new observation $\widehat{\mathbf{x}}$, which is

$$p(\widehat{\mathbf{x}}|\mathbf{X}) = \sum_{\widehat{\mathbf{z}}} \iiint p(\widehat{\mathbf{x}}|\widehat{\mathbf{z}}, \boldsymbol{\mu}, \boldsymbol{\Lambda})\, p(\widehat{\mathbf{z}}|\boldsymbol{\pi})\, p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{X})\; d\boldsymbol{\pi}\, d\boldsymbol{\mu}\, d\boldsymbol{\Lambda}, \tag{1.72}$$

where $\widehat{\mathbf{z}}$ is the latent variable associated with $\widehat{\mathbf{x}}$ and $p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{X})$ the (unknown) true posterior distribution of the parameters. According to (Bishop, 2006) and using result (A.26), an approximation of the predictive density is given by a mixture of (multivariate) Student's t-distributions

$$p(\widehat{\mathbf{x}}|\mathbf{X}) \simeq \frac{1}{\widehat{\alpha}} \sum_{k=1}^{K} \alpha_k \,\mathrm{St}(\widehat{\mathbf{x}} \,|\, \mathbf{m}_k, \mathbf{L}_k, \nu_k + 1 - D), \qquad \mathbf{L}_k = \frac{(\nu_k + 1 - D)\beta_k}{(1 + \beta_k)}\, \mathbf{W}_k \tag{1.73}$$

where $\widehat{\alpha} = \sum_{k=1}^{K} \alpha_k$ and the $k$-th component has mean $\mathbf{m}_k$, precision $\mathbf{L}_k$ and $\nu_k + 1 - D$ degrees of freedom. Clearly, when the size $N$ of the dataset is large, the predictive distribution reduces to a mixture of Gaussians.

### 1.2.3 Simulation studies

We now demonstrate the Gaussian mixture model in action by means of two simulations studies. The first one illustrates the variational Bayesian estimation and the second one compares it with the EM algorithm.

Firstly, we generate a synthetic dataset by simulating two-dimensional observations from $K = 3$ different Gaussians with fixed means and covariances. These data are shown in Figure 1.5, where each observation is colored according to its true cluster assignment, that is the actual Gaussian which generated it. The subplots illustrate the evolution of the variational densities for the mixture components, as the CAVI algorithm progresses, from random initialization until convergence.

The corresponding progression of the ELBO is plotted in Figure 1.6. We observe that the curve develops "elbows", which occur because the ELBO is generally a non-convex objective function. Indeed CAVI only guarantees convergence to a local optimum, which can be sensitive to initialization. For this experiment the responsibilities are randomly initialized, but it is common practice to initialize them through the K-means algorithm.

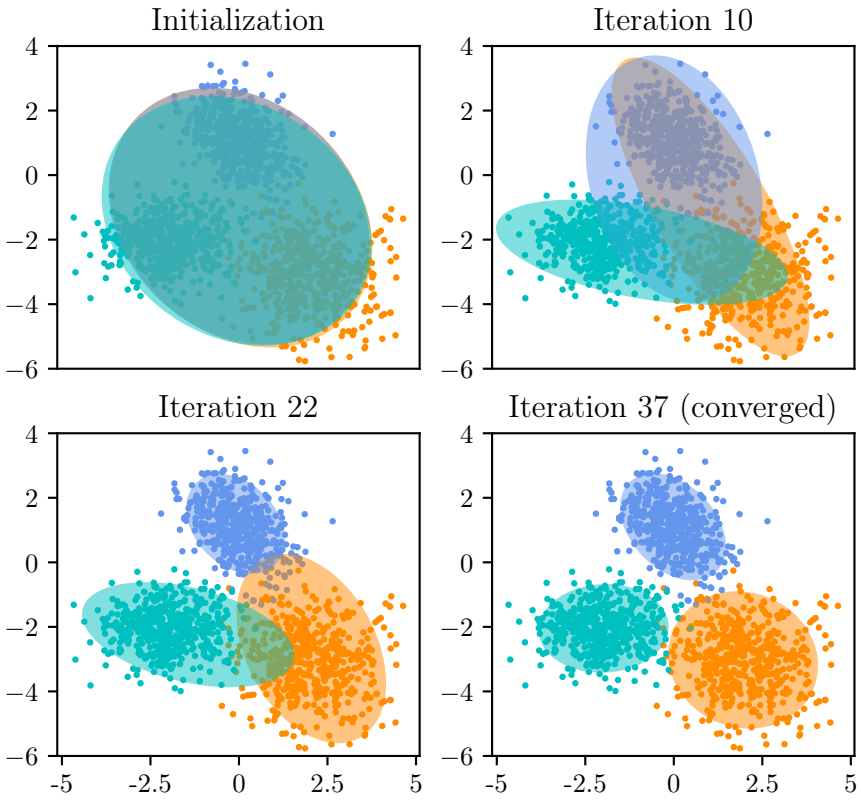



**Figure 1.5:** Variational Bayesian Gaussian mixture model applied to a synthetic dataset. The ellipses denote the two-standard deviations density contours for each of the mixture components, as the CAVI algorithm progresses.

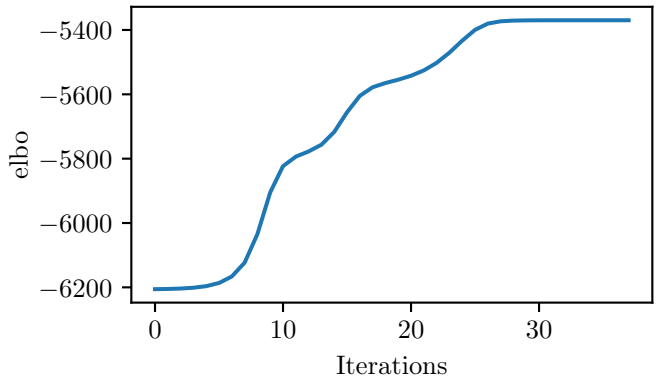



**Figure 1.6:** Progression of the evidence lower bound for the CAVI algorithm in Figure 1.5

.

In the second simulation study, we generate a toy dataset composed of $K = 2$ clusters, obtained by simulating two-dimensional observations from two Gaussians with different means and covariances. The purpose of the experiment is to compare the variational Bayesian estimation and the maximum likelihood EM algorithm in fitting a Gaussian mixture model. Both methods have access to $K = 4$ mixture components, number that does not match the true generative distribution of the dataset. Results are shown in Figure 1.7.

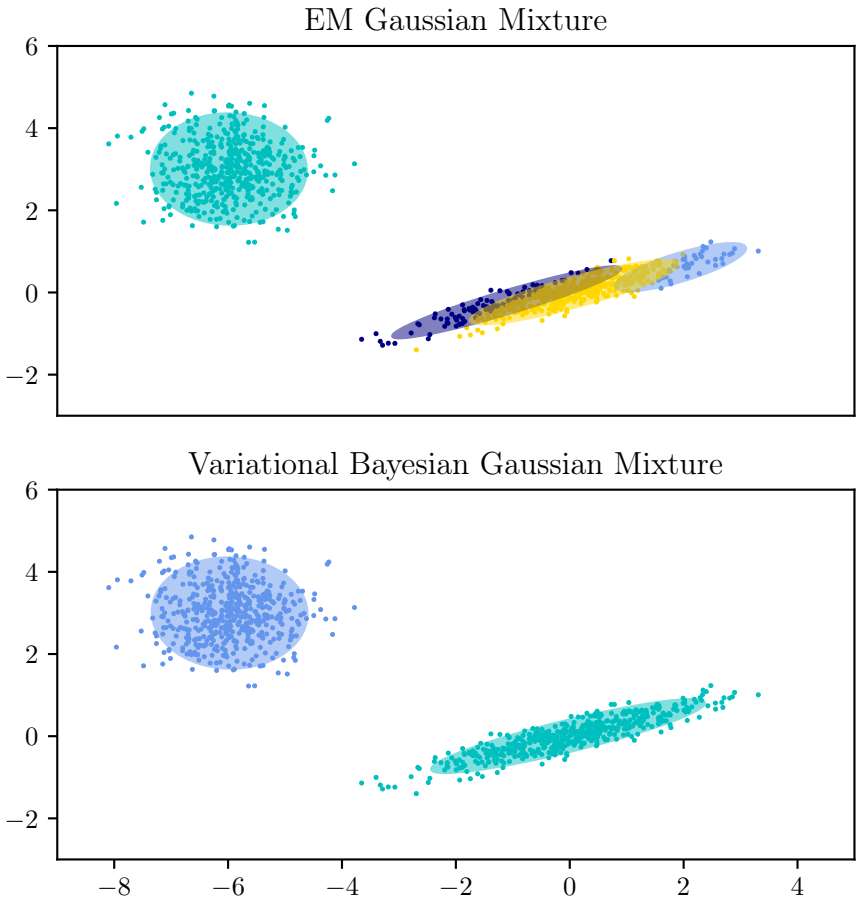



**Figure 1.7:** Comparison between EM and CAVI algorithms in fitting a Gaussian mixture model with $K = 4$ components to a synthetic dataset. Each data point is colored according to the mixture component that maximizes the responsibility $r_{nk}$ for that specific observation. The ellipses denote the $2\sigma$ variational contours; components to which no observation is assigned are not plotted.

Observe that the EM algorithm model uses all four input components making unnecessary splits, because it is trying to fit a number of components that is greater than the actual one. On the other hand, the variational Bayesian model is able to limit itself by assigning observations to only two of the components. Indeed, as we have seen from equation (1.62), this method has a natural tendency to set some mixture weights values close to zero.[2] This allows the model to adapt its effective number of components automatically starting from a provided upper bound.

In general, as seen in Algorithm 3, CAVI reflects the same steps as the maximum likelihood EM algorithm with the addition of prior information. In fact, when the dataset size $N \to \infty$, the prior effect vanishes and the variational Bayesian approach converges to the maximum likelihood EM algorithm. Therefore, for finite-size datasets, the variational method additionally involves only computations concerning the extra prior parameterization, which makes inference slightly slower.

However, the Bayesian approach has substantial advantages over the maximum likelihood one. First of all, the Bayesian treatment allows to remove the singularities that often arise in maximum likelihood solutions. This is possible because appropriate priors induce regularization, at the cost of introducing some subtle biases to the model. Moreover, there is no overfitting if one chooses a number $K$ of components too large: the Dirichlet prior over $\boldsymbol{\pi}$ automatically encourages a sparse posterior mixing weight vector.

A slower alternative to adapt an effective number of components is to fit multiple models considering several values of K and then select the number of components that maximizes the ELBO (or the predictive density 1.73). However, the mixture of Gaussians presents the so-called *label switching* problem: permuting cluster labels induces $K!$ symmetric posterior modes. The issue is also known as *unidentifiability* (Casella & Berger, 2002). That does not affect a suitable variational Bayesian model because reaching one of the $K!$ modes leads to a solution that is equivalent to the others $K! - 1$.

[2] Variational hyperparameters values used for this experiment are
$\alpha_0 = 1/K = 0.25,\ \beta_0 = 1,\ \mathbf{m}_0 = \mathbf{0},\ \mathbf{W}_0 = \mathbf{I}_2,\ \nu_0 = D = 2.$

Nevertheless, if we want to compare different values of $K$, we must take this multimodality into consideration. A valid approximate solution is to add a term $\log K!$ onto the lower bound when used for model comparison and averaging (Bishop, 2006).

### 1.2.4 Image analysis

Finally, the variational Bayesian Gaussian mixture can be applied to a dataset of images. We work with the STL-10 dataset http://cs.stanford.edu/~acoates/stl10 (Coates et al., 2011), which contains 100 000 unlabeled images and 13 000 labeled images from 10 classes.

Inspired by (Blei et al., 2017), we consider the task of grouping images according to their color profiles. The approach followed is based on computing the color histograms of the images, which represent the distribution of pixel intensities for *red*, *green* and *blue* color channels. Figure 1.8 shows the RGB color histograms for a sample image, where the pixel intensities are divided into 64 equal bins for each channel.

The counts of the three histograms are concatened to provide a $D = 192$ dimensional representation of each image, regardless of its resolution (in this case $96 \times 96$ pixels). We use these data to fit a Bayesian Gaussian mixture with $K = 30$ clusters via CAVI, randomly selecting $N = 10\,000$ unlabeled images from STL-10. Figure 1.9 shows images with similar color profiles representing four selected clusters. Each image is assigned to the cluster that maximize its responsibility: the images displayed are the nine with the highest responsibility within the specific cluster.

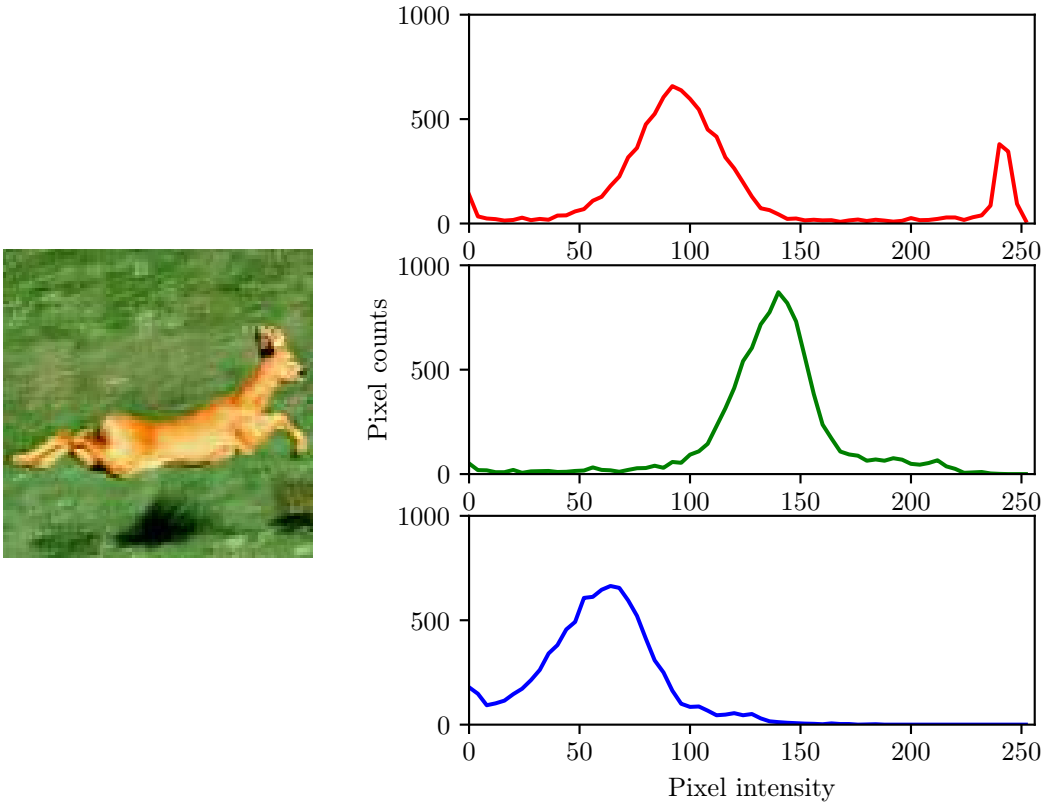
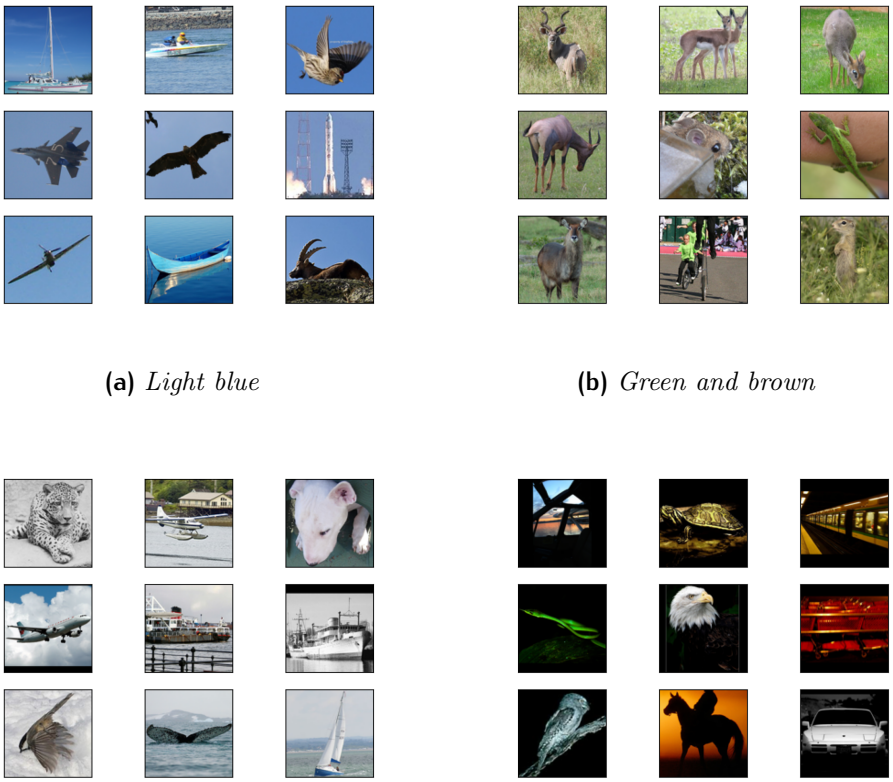



**Figure 1.8:** RGB color histograms for a sample image from STL-10 dataset; each histogram is constructed from the same 64 bins of the pixel intensities (range 0-255). The image is characterized by red and green hues, as shown by the histograms.



**(a)** *Light blue* **(b)** *Green and brown*

**(c)** *White grey* **(d)** *Dark*

**Figure 1.9:** Most representative images from four selected clusters. Each image is assigned to its most likely mixture component. Namings of the clusters are subjective since we are performing unsupervised learning.

# Chapter 2

# Beyond Vanilla Variational Inference

Classical variational inference presented so far has limitations when it comes to modern applications, that involve large datasets and complex models. In order to tackle big datasets, we first derive the general CAVI algorithm in the case of *conditionally conjugate exponential family* models and then we show an extension that scales up VI to massive data in this setting. Further novel methods are finally addressed in order to make VI generically applicable to non-conjugate models.

## 2.1 CAVI for conditionally conjugate models

To begin with, we are interested in a class of models called *conditionally conjugate* models, which involve latent variables that are *local* to a data point and latent variables that are *global* to the entire dataset (usually regarded as parameters). Let $\mathbf{x} = x_{1:N}$ be the $N$ observations, $\mathbf{z} = z_{1:N}$ the corresponding local latent variables and $\boldsymbol{\vartheta}$ the global latent variables. The generic joint distribution of a conditionally conjugate model factorizes into a global term and a product of local terms:

$$p(\mathbf{x}, \mathbf{z}, \boldsymbol{\vartheta}) = p(\boldsymbol{\vartheta}) \prod_{n=1}^{N} p(x_n, z_n | \boldsymbol{\vartheta}). \tag{2.1}$$

The graphical model is illustrated in Figure 2.1. Observe that each observation $x_n$ and each local variable $z_n$ are, given $\boldsymbol{\vartheta}$, conditionally independent of

all the other observations $\mathbf{x}_{-n}$ and local variables $\mathbf{z}_{-n}$.

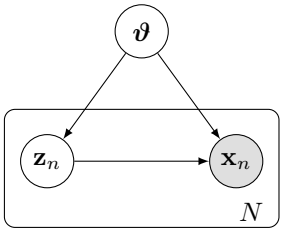



**Figure 2.1:** Graphical representation of a conditionally conjugate model with observations $x_{1:N}$, local latent variables $z_{1:N}$ and global latent variables $\boldsymbol{\vartheta}$. The distribution of each data point $x_n$ only depends on its local variable $z_n$ and the global parameters $\boldsymbol{\vartheta}$.

Standard Bayesian models fall into this class. An example is the Bayesian mixture of Gaussians, where the global variables are the parameters $\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}$ associated to the mixture components and the $n$-th local variable $\mathbf{z}_n$ is the hidden cluster assignment for the observation $\mathbf{x}_n$. Working within this family of models allows us to derive a closed-form CAVI algorithm in a general manner (Blei et al., 2017).

The underlying assumption is that the joint density of $(x_n, z_n)$ given $\boldsymbol{\vartheta}$ has an exponential family form

$$p(x_n, z_n | \boldsymbol{\vartheta}) = h(x_n, z_n) \exp\{\eta(\boldsymbol{\vartheta})^T t(x_n, z_n) - a_\ell(\boldsymbol{\vartheta})\}, \tag{2.2}$$

where $\eta(\cdot)$ is the *natural parameter*, $t(\cdot)$ the *sufficient statistic*, $h(\cdot)$ the *base measure* and $a(\cdot)$ the *log-normalizer*.[1] We can thus write the *complete* likelihood function as

$$\mathcal{L}(\boldsymbol{\vartheta}; \mathbf{x}, \mathbf{z}) = \prod_{n=1}^{N} h(x_n, z_n) \exp\left\{ \eta(\boldsymbol{\vartheta})^T \sum_{n=1}^{N} t(x_n, z_n) - N\, a_\ell(\boldsymbol{\vartheta}) \right\}. \tag{2.3}$$

The prior on the global variables is assumed to be closed under sampling (Bernardo & Smith, 2000):

$$p(\boldsymbol{\vartheta}) = \exp\{\, \boldsymbol{\zeta}^T [\, \eta(\boldsymbol{\vartheta}), -a_\ell(\boldsymbol{\vartheta})] - a_g(\boldsymbol{\zeta})\}, \tag{2.4}$$

that is the conjugate prior with hyperparameters $\boldsymbol{\zeta} = [\, \zeta_1, \zeta_2 \,]^T$.[2]

[1] the subscript $\ell$ indicates that we refer to local variables

[2] the subscript $g$ stands for "global variables"

At this point, the conjugacy leads the full conditional of the global variables to be in the same family of the corresponding prior, namely

$$p(\boldsymbol{\vartheta}|\,\mathbf{x},\mathbf{z}) = \exp\{\,\hat{\boldsymbol{\zeta}}^{\,T}\,[\,\eta(\boldsymbol{\vartheta}), -a_\ell(\boldsymbol{\vartheta})] - a_g(\hat{\boldsymbol{\zeta}})\}, \quad \text{with} \tag{2.5}$$

$$\hat{\boldsymbol{\zeta}} = \left[\zeta_1 + \sum_{n=1}^{N} t(x_n, z_n),\ \zeta_2 + N\right]^T. \tag{2.6}$$

Consider now the full conditionals of the local variables. From Figure 2.1 we have seen that $z_n$ is conditionally independent, given $\boldsymbol{\vartheta}$ and $x_n$, of the other local variables $\mathbf{z}_{-n}$ and observations $\mathbf{x}_{-n}$. Then, as a property of (2.2), it follows that the full conditional of $z_n$ has an exponential family form

$$p(z_n|x_n,\boldsymbol{\vartheta}) = h(z_n)\exp\{\eta(\boldsymbol{\vartheta},x_n)^T z_n - a_\ell(\eta(\boldsymbol{\vartheta},x_n))\}. \tag{2.7}$$

With the probabilistic structure just described, we can now obtain the general CAVI algorithm for what we call *conditionally conjugate exponential family models*. Consider the mean field approximation

$$q(\mathbf{z},\boldsymbol{\vartheta}) = q(\boldsymbol{\vartheta}|\boldsymbol{\lambda})\prod_{n=1}^{N} q(z_n|\phi_n). \tag{2.8}$$

Contrary to what was done in the previous chapter, we make the parameters explicit in the variational distributions. The global variational parameter $\boldsymbol{\lambda}$ indexes the posterior approximation on $\boldsymbol{\vartheta}$, while the local variational parameters $\phi_n$ govern the approximations of the respective local variables $z_n$.

With result (2.7), the coordinate update of equation (1.13) for the local hidden variable $z_n$ becomes

$$\begin{aligned} q^*(z_n|\phi_n) &\propto \exp\left\{\mathbb{E}_{\boldsymbol{\lambda}}\left[\log p(z_n|x_n,\boldsymbol{\vartheta})\right]\right\} \\ &= \exp\{\log h(z_n) + \mathbb{E}_{\boldsymbol{\lambda}}\left[\eta(\boldsymbol{\vartheta},x_n)\right]^T z_n - \mathbb{E}_{\boldsymbol{\lambda}}\left[a_\ell(\eta(\boldsymbol{\vartheta},x_n))\right]\} \\ &\propto h(z_n)\exp\{\,\mathbb{E}_{\boldsymbol{\lambda}}\left[\eta(\boldsymbol{\vartheta},x_n)\right]^T z_n\}. \end{aligned} \tag{2.9}$$

This optimal variational factor takes on the same exponential family form as its corresponding full conditional. Consequently, the local variational update simply consists of setting $\phi_n$ equal to the expected natural parameter of its full conditional,

$$\phi_n = \mathbb{E}_{\boldsymbol{\lambda}}\left[\eta(\boldsymbol{\vartheta},x_n)\right]. \tag{2.10}$$

The same reasoning leads to the global variational updates

$$\boldsymbol{\lambda} = \left[ \zeta_1 + \sum_{n=1}^{N} \mathbb{E}_{\phi_n}[t(x_n, z_n)],\ \zeta_2 + N \right]^T, \tag{2.11}$$

which is obtained taking the expectation of (2.6).

CAVI iterates between local updates of each local variational parameter (2.10) and global updates of the global variational parameters (2.11). To assess convergence, it is possible to compute the general form of the ELBO. By means of equation (1.6):

$$\begin{aligned} \text{ELBO}(\boldsymbol{\lambda}, \boldsymbol{\phi}) &= \mathbb{E}_{\boldsymbol{\lambda},\boldsymbol{\phi}} \left[\log p(\mathbf{x}, \mathbf{z}, \boldsymbol{\vartheta})\right] - \mathbb{E}_{\boldsymbol{\lambda},\boldsymbol{\phi}} \left[\log q(\mathbf{z}, \boldsymbol{\vartheta})\right] \\ &= \mathbb{E}_{\boldsymbol{\phi}}[\hat{\boldsymbol{\zeta}}]^T \, \mathbb{E}_{\boldsymbol{\lambda}}[\, \eta(\boldsymbol{\vartheta}), -a_\ell(\boldsymbol{\vartheta})] - \boldsymbol{\lambda}^T \, \mathbb{E}_{\boldsymbol{\lambda}}[\, \eta(\boldsymbol{\vartheta}), -a_\ell(\boldsymbol{\vartheta})] \\ &\quad + a_g(\boldsymbol{\lambda}) - \sum_{n=1}^{N} \phi_n^T \, \mathbb{E}_{\phi_n}[z_n] + a_\ell(\phi_n) + \text{const}, \end{aligned} \tag{2.12}$$

where we used equation (2.5) together with the optimal densities forms derived from (2.8); the constant absorbs all terms that do not depend on the variational parameters.

### 2.1.1 GMM example and comparison to Gibbs sampling

Algorithm 3 for the Bayesian Gaussian mixture model is an instance of the procedure described. In fact, relying on notation and formulas of Section 1.2, we can write down the joint density of $(\mathbf{x}_n, \mathbf{z}_n)$ given $\boldsymbol{\vartheta} = (\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})^T$ as

$$\begin{aligned} p(\mathbf{x}_n, \mathbf{z}_n | \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) &= \exp\left\{ \sum_{k=1}^{K} z_{nk} \left\{ \log \pi_k + \log \mathcal{N}\big(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}\big) \right\} \right\} \\ &\propto \exp\Bigg\{ \sum_{k=1}^{K} z_{nk} \Big( \log \pi_k + \frac{1}{2} \log|\boldsymbol{\Lambda}_k| - \frac{1}{2} \boldsymbol{\mu}_k^T \, \boldsymbol{\Lambda}_k \, \boldsymbol{\mu}_k \\ &\qquad - \frac{1}{2} \operatorname{Tr}\left(\mathbf{x}_n^T \, \boldsymbol{\Lambda}_k \, \mathbf{x}_n\right) + \operatorname{Tr}\left(\boldsymbol{\mu}_k^T \, \boldsymbol{\Lambda}_k \, \mathbf{x}_n\right) \Big) \Bigg\} \\ &= \exp\Bigg\{ \sum_{k=1}^{K} z_{nk} \Big( \log \pi_k + \frac{1}{2} \log|\boldsymbol{\Lambda}_k| - \frac{1}{2} \boldsymbol{\mu}_k^T \, \boldsymbol{\Lambda}_k \, \boldsymbol{\mu}_k \\ &\qquad - \frac{1}{2} \mathbf{x}_n \, \mathbf{x}_n^T \, \boldsymbol{\Lambda}_k + \mathbf{x}_n \, \boldsymbol{\mu}_k^T \, \boldsymbol{\Lambda}_k \Big) \Bigg\}. \end{aligned} \tag{2.13}$$

We recognize this distribution to be in the form (2.2), with $a_\ell(\boldsymbol{\vartheta}) = 0$,

$$\eta(\boldsymbol{\vartheta}) = \begin{bmatrix} \log \pi_k \\ -\frac{1}{2}\,\boldsymbol{\mu}_k^T\,\boldsymbol{\Lambda}_k\,\boldsymbol{\mu}_k \\ \frac{1}{2}\log|\boldsymbol{\Lambda}_k| \\ \boldsymbol{\mu}_k^T\boldsymbol{\Lambda}_k \\ -\frac{1}{2}\,\mathrm{vec}(\boldsymbol{\Lambda}_k) \end{bmatrix} \quad t(\mathbf{x}_n, \mathbf{z}_n) = \begin{bmatrix} z_{nk} \\ z_{nk} \\ z_{nk} \\ z_{nk}\,\mathbf{x}_n \\ z_{nk}\,\mathrm{vec}(\mathbf{x}_n\mathbf{x}_n^T) \end{bmatrix} \quad \text{for } k = 1, \ldots, K, \tag{2.14}$$

where the $\mathrm{vec}(\cdot)$ operator simply stacks the columns of the argument matrix on top of each other.

The conjugate prior over the parameters $\boldsymbol{\vartheta}$ can then be written as

$$\begin{aligned}
p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) &= C(\boldsymbol{\alpha}_0) \prod_{k=1}^{K} \pi_k^{\alpha_0 - 1} \mathcal{N}\left(\boldsymbol{\mu}_k | \mathbf{m}_0, (\beta_0\,\boldsymbol{\Lambda}_k)^{-1}\right) \mathcal{W}\left(\boldsymbol{\Lambda}_k | \mathbf{W}_0, \nu_0\right) \\
&\propto \exp\Bigg\{ \sum_{k=1}^{K} \Bigg[ (\alpha_0 - 1)\log \pi_k + \frac{(\nu_0 - D - 1)}{2}\log|\boldsymbol{\Lambda}_k| + \frac{1}{2}\log|\boldsymbol{\Lambda}_k| \\
&\qquad - \frac{1}{2}\,\mathrm{Tr}\left(\mathbf{W}_0^{-1}\boldsymbol{\Lambda}_k\right) - \frac{1}{2}\,\beta_0\,\mathrm{Tr}\left((\boldsymbol{\mu}_k - \mathbf{m}_0)^T\,\boldsymbol{\Lambda}_k\,(\boldsymbol{\mu}_k - \mathbf{m}_0)\right) \Bigg] \Bigg\} \\
&= \exp\Bigg\{ \sum_{k=1}^{K} \Bigg[ (\alpha_0 - 1)\log \pi_k + \frac{(\nu_0 - D)}{2}\log|\boldsymbol{\Lambda}_k| \\
&\qquad - \frac{1}{2}\,\mathrm{Tr}\left(\boldsymbol{\Lambda}_k\left(\beta_0\,(\boldsymbol{\mu}_k - \mathbf{m}_0)\,(\boldsymbol{\mu}_k - \mathbf{m}_0)^T + \mathbf{W}_0^{-1}\right)\right) \Bigg] \Bigg\} \\
&= \exp\Bigg\{ \sum_{k=1}^{K} \Bigg[ (\alpha_0 - 1)\log \pi_k + \frac{(\nu_0 - D)}{2}\log|\boldsymbol{\Lambda}_k| - \frac{1}{2}\,\beta_0\,\boldsymbol{\mu}_k^T\,\boldsymbol{\Lambda}_k\,\boldsymbol{\mu}_k \\
&\qquad + \beta_0\,\mathbf{m}_0\,\boldsymbol{\mu}_k^T\,\boldsymbol{\Lambda}_k - \frac{1}{2}\left(\beta_0\,\mathbf{m}_0\,\mathbf{m}_0^T + \mathbf{W}_0^{-1}\right)\boldsymbol{\Lambda}_k \Bigg] \Bigg\}.
\end{aligned} \tag{2.15}$$

We recognize this density to have the same form as (2.4), where

$$\boldsymbol{\zeta} = \begin{bmatrix} \alpha_0 - 1 \\ \beta_0 \\ \nu_0 - D \\ \beta_0\,\mathbf{m}_0 \\ \mathrm{vec}(\beta_0\,\mathbf{m}_0\,\mathbf{m}_0^T + \mathbf{W}_0^{-1}) \end{bmatrix} \quad \eta(\boldsymbol{\vartheta}) = \begin{bmatrix} \log \pi_k \\ -\frac{1}{2}\,\boldsymbol{\mu}_k^T\,\boldsymbol{\Lambda}_k\,\boldsymbol{\mu}_k \\ \frac{1}{2}\log|\boldsymbol{\Lambda}_k| \\ \boldsymbol{\mu}_k^T\boldsymbol{\Lambda}_k \\ -\frac{1}{2}\,\mathrm{vec}(\boldsymbol{\Lambda}_k) \end{bmatrix}_{k=1,\ldots,K} \tag{2.16}$$

At this point the conjugacy between (2.13) and (2.15) allows us to derive the global variational updates as

$$\boldsymbol{\lambda} = \mathbb{E}_{\boldsymbol{\phi}}[\hat{\boldsymbol{\zeta}}] = \boldsymbol{\zeta} + \sum_{n=1}^{N} \mathbb{E}_{\phi_n}[t(\mathbf{x}_n, \mathbf{z}_n)]. \tag{2.17}$$

By substituting, we obtain

$$\begin{cases} \alpha_k - 1 = \alpha_0 - 1 + \sum_{n=1}^{N} \mathbb{E}[z_{nk}] \\ \beta_k = \beta_0 + \sum_{n=1}^{N} \mathbb{E}[z_{nk}] \\ \nu_k - D = \nu_0 - D + \sum_{n=1}^{N} \mathbb{E}[z_{nk}] \\ \beta_k \, \mathbf{m}_k = \beta_0 \, \mathbf{m}_0 + \sum_{n=1}^{N} \mathbb{E}[z_{nk}] \, \mathbf{x}_n \\ \mathrm{vec}(\beta_k \, \mathbf{m}_k \, \mathbf{m}_k^T + \mathbf{W}_k^{-1}) = \mathrm{vec}(\beta_0 \, \mathbf{m}_0 \, \mathbf{m}_0^T + \mathbf{W}_0^{-1}) + \sum_{n=1}^{N} \mathbb{E}[z_{nk}] \, \mathrm{vec}(\mathbf{x}_n \, \mathbf{x}_n^T) \end{cases}$$

Defining $N_k$, $\overline{\mathbf{x}}_k$, $\mathbf{S}_k$ as in equations (1.45), (1.46), (1.47) gives

$$\begin{cases} \alpha_k = \alpha_0 + N_k \\ \beta_k = \beta_0 + N_k \\ \nu_k = \nu_0 + N_k \\ \mathbf{m}_k = \frac{1}{\beta_k} \left( \beta_0 \mathbf{m}_0 + N_k \overline{\mathbf{x}}_k \right) \\ \mathbf{W}_k^{-1} = \mathbf{W}_0^{-1} + N_k \mathbf{S}_k + \frac{\beta_0 N_k}{\beta_0 + N_k} \left( \overline{\mathbf{x}}_k - \mathbf{m}_0 \right) \left( \overline{\mathbf{x}}_k - \mathbf{m}_0 \right)^T \end{cases}$$

that are the same global variational updates obtained for Algorithm 3 in the previous chapter. The algebraic steps required to compute the last update can be found in (Murphy, 2007).

Instead, as for local variational updates, we can express the full conditional of the local latent variable $\mathbf{z}_n$ in the form (2.7) as follows:

$$\begin{aligned} p(z_{nk} = 1 \,|\, \mathbf{x}_n, \mathbf{z}_{-n}, \mathbf{x}_{-n}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) &= p(z_{nk} = 1 \,|\, \mathbf{x}_n, \pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) \\ &\propto p(z_{nk} = 1 \,|\, \pi_k) \, p(\mathbf{x}_n \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = \left[ \pi_k \, \mathcal{N}(\mathbf{x}_n \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}) \right]^{z_{nk}} \\ &= \exp\left\{ z_{nk} \left[ \log \pi_k + \log \mathcal{N}(\mathbf{x}_n \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}) \right] \right\} \\ &\propto \exp\left\{ \left[ \log \pi_k + \frac{1}{2} \log |\boldsymbol{\Lambda}_k| - \frac{1}{2} (\mathbf{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Lambda}_k (\mathbf{x}_n - \boldsymbol{\mu}_k) \right] z_{nk} \right\} \\ &\propto \exp\{ \, \eta(\boldsymbol{\vartheta}, \mathbf{x}_n)^T z_{nk} \, \}, \quad \text{with} \end{aligned} \tag{2.18}$$

$$\eta(\boldsymbol{\vartheta}, \mathbf{x}_n) = \log \pi_k + \frac{1}{2} \log |\boldsymbol{\Lambda}_k| - \frac{1}{2} (\mathbf{x}_n - \boldsymbol{\mu}_k)^T \boldsymbol{\Lambda}_k (\mathbf{x}_n - \boldsymbol{\mu}_k). \tag{2.19}$$

Since we have seen that each local variational update takes the form $\phi_{nk} = \mathbb{E}_{\boldsymbol{\lambda}}\left[\eta(\boldsymbol{\vartheta}, x_n)\right]$, we arrive at the same result as in formula (1.61) for the responsibilities' updates. We thus derived CAVI algorithm for the Bayesian GMM as a special case of conditionally conjugate exponential family models.

We now want to compare CAVI for this example to a corresponding (collapsed) Gibbs sampler. Taking advantage of the work already done, it is straightforward to derive the full conditionals that define Gibbs sampling. Specifically, the full conditionals for the hidden variables are categorical

$$p(z_{nk} = 1 \,|\, \mathbf{x}_n, \mathbf{z}_{-n}, \mathbf{x}_{-n}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) \propto \left[\pi_k \mathcal{N}(\mathbf{x}_n \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1})\right]^{z_{nk}}. \tag{2.20}$$

Similarly to (1.51), the full conditional of the mixing coefficients is

$$p(\boldsymbol{\pi}|\mathbf{Z}) = \mathrm{Dir}\big(\boldsymbol{\pi} \,|\, \{\alpha_0 + \widetilde{N}_k\}_{k=1}^K\big), \tag{2.21}$$

where $\widetilde{N}_k$ is the number of hidden variables that fall within the cluster $k$

$$\widetilde{N}_k = \sum_{n=1}^{N} z_{nk}. \tag{2.22}$$

For the precisions full conditionals, we can exploit the conjugacy between the Wishart prior and the normal (1.35) with known means, to obtain

$$p(\boldsymbol{\Lambda}_k|\boldsymbol{\mu}_k, \mathbf{X}, \mathbf{Z}) = \mathcal{W}\left(\boldsymbol{\Lambda}_k|\mathbf{V}_{\boldsymbol{\mu}_k}, \nu_k\right), \tag{2.23}$$

$$\mathbf{V}_{\boldsymbol{\mu}_k}^{-1} = \mathbf{W}_0^{-1} + \sum_{n=1}^{N} z_{nk}\left(\mathbf{x}_n - \boldsymbol{\mu}_k\right)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T, \quad \nu_k = \nu_0 + \widetilde{N}_k. \tag{2.24}$$

However, in practice we can directly sample the $(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)$ block from

$$p(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k|\mathbf{X}, \mathbf{Z}) = \mathcal{N}\left(\boldsymbol{\mu}_k|\mathbf{m}_k, (\beta_k\,\boldsymbol{\Lambda}_k)^{-1}\right)\,\mathcal{W}\left(\boldsymbol{\Lambda}_k|\mathbf{W}_k, \nu_k\right), \tag{2.25}$$

$$\beta_k = \beta_0 + \widetilde{N}_k \tag{2.26}$$

$$\mathbf{m}_k = \frac{1}{\beta_k}\left(\beta_0\,\mathbf{m}_0 + \widetilde{N}_k\,\widetilde{\mathbf{x}}_k\right) \tag{2.27}$$

$$\mathbf{W}_k^{-1} = \mathbf{W}_0^{-1} + \widetilde{N}_k\,\widetilde{\mathbf{S}}_k + \frac{\beta_0\,\widetilde{N}_k}{\beta_0 + \widetilde{N}_k}\left(\widetilde{\mathbf{x}}_k - \mathbf{m}_0\right)\left(\widetilde{\mathbf{x}}_k - \mathbf{m}_0\right)^T \tag{2.28}$$

$$\nu_k = \nu_0 + \widetilde{N}_k \tag{2.29}$$

$$\widetilde{\mathbf{x}}_k = \frac{1}{\widetilde{N}_k}\sum_{n=1}^{N} z_{nk}\,\mathbf{x}_n \tag{2.30}$$

$$\widetilde{\mathbf{S}}_k = \frac{1}{\widetilde{N}_k}\sum_{n=1}^{N} z_{nk}\left(\mathbf{x}_n - \widetilde{\mathbf{x}}_k\right)(\mathbf{x}_n - \widetilde{\mathbf{x}}_k)^T. \tag{2.31}$$

We thus presented a Gibbs sampler for the Bayesian GMM that consists of iteratively drawing from (2.20), (2.21) and (2.25).

As an aside, Gibbs sampling for mixture models is affected by what we called *label switching* problem: the posterior distribution is invariant to switching component labels.[3] Consequently, it is often inappropriate to compute posterior means just taking a Monte Carlo average of the samples, because what one sample considers the parameters for cluster 1 may be what another sample considers the parameters for cluster 2. Common strategies to deal with this issue are imposing identifiability constraints on the parameter space (valid only in the one-dimensional case) or post-processing the output to relabel the components through a permutation that minimizes some loss function (Stephens, 2000).

In the specific case of the mixture model, it is possible to integrate out the model parameters $\boldsymbol{\vartheta}$ and just sample the hidden variables $\mathbf{z}$. This is called a collapsed Gibbs sampler (J. S. Liu, 1994) and it tends to be more efficient, intuitively because it samples in a lower dimensional space. More specifically, marginalizing $\boldsymbol{\vartheta}$ parameters from the joint distribution of $(\mathbf{z}, \boldsymbol{\vartheta})$ reduces the variance of the samples. The process is called *Rao-Blackwellization* (Casella & Robert, 1996) as a result of the following theorem:

**Theorem 1** (Rao-Blackwell)**.** *Let $\boldsymbol{z}$ and $\boldsymbol{\vartheta}$ be dependent random variables, and $f(\boldsymbol{z}, \boldsymbol{\vartheta})$ be some scalar function. Then*

$$\begin{aligned} Var_{\boldsymbol{z},\boldsymbol{\vartheta}}\left[f(\boldsymbol{z},\boldsymbol{\vartheta})\right] &= Var_{\boldsymbol{z}}\left[\mathbb{E}_{\boldsymbol{\vartheta}}\left[f(\boldsymbol{z},\boldsymbol{\vartheta})|\boldsymbol{z}\right]\right] + \mathbb{E}_{\boldsymbol{z}}\left[Var_{\boldsymbol{\vartheta}}\left[f(\boldsymbol{z},\boldsymbol{\vartheta})|\boldsymbol{z}\right]\right] \\ &\geq Var_{\boldsymbol{z}}\left[\mathbb{E}_{\boldsymbol{\vartheta}}\left[f(\boldsymbol{z},\boldsymbol{\vartheta})|\boldsymbol{z}\right]\right]. \end{aligned} \tag{2.32}$$

The theorem guarantees that the variance of the estimate obtained by analytically integrating out $\boldsymbol{\vartheta}$ will never be higher than the variance of a direct Monte Carlo estimate. Therefore, when analytically possible, it is worth collapsing Gibbs sampling because it leads to faster convergence of the Markov chain to the stationary distribution, even if it is usually more costly per iteration than the standard one.

[3] We have seen that this problem does not affect the variational algorithm, which locks on to only one of the symmetric posterior modes.

Returning to the GMM example we can integrate out the model parameters $\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$, and only sample the local hidden variables $\mathbf{Z}$. Following (Murphy, 2023), we can write the full conditional

$$\begin{aligned} p(z_{nk}=1 \,|\, \mathbf{X}, \mathbf{z}_{-n}, \alpha_0, \boldsymbol{\gamma}_0) &\propto p(z_{nk}=1 \,|\, \mathbf{z}_{-n}, \alpha_0, \cancel{\boldsymbol{\gamma}_0})\, p(\mathbf{X} \,|\, z_{nk}=1, \mathbf{z}_{-n}, \cancel{\alpha_0}, \boldsymbol{\gamma}_0) \\ &\propto p(z_{nk}=1 \,|\, \mathbf{z}_{-n}, \alpha_0)\, p(\mathbf{x}_n \,|\, z_{nk}=1, \mathbf{x}_{-n}, \mathbf{z}_{-n}, \boldsymbol{\gamma}_0)\, p(\mathbf{x}_{-n} | \cancel{z_{nk}=1}, \mathbf{z}_{-n}, \boldsymbol{\gamma}_0) \\ &\propto p(z_{nk}=1 \,|\, \mathbf{z}_{-n}, \alpha_0)\, p(\mathbf{x}_n \,|\, z_{nk}=1, \mathbf{x}_{-n}, \mathbf{z}_{-n}, \boldsymbol{\gamma}_0), \end{aligned} \tag{2.33}$$

where $\boldsymbol{\gamma}_0 = (\beta_0, \mathbf{m}_0, \mathbf{W}_0, \nu_0)$ are the Normal-Wishart hyperparameters; we crossed out the variables according to the Markov blankets.

The first term of (2.33) arises by integrating out the mixing weights $\boldsymbol{\pi}$. Indeed we have

$$p(z_{nk}=1 \,|\, \mathbf{z}_{-n}, \alpha_0) = \frac{p(z_{nk}=1, \mathbf{z}_{-n}|\alpha_0)}{p(\mathbf{z}_{-n}|\alpha_0)} = \frac{p(\mathbf{Z}|\alpha_0)}{p(\mathbf{z}_{-n}|\alpha_0)}, \tag{2.34}$$

$$p(\mathbf{Z}|\alpha_0) = \int_{\boldsymbol{\pi}} p(\mathbf{Z}|\boldsymbol{\pi})\, p(\boldsymbol{\pi}|\alpha_0)\, d\boldsymbol{\pi}. \tag{2.35}$$

We can substitute (1.34) and (1.36) inside the integral to give

$$\begin{aligned} p(\mathbf{Z}|\alpha_0) &= \int_{\boldsymbol{\pi}} \prod_{k=1}^{K} \pi_k^{\widetilde{N}_k} C(\boldsymbol{\alpha}_0) \prod_{k=1}^{K} \pi_k^{\alpha_0 - 1} d\boldsymbol{\pi} = C(\boldsymbol{\alpha}_0) \int_{\boldsymbol{\pi}} \prod_{k=1}^{K} \pi_k^{\widetilde{N}_k + \alpha_0 - 1} d\boldsymbol{\pi} \\ &\overset{(A.6)}{=} \frac{\Gamma(K\alpha_0)}{\Gamma(\alpha_0)^K} \frac{\prod_{k=1}^{K} \Gamma(\widetilde{N}_k + \alpha_0)}{\Gamma(K\alpha_0 + N)} \\ &= \frac{\Gamma(K\alpha_0)}{\Gamma(K\alpha_0 + N)} \prod_{k=1}^{K} \frac{\Gamma(\widetilde{N}_k + \alpha_0)}{\Gamma(\alpha_0)}. \end{aligned} \tag{2.36}$$

From here we can obtain the term (2.34) as

$$\begin{aligned} &p(z_{nk}=1 \,|\, \mathbf{z}_{-n}, \alpha_0) = \\ &= \frac{\cancel{\Gamma(K\alpha_0)}}{\Gamma(K\alpha_0 + N)} \prod_{k=1}^{K} \frac{\Gamma(\widetilde{N}_k + \alpha_0)}{\cancel{\Gamma(\alpha_0)}} \Big/ \frac{\cancel{\Gamma(K\alpha_0)}}{\Gamma(K\alpha_0 + N - 1)} \prod_{k=1}^{K} \frac{\Gamma(\widetilde{N}_k + \alpha_0)}{\cancel{\Gamma(\alpha_0)}} \\ &= \frac{\Gamma(K\alpha_0 + N - 1)}{\Gamma(K\alpha_0 + N)} \frac{\Gamma(\widetilde{N}_k + \alpha_0)}{\Gamma(\widetilde{N}_k + \alpha_0)} = \frac{\widetilde{N}_k + \alpha_0}{N + K\alpha_0 - 1}, \end{aligned} \tag{2.37}$$

where we have defined $\widetilde{N}_k = \sum_{j \neq n} z_{jk}$, which is equal to $\widetilde{N}_k - 1$ when $z_{nk} = 1$, and we have used the property $\Gamma(x+1) = x\Gamma(x)$.

For the second term of (2.33) we have that

$$p(\mathbf{x}_n \,|\, z_{nk} = 1, \mathbf{x}_{-n}, \mathbf{z}_{-n}, \boldsymbol{\gamma}_0) = p(\mathbf{x}_n \,|\, \mathcal{D}_{k,-n}, \boldsymbol{\gamma}_0), \tag{2.38}$$

where $\mathcal{D}_{k,-n} = \{\mathbf{x}_j : z_{jk} = 1,\, j \neq n\}$. Consequently (2.38) is the posterior predictive density for $\mathbf{x}_n$ given all the assignments and all the data except $\mathbf{x}_n$. It can be obtained by marginalizing out $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$:

$$p(\mathbf{x}_n \,|\, \mathcal{D}_{k,-n}, \boldsymbol{\gamma}_0) = \frac{p(\mathcal{D}_k|\boldsymbol{\gamma}_0)}{p(\mathcal{D}_{k,-n}|\boldsymbol{\gamma}_0)}, \tag{2.39}$$

$$p(\mathcal{D}_k|\boldsymbol{\gamma}_0) = \int_{\boldsymbol{\mu}_k} \int_{\boldsymbol{\Lambda}_k} p(\mathcal{D}_k|\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)\, p(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k|\boldsymbol{\gamma}_0)\, d\boldsymbol{\mu}_k\, d\boldsymbol{\Lambda}_k. \tag{2.40}$$

We note that, as a consequence of result (A.26), this posterior predictive distribution is a Student's t with mean $\mathbf{m}_k$, covariance $\frac{(1+\beta_k)}{(\nu_k+1-D)\beta_k}\mathbf{W}_k^{-1}$ and $\nu_k + 1 - D$ degrees of freedom.[4] These quantities are obtained from statistics of the data and the cluster assignments, exactly as illustrated for standard Gibbs sampling. We just have to consider the fact that here we want to calculate the predictive density for observation $\mathbf{x}_n$ while leaving $\mathbf{x}_n$ out.

Algorithm 4 illustrates the steps of a collapsed Gibbs sampler for the Gaussian mixture model. Each iteration resamples the indices of the observations without replacement: the random order helps to improve the mixing time of the Markov chain. The predictive densities (2.38) are efficiently updated by caching $t(\mathbf{x}_n, \mathbf{z}_n)$: the sufficient statistics (2.14) associated with each cluster. Specifically, when we are at observation $\mathbf{x}_n$, we remove $t(\mathbf{x}_n, \mathbf{z}_n)$ statistics from its current cluster $k_{\text{old}}$ and then compute the posterior predictive of $\mathbf{x}_n$ for each cluster. Once we have sampled a new cluster $k_{\text{new}}$, we add $t(\mathbf{x}_n, \mathbf{z}_n)$ to this new cluster.

We can now apply this collapsed Gibbs sampler to the simulated data in Figure 1.5 ($N = 1500$) and compare it to CAVI. We generate an additional 1500 observations from the original distribution to be used as held-out data to test the predictive performance of the two methods. Results of the comparison are shown in Figure 2.2. The two algorithms reach the same average log predictive density (1.73) on the test set, with CAVI having a shorter

[4] This mirrors the variational case (1.73)

**Algorithm 4:** Collapsed Gibbs sampling for the Bayesian GMM

**for** *R iterations* **do**
  **for** *each* $n = 1 : N$ *in random order* **do**
    Remove sufficient statistics $t(\mathbf{x}_n, \mathbf{z}_n)$ from old cluster $k_{\text{old}}$
    **for** *each* $k = 1 : K$ **do**
      Compute $p(\mathbf{x}_n \,|\, \mathcal{D}_{k,-n}, \boldsymbol{\gamma}_0)$
      Compute $p(z_{nk} = 1 \,|\, \cdot\,) \propto (\widetilde{N}_k + \alpha_0)\, p(\mathbf{x}_n \,|\, \mathcal{D}_{k,-n}, \boldsymbol{\gamma}_0)$
    **end**
    Sample $k_{\text{new}}$ with probabilities $p(z_{nk} = 1 \,|\, \cdot\,)_{\,k=1,\dots,K}$
    Add sufficient statistics $t(\mathbf{x}_n, \mathbf{z}_n)$ to new cluster $k_{\text{new}}$
  **end**
**end**

execution time.[5] For CAVI algorithm, we declare convergence once the change in the average log predictive density falls below some small threshold (here $1e^{-5}$), whereas for collapsed Gibbs sampling assessing convergence is more questionable. A good impression of convergence is given by the trace plots of the parameters or of the log-likelihood (we are using fairly diffuse priors); while a formal approach may for example be the diagnostic proposed by (Raftery & Lewis, 1992).

Table 2.1 also shows the means and the covariances of the posterior predictive distribution for the two methods: both algorithms return similar values. Note that for collapsed Gibbs sampling these quantities depend only on the data $X$ and the latent variables $Z$, therefore it is sufficient to sample $Z$ without the need to simulate the parameters $\boldsymbol{\pi}, \boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$.

Lastly, we also want to conduct the comparison with a larger dataset. Specifically, we consider again the STL-10 data histograms, but this time sampling $N = 30\,000$ images to serve as training set and another 30 000 as test set. We fit a BGMM with $K = 30$ clusters to training data using both

[5] The experiments were run on a computer with *macOS 12.1 (21C52)* operating system, *8-Core Apple M1 CPU @ 3.2GHz* and 8 GB of RAM. Python implementations of the algorithms were made as similar as possible.

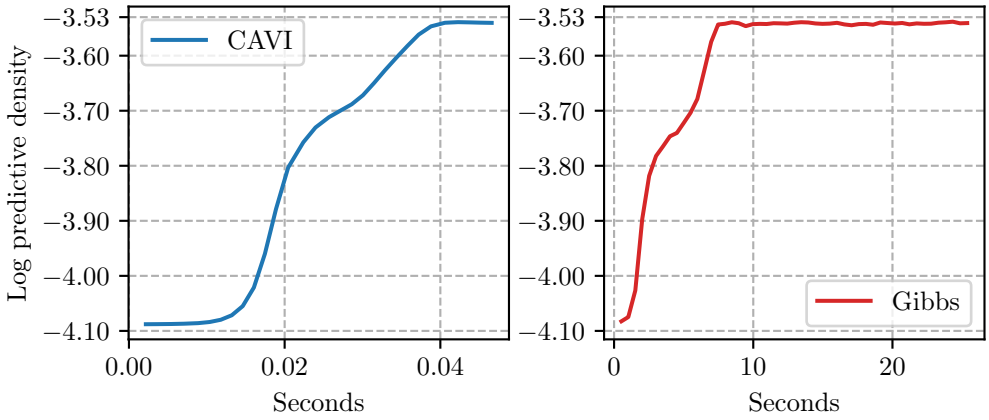



**Figure 2.2:** Evolution of the average log predictive density on the held-out simulated data, as a function of running time. CAVI (left) and collapsed Gibbs sampling (right).

**Table 2.1:** Comparison of CAVI and collapsed Gibbs sampling with regard to the means and the covariances of the posterior predictive distribution. For Gibbs sampling these values are the posterior means over 200 iterations, 50 of which used as burn-in; the output of the chain was first processed manually to deal with label switching.

| Quantity | CAVI | Gibbs |
|---|---|---|
| $\mathbf{m}_1^T$ | 1.96773 −3.03318 | 1.94143 −2.99780 |
| $\mathbf{m}_2^T$ | −1.97734 2.04785 | −1.96910 −2.03304 |
| $\mathbf{m}_3^T$ | −0.01767 1.09557 | −0.02267 1.09766 |
| $\mathbf{L}_1^{-1}$ | 1.07630 −0.09140<br>−0.09140 1.09184 | 1.08883 −0.10636<br>−0.10636 1.11176 |
| $\mathbf{L}_2^{-1}$ | 0.82661 0.03071<br>0.03071 0.45330 | 0.81461 0.03124<br>0.03124 0.45069 |
| $\mathbf{L}_3^{-1}$ | 0.53616 −0.22738<br>−0.22738 0.67211 | 0.52680 −0.21960<br>−0.21960 0.65287 |

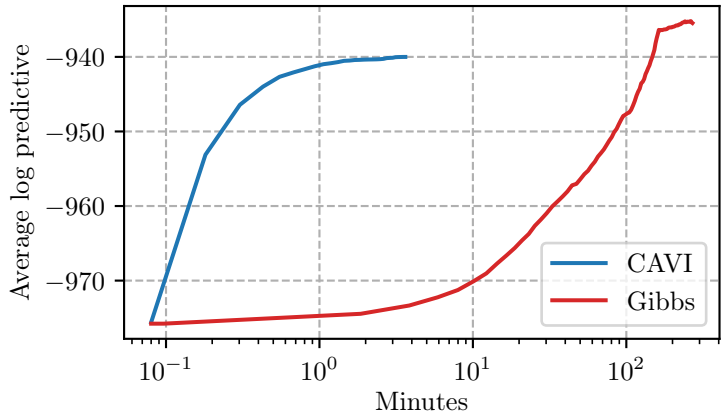



**Figure 2.3:** Predictive performance comparison between CAVI and collapsed Gibbs on STL-10 data histograms; running time is on the log scale. Both algorithms are initialized through k-means; Gibbs sampling is so costly that it only does two iterations over the entire dataset.

CAVI and a collapsed Gibbs sampler; as in the previous chapter variational hyperparameters used are $\alpha_0 = 1/K$, $\beta_0 = 1$, $\mathbf{m}_0 = \underline{\mathbf{0}}_{192}, \mathbf{W}_0 = \mathbf{I}_{192}$ and $\nu_0 = D = 192$, reflecting the absence of specific prior information.

As Figure 2.3 explains, Gibbs sampling reaches a slightly better predictive density at the cost of an exaggeratedly higher execution time. In fact, Gibbs sampling takes about 4.5 hours against the 4 minutes of CAVI. This happens because the sampler requires looping through the dataset by carrying out expensive operations for each individual data point. We have thus shown how MCMC methods can start to become untenable as the size of the data becomes larger. One can also try to compare variational inference with a Hamiltonian Monte Carlo-based sampler (Hoffman & Gelman, 2014). Nevertheless, we now see how variational inference for large data can be further speeded up and improved thanks to stochastic optimization.

## 2.2 Stochastic variational inference

Modern applications often require analyzing complicated data in terms of many data points, many dimensions and structure (e.g. images, videos, text, links). In this scenario CAVI is inefficient because, at each iteration, it needs

to perform some local computations for each data point and then aggregate them in order to re-estimate the global structure. As the dataset size grows, this process becomes computationally more expensive per iteration and wasteful, if we expect that we can learn something about the global hidden structure from only a subset of the data. This can apply to image histograms color profiles.

The key idea behind scalable VI approaches is stochastic gradient-based optimization, which climbs the ELBO objective function by following noisy estimates of its gradient. This is the case with *stochastic variational inference* (SVI) (Hoffman et al., 2013), which repeatedly (a) subsamples one or more data points from the dataset; (b) optimizes the local variational parameters of the subsample; (c) updates the current global variational parameters. These steps are straightforward and cheap to achieve for conditionally conjugate exponential family models, since it is possible to express the ELBO (and thus its gradient) as a sum of terms that can be computed independently.

### 2.2.1 SVI for conditionally conjugate models

Within this class of models, we can apply SVI and scale up inference to massive data by taking advantage of the idea of *natural gradient*, without the need for additional mathematical derivations other than those of CAVI. The natural gradient (Amari, 1998) incorporates the information geometry of the parameter space, adjusting the direction of the standard gradient by scaling it by a Riemannian metric.

In fact, the traditional Euclidean gradient metric does not convey a meaningful notion of distance between distributions parameters. Conversely, the natural gradient expresses the steepest ascent direction in the statistical manifold where local distance is defined by KL divergence.

More importantly, the natural gradient of the ELBO is easier to compute than the standard one for conditionally conjugate exponential models. This is due to the fact that the natural gradient is defined by pre-multiplying the standard gradient with the inverse Fisher information matrix (Amari, 1998).

Formally, focusing on the global variational parameters $\boldsymbol{\lambda}$, we can write the lower bound (2.12) in the form

$$\text{ELBO}(\boldsymbol{\lambda}) = \mathbb{E}_{\boldsymbol{\phi}}[\hat{\boldsymbol{\zeta}}]^T \, \mathbb{E}_{\boldsymbol{\lambda}}\left[t(\boldsymbol{\vartheta})\right] - \boldsymbol{\lambda}^T \, \mathbb{E}_{\boldsymbol{\lambda}}\left[t(\boldsymbol{\vartheta})\right] + a_g(\boldsymbol{\lambda}) + \text{const}, \tag{2.41}$$

where the constant has absorbed all terms that do not depend on $\boldsymbol{\lambda}$ and $t(\boldsymbol{\vartheta}) = [\,\eta(\boldsymbol{\vartheta}), -a_\ell(\boldsymbol{\vartheta})]$ are the sufficient statistics of the variational distribution on $\boldsymbol{\vartheta}$. Indeed, the latter has an exponential family form

$$q(\boldsymbol{\vartheta}|\boldsymbol{\lambda}) = h(\boldsymbol{\vartheta}) \exp\{\boldsymbol{\lambda}^T t(\boldsymbol{\vartheta}) - a_g(\boldsymbol{\lambda})\}. \tag{2.42}$$

From the properties of exponential families, it follows that the expectation of the sufficient statistics is the gradient of the log normalizer: $\mathbb{E}_{\boldsymbol{\lambda}}\left[t(\boldsymbol{\vartheta})\right] = \nabla_{\boldsymbol{\lambda}} \, a_g(\boldsymbol{\lambda})$. We can use this identity to derive the traditional Euclidean gradient of the ELBO as a function of $\boldsymbol{\lambda}$,

$$\nabla_{\boldsymbol{\lambda}} \, \text{ELBO} = \nabla^2_{\boldsymbol{\lambda}} a_g(\boldsymbol{\lambda}) \, (\mathbb{E}_{\boldsymbol{\phi}}[\hat{\boldsymbol{\zeta}}] - \boldsymbol{\lambda}). \tag{2.43}$$

Since in exponential families it also holds that the Fisher information is equal to $\nabla^2_{\boldsymbol{\lambda}} \, a_g(\boldsymbol{\lambda})$, we can easily obtain the natural gradient $g(\boldsymbol{\lambda})$. Pre-multiplying (2.43) by the inverse Fisher information, we have

$$g(\boldsymbol{\lambda}) = \mathbb{E}_{\boldsymbol{\phi}}[\hat{\boldsymbol{\zeta}}] - \boldsymbol{\lambda}. \tag{2.44}$$

Note that setting this gradient equal to zero, leads to the global variational updates (2.11).

At this point, we can use the natural gradient of the ELBO inside a gradient ascent algorithm for the global variational parameters (Blei et al., 2017). Namely, at iteration $t$ we have

$$\boldsymbol{\lambda}_t = \boldsymbol{\lambda}_{t-1} + \rho_t \, g(\boldsymbol{\lambda}_{t-1}), \tag{2.45}$$

where $\rho_t$ is the *step size* or *learning rate*. Substituting for (2.44) reveals a particular structure for the updates,

$$\boldsymbol{\lambda}_t = (1 - \rho_t)\boldsymbol{\lambda}_{t-1} + \rho_t \, \mathbb{E}_{\boldsymbol{\phi}}[\hat{\boldsymbol{\zeta}}]. \tag{2.46}$$

However this gradient-based optimization of the global variational parameters has the same cost of corrispondent CAVI updates, since it requires iterating through all data points.

The key idea behind SVI is stochastic optimization (Robbins & Monro, 1951), which has enabled modern machine learning. Stochastic optimization algorithms replace the gradient of the objective function with cheaper noisy estimates to reach the optimum. Convergence to at least local optima is guaranteed, as long as the gradient estimates are unbiased and the step size sequence satisfies the Robbins-Monro conditions:

$$\sum_{t=1}^{\infty} \rho_t = \infty, \quad \sum_{t=1}^{\infty} \rho_t^2 < \infty. \tag{2.47}$$

These ensure that every point in the parameter space can be reached, while the learning rate decreases quickly enough to guarantee convergence.

Returning to variational inference, the natural gradient of the ELBO is

$$g(\boldsymbol{\lambda}) = \boldsymbol{\zeta} + \left[ \sum_{n=1}^{N} \mathbb{E}_{\phi_n^*}[t(x_n, z_n)],\ N \right]^T - \boldsymbol{\lambda}, \tag{2.48}$$

where $\phi_n^*$ highlights that we are considering the optimized local variational parameters (2.10). We can construct a noisy natural gradient by sampling an index $j$ from the data and pretending to replicate the corresponding data point $N$ times, which means

$$\widehat{g}(\boldsymbol{\lambda}) = \boldsymbol{\zeta} + N \left[ \mathbb{E}_{\phi_j^*}[t(x_j, z_j)],\ 1 \right]^T - \boldsymbol{\lambda}. \tag{2.49}$$

The noisy natural gradient $\widehat{g}(\boldsymbol{\lambda})$ is unbiased, $\mathbb{E}[\widehat{g}(\boldsymbol{\lambda})] = g(\boldsymbol{\lambda})$, and cheap to compute, as it only depends on optimized local parameters of one sampled data point. Therefore, this is a valid estimate to plug into equation (2.45) to give

$$\begin{aligned} \boldsymbol{\lambda}_t &= (1 - \rho_t)\boldsymbol{\lambda}_{t-1} + \rho_t \widehat{\boldsymbol{\lambda}}, \quad \text{where} \\ \widehat{\boldsymbol{\lambda}} &= \boldsymbol{\zeta} + N \left[ \mathbb{E}_{\phi_j^*}[t(x_j, z_j)],\ 1 \right]^T. \end{aligned} \tag{2.50}$$

The full SVI algorithm is presented in Algorithm 5. At each iteration we update the global variational parameters as a weighted average of the previous setting and the estimate we would obtain replicating the sampled data point $N$ times. Algorithm 6 describes the BGMM example. It should be emphasized that implementing SVI does not require new mathematical derivations beyond those needed for CAVI algorithm.

**Algorithm 5:** SVI for conditionally conjugate models

**Initialize:** Variational global parameters $\boldsymbol{\lambda}$

Schedule the step size sequence $\rho_t$ appropriately

**while** *not converged* **do**

Sample a data point $x_j$ uniformly from the dataset.

Optimize its local variational parameters:

$$\phi_j^* = \mathbb{E}_{\boldsymbol{\lambda}}\left[\eta(\boldsymbol{\vartheta}, x_j)\right]$$

Compute the intermediate global parameters as though $x_j$ were replicated $N$ times:

$$\widehat{\boldsymbol{\lambda}} = \boldsymbol{\zeta} + N\left[\mathbb{E}_{\phi_j^*}[t(x_j, z_j)],\ 1\right]^T$$

Update the current global variational parameters:

$$\boldsymbol{\lambda}_t = (1-\rho_t)\boldsymbol{\lambda}_{t-1} + \rho_t\,\widehat{\boldsymbol{\lambda}}$$

**end**

**Algorithm 6:** SVI for the Bayesian Gaussian mixture model

**Initialize:** $\boldsymbol{\lambda} = [\,\alpha_k, \beta_k, \mathrm{vec}(\mathbf{m}_k), \mathrm{vec}(\mathbf{W}_k^{-1}), \nu_k\,]_{k=1,\ldots,K}$. Schedule $\rho_t$

**while** *not converged* **do**

Sample a data point $\mathbf{x}_j$ uniformly from the dataset.

Optimize its local variational parameters:

$$r_{jk} \propto \widetilde{\pi}_k\,\widetilde{\boldsymbol{\Lambda}}_k\,\exp\left\{-\tfrac{D}{2\beta_k} - \tfrac{\nu_k}{2}(\mathbf{x}_j - \mathbf{m}_k)^T\,\mathbf{W}_k\,(\mathbf{x}_j - \mathbf{m}_k)\right\}$$

Compute the intermediate $\widehat{\boldsymbol{\lambda}}$ defined by

$$\alpha_k \leftarrow \alpha_0 + N \cdot r_{jk}$$
$$\beta_k \leftarrow \beta_0 + N \cdot r_{jk}$$
$$\mathbf{m}_k \leftarrow \tfrac{1}{\beta_k}\left(\beta_0\mathbf{m}_0 + N \cdot r_{jk}\,\mathbf{x}_j\right)$$
$$\mathbf{W}_k^{-1} \leftarrow \mathbf{W}_0^{-1} + \beta_0\mathbf{m}_0\mathbf{m}_0^T - \beta_k\mathbf{m}_k\mathbf{m}_k^T + N \cdot r_{jk}\,\mathbf{x}_j\,\mathbf{x}_j^T$$
$$\nu_k \leftarrow \nu_0 + N \cdot r_{jk}$$

Update the current global variational parameters:

$$\boldsymbol{\lambda}_t = (1-\rho_t)\boldsymbol{\lambda}_{t-1} + \rho_t\,\widehat{\boldsymbol{\lambda}}$$

**end**

A common choice for the step size sequence that satisfies conditions (2.47) is, at iteration $t$,

$$\rho_t = (t + \omega)^{-\delta}, \tag{2.51}$$

where the *forgetting rate* $\delta \in (0.5, 1]$ controls how quickly old information decays and $\omega \geq 0$ down-weights early iterations (Hoffman et al., 2013). Following noisy but fast-to-compute natural gradients with an appropriate step size, SVI algorithm converges to a local optimum of the ELBO.

We have therefore seen how, within conditionally conjugate exponential models, SVI is appropriate for scaling up to massiva data, especially when these do not fit in memory. The method can be further made more efficient with the help of distributed optimization and cluster computing, though in principle it is designed to run on a single machine. Moreover, the noise introduced by the gradient estimates makes the stochastic variational algorithm more likely to circumvent shallow local optima of the objective in which the deterministic version gets stuck (see Bottou, 1991 and the link with *simulated annealing*).

A remarkable demonstration of the usefulness of SVI lies in *topic models*, probabilistic models of text used to uncover the hidden thematic structure in a collection of documents. Topic models assume that the words of each document arise from a mixture of the latent topics; the topics are shared across a collection, but each document mixes them with its own proportions. (Hoffman et al., 2013) applied posterior inference by using a conditionally conjugate topic model, *Latent Dirichlet Allocation* (Blei et al., 2003), with 3.8 million *Wikipedia* documents, an analysis that would not have been possible without SVI.

### 2.2.2 Mini-batching and adaptive learning rate

From a practical point of view, estimating the gradients with only one random data point, may often lead to a very noisy optimization process that oscillate far from a equilibrium. A solution to address that is sampling a *minibatch* of $B$ data points at each iteration (usually without replacement).

The *batch size* $B$ trades off between achieving a relative speedy convergence and injecting enough randomness to each gradient estimate. Generally, a well-tuned batch size helps the stochastic algorithm to escape from poor local optima/saddle points and to amortize the computational expenses associated to global variational updates.

The SVI algorithm is easily extended to minibatches by considering that a stochastic estimate of the ELBO can be obtained as a sum of terms appropriately rescaled:

$$\begin{aligned}\widetilde{\mathrm{ELBO}}(\boldsymbol{\lambda}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{\lambda},\boldsymbol{\phi}}\left[\log p(\boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\boldsymbol{\lambda})\right] + \\ \frac{N}{B}\sum_{b=1}^{B}\mathbb{E}_{\boldsymbol{\lambda},\boldsymbol{\phi}}\left[\log p(x_{j_b}|z_{j_b}, \boldsymbol{\vartheta}) + \log p(z_{j_b}|\boldsymbol{\vartheta}) - \log q(z_{j_b}|\phi_{j_b})\right],\end{aligned} \tag{2.52}$$

where $j_b$ are the indices of the data points sampled in the minibatch considered. Similarly, the global variational parameters updates are obtained by first optimizing the local parameters of the minibatch $\phi^*_{j_b},\ b = 1, \ldots, B$, and then computing the intermediate values

$$\widehat{\boldsymbol{\lambda}} = \boldsymbol{\zeta} + \frac{N}{B}\left[\sum_{b=1}^{B}\mathbb{E}_{\phi^*_{j_b}}[t(x_{j_b}, z_{j_b})],\ B\right]^T, \tag{2.53}$$

that have been rescaled appropriately.

Another trick that helps the algorithm converge faster and to better approximations is *learning rate adaptation*. In fact, SVI is sensitive to learning rate scheduling: with a sequence that decays too quickly the algorithm advances languidly; with a sequence that decreases too slowly it makes erratic and unreliable progress. To address this, we rely on a method that automatically adapts the step size to the sampled data. We follow the approach of (Ranganath et al., 2013) who proposed a tailor-made adaptive learning rate for SVI, which requires no hand-tuning and uses computations already made within the SVI algorithm.

The method aims to estimate the learning rate $\rho_{t+1}$ that minimizes the expected error between the stochastic update $\boldsymbol{\lambda}_{t+1}$ (2.50) and the coordinate update $\boldsymbol{\lambda}^*_{t+1}$ (2.11). The squared norm of the error is

$$J(\rho_{t+1}) = (\boldsymbol{\lambda}_{t+1} - \boldsymbol{\lambda}^*_{t+1})^T(\boldsymbol{\lambda}_{t+1} - \boldsymbol{\lambda}^*_{t+1}). \tag{2.54}$$

The latter is a random variable because $\boldsymbol{\lambda}_{t+1}$ depends on the intermediate parameters $\widehat{\boldsymbol{\lambda}}$, calculated with a random minibatch.

Let $g_t$ be the stochastic natural gradient (2.49) as a function of $\boldsymbol{\lambda}_t$, then by minimizing $\mathbb{E}_B[J(\rho_{t+1})|\boldsymbol{\lambda}_t]$ it can be proved (Ranganath et al., 2013) that the optimal adaptive learning rate is

$$\rho^*_{t+1} = \frac{\mathbb{E}_B[g_t]^T \mathbb{E}_B[g_t]}{\mathbb{E}_B[g_t^T g_t]}, \tag{2.55}$$

where the subscript $B$ indicates that the expectation is taken with respect to a specific minibatch. The adaptive learning rate shrinks when the variance of the noisy gradient is large and grows when the square of the expected noisy gradient is large, indicating that the algorithm is far from a optima.

These expectations are approximated with exponential moving averages across iterations. Let $\overline{g}_t$ and $\overline{h}_t$ be respectively the moving averages for $\mathbb{E}[g_t]$ and $\mathbb{E}[g_t^T g_t]$. These values are updated as

$$\begin{aligned} \overline{g}_t &= (1 - \tau_t^{-1})\overline{g}_{t-1} + \tau_t^{-1} g_t \\ \overline{h}_t &= (1 - \tau_t^{-1})\overline{h}_{t-1} + \tau_t^{-1} g_t^T g_t, \end{aligned} \tag{2.56}$$

where $\tau_t$ is the window size of the exponential moving averages. Substituting these estimates into equation (2.55), the adaptive learning rate can be approximated with

$$\rho^*_{t+1} \approx \frac{\overline{g}_t^T \overline{g}_t}{\overline{h}_t}. \tag{2.57}$$

Since the moving averages are less reliable after large steps, the window size can be updated as

$$\tau_{t+1} = \tau_t \left(1 - \rho^*_{t+1}\right) + 1. \tag{2.58}$$

By keeping the necessary quantities in memory and carrying out the updates described above, we can thus adapt the learning rate at each iteration, in which a new minibatch is sampled. The authors suggest to initialize the moving averages through Monte Carlo estimates obtained by forming noisy gradients on several samples and to inizialize the memory size as the Monte Carlo sample size used. In this way we only need to specify an initial memory size $\tau_0$ and we can make use of a learning-rate-free SVI algorithm.

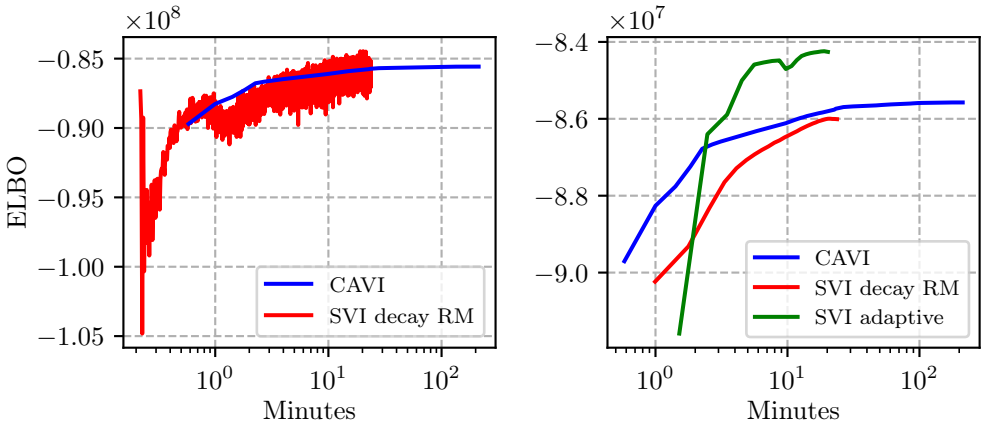
We implemented the methods presented and performed some experiments. We considered again the STL-10 data histograms but this time using all the unlabeled images, of which $N = 90\,000$ were sampled to serve as training set and the remaining $10\,000$ as held-out set. The rest of the setting was the same as the previous sections.

Figure 2.4 compares SVI and CAVI for these data: the plot on the left-hand side shows the evolution of the stochastic ELBO (2.52); the plot on the right-hand side illustrates the average of the stochastic ELBO in each *epoch*, which is one complete pass through all the training data. For CAVI the two plots represent the same thing and the ELBO is guaranteed to improve with each step (the proof is similar to EM algorithm), for SVI this is not the case.

From Figure 2.4(a) we see that SVI with the best Robbins-Monro learning rate (2.51) ($\delta = 0.9$, $\omega = 1$) climbs the ELBO with noise and converges faster than CAVI, but achieving a slightly worse optimum, as shown in Figure 2.4(b). Whereas, using the adaptive learning rate, the stochastic variational algorithm reaches a higher lower bound than the *batch* algorithm (that is CAVI). This is due to the fact that, by subsampling a minibatch at each iteration and using a suitable step size, the stochastic algorithm explores the parametric space more widely and it is more likely to find new and better maxima in the non-convex optimization problem.

Figure 2.5 shows the behaviour of the two learning rate types: the adaptive learning rate roughly follows the shape of the best Robbins-Monro decay rate, but automatically adapts to the sampled data. Maintaining the adaptive step size, we also tested the sensitivity of the stochastic variational algorithm to the batch sizes: results are illustrated in Table 2.2. Larger batch sizes are preferred in terms of predictive performance, however there is not a big difference between $B = 512$ and $B = 1024$, therefore 512 is a suitable value for batch size.

Finally, we also tried varying the number of clusters: Table 2.3 shows the average log predictive densities obtained for different values of $K$. We found that $K = 30$ leads to the best performance, however for this purpose *hierarchical Dirichlet process* (HDP) mixture models (Teh et al., 2006) often




**(a)** Stochastic ELBO **(b)** Average ELBO

**Figure 2.4:** Lower bound trajectories for CAVI and SVI as a function of running time (log scale). SVI uses a batch size $B = 512$: for the RM step size we use $\delta = 0.9$, $\omega = 1$; for the adaptive learning rate we set $\tau_0 = 500$. All algorithms start from the same k-means initialization (not plotted); convergence is declared when the (average) ELBO changes below a small threshold or decreases for three consecutive epochs. Different initializations yield similar results.

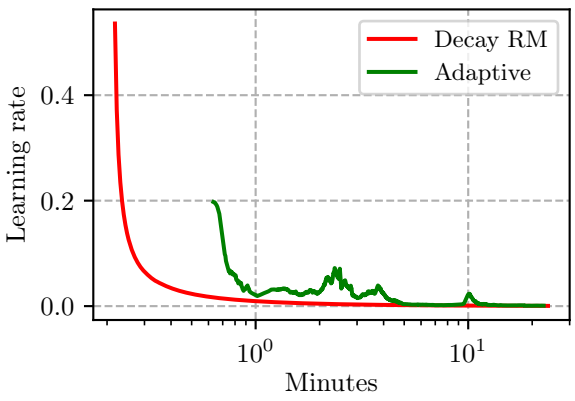



**Figure 2.5:** Path of the adaptive learning rate and of the best Robbins-Monro decay rate, as a function of execution time (on the log scale). The same training procedures as above are used. The adaptive learning rate seems too low at first, but then it adapts appropriately to the data.

achieve the best results. These are Bayesian nonparametric models that allow to extend mixtures to infinitely countable numbers of components. We did not explore this class of models in the thesis.

**Table 2.2:** Log predictive densities on held-out data for different batch sizes $B$. We show means and standard errors over 5 different initializations. The initial $\tau_0$ are set according to the batch size.

| **Log predictive** | **Batch size** | | | | |
|---|---|---|---|---|---|
| **density** | 64 | 128 | 256 | 512 | 1024 |
| Mean | −984.70 | −962.26 | −945.00 | −921.10 | −919.03 |
| Std.error | (6.335) | (6.310) | (0.884) | (0.969) | (0.962) |

**Table 2.3:** Log predictive densities on held-out data for different numbers of mixture components $K$. We show means and standard errors over 5 different initializations. The adaptive learning rate and a batch size $B = 512$ are used.

| **Log predictive** | **Number of clusters** | | | | |
|---|---|---|---|---|---|
| **density** | 10 | 20 | 30 | 40 | 50 |
| Mean | −962.12 | −925.44 | −920.87 | −927.93 | −925.48 |
| Std.error | (2.739) | (1.512) | (1.053) | (1.611) | (2.099) |

## 2.3 Black box variational inference

We have shown how to develop scalable variational inference for conditionally conjugate models, where each full conditional is in the exponential family and we can obtain closed-form variational updates. Some models fall into this class, including Bayesian mixtures of exponential families, hierarchical linear and probit regression models, factorial models, hidden Markov models and mixed-membership models (e.g. Latent Dirichlet Allocation).

For generic models outside this class, however, it is not possible to directly attain an analytic solution: the required expectations for mean field VB become intractable. A simple example is Bayesian logistic regression:

$$y_n \,|\, \mathbf{x}_n, \boldsymbol{\vartheta} \sim \text{Bern}(\sigma(\boldsymbol{\vartheta}^T \mathbf{x}_n)), \quad \boldsymbol{\vartheta} \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0), \tag{2.59}$$

where $y_n$ is the binary response, $\mathbf{x}_n$ the covariates, $\boldsymbol{\vartheta}$ the regression coefficients and $\sigma(\cdot)$ the logistic function. The problem is that we cannot analytically compute the expectation in the lower bound and in the variational updates. This characterizes the so-called *nonconjugate* models, for example Bayesian generalized regression models, deep exponential families (Ranganath et al., 2015), deep latent Gaussian models (Rezende et al., 2014), Bayesian neural networks (Neal, 1996), and many others.

To cope with nonconjugate models, researchers have developed model-specific variational methods. In particular, (Jaakkola & Jordan, 2000) proposed a tangent quadratic bound tailored to logistic regression; while (Wang & Blei, 2013) developed generic variational algorithms that use Laplace approximations and the delta method, but require model-specific calculations. Deriving these algorithms when developing and exploring new models, however, is challenging work. Tedious model-based derivations and implementation hinder us from quickly investigating and refining a range of models.

To address that, (Ranganath et al., 2014) came up with an approach called *black box variational inference* (BBVI), which can be applied with little effort to a broad class of generic models and arbitrary variational families. The goal is to make VI algorithms more automatic and easy-to-use. Operationally, the key idea behind the method is to express the gradient of the ELBO as an expectation and obtain Monte Carlo estimates of it for use in stochastic optimization.

In effect, the generic variational objective has the form

$$\text{ELBO}(\boldsymbol{\psi}) = \mathbb{E}_q[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})], \tag{2.60}$$

where the expectation is taken with respect to the variational distribution $q(\mathbf{z}|\boldsymbol{\psi})$ with free variational parameters $\boldsymbol{\psi}$. Here we have to pick a convenient

*fixed-form* for $q$ (whether mean field factorized or not), in contrast to what we saw in previous conditionally conjugate examples, where the functional form of the variational factors was specified directly by the optimization.

Depending on how the gradient of (2.60) is rewritten as an expectation w.r.t. the variational distribution, there are two main BBVI strategies: the *score gradient* (Ranganath et al., 2014) and the *reparameterization gradient* (Kingma & Welling, 2014).

### 2.3.1 Score gradient

To define the score estimator, we write the gradient of the ELBO as follows,

$$\begin{aligned}
\nabla_{\boldsymbol{\psi}} \text{ELBO} &= \nabla_{\boldsymbol{\psi}} \int \big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big)\, q(\mathbf{z}|\boldsymbol{\psi})\, d\mathbf{z} \\
&= \int \nabla_{\boldsymbol{\psi}} \big[\big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big)\, q(\mathbf{z}|\boldsymbol{\psi})\big]\, d\mathbf{z} \\
&= \int \nabla_{\boldsymbol{\psi}} \big[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big]\, q(\mathbf{z}|\boldsymbol{\psi})\, d\mathbf{z} \\
&\quad + \int \nabla_{\boldsymbol{\psi}}\, q(\mathbf{z}|\boldsymbol{\psi}) \big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big)\, d\mathbf{z} \\
&= -\mathbb{E}_q[\nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi})] + \int \nabla_{\boldsymbol{\psi}}\, q(\mathbf{z}|\boldsymbol{\psi}) \big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big)\, d\mathbf{z}.
\end{aligned} \tag{2.61}$$

We have exchanged derivatives with integrals via the dominated convergence theorem and used the fact that $\nabla_{\boldsymbol{\psi}} \log p(\mathbf{x}, \mathbf{z}) = 0$.

The quantity $\nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi})$ is known as score function and has expectation equal to zero:

$$\begin{aligned}
\mathbb{E}_q[\nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi})] = \mathbb{E}_q\left[\frac{\nabla_{\boldsymbol{\psi}}\, q(\mathbf{z}|\boldsymbol{\psi})}{q(\mathbf{z}|\boldsymbol{\psi})}\right] &= \int \nabla_{\boldsymbol{\psi}}\, q(\mathbf{z}|\boldsymbol{\psi})\, d\mathbf{z} \\
&= \nabla_{\boldsymbol{\psi}} \int q(\mathbf{z}|\boldsymbol{\psi})\, d\mathbf{z} = \nabla_{\boldsymbol{\psi}}\, 1 = 0,
\end{aligned} \tag{2.62}$$

which cancels the first term in equation (2.61). The second term is simplified by using again the *log derivative trick* $\nabla_{\boldsymbol{\psi}}\, q(\mathbf{z}|\boldsymbol{\psi}) = q(\mathbf{z}|\boldsymbol{\psi})\, \nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi})$, to give the score (or REINFORCE) gradient:

$$\nabla_{\boldsymbol{\psi}} \text{ELBO} = \mathbb{E}_q\big[\nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi}) \big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big)\big]. \tag{2.63}$$

This allows to compute noisy but unbiased gradients of the ELBO using

Monte Carlo samples from the variational distribution, as

$$\widehat{\nabla_{\boldsymbol{\psi}} \text{ELBO}} = \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}_s|\boldsymbol{\psi}) \left(\log p(\mathbf{x}, \mathbf{z}_s) - \log q(\mathbf{z}_s|\boldsymbol{\psi})\right), \quad \text{where } \mathbf{z}_s \sim q(\mathbf{z}|\boldsymbol{\psi}). \tag{2.64}$$

Note that sampling from $q(\mathbf{z}|\boldsymbol{\psi})$ and evaluating the variational distribution together with its score function do not involve the model $p(\cdot)$: these functions can be build up in a library for common variational distributions and reused for many models. In this way the gradients satisfy a black box criteria, since the only model-specific operation required of the practitioner is computing the log of the joint distribution $p(\mathbf{x}, \mathbf{z}_s)$. For this reason, the BBVI score method can be quickly applied to a vast range of nonconjugate models for which the joint distribution is evaluable, and plays in VI the role that Metropolis-Hastings algorithm plays in MCMC.

The Monte Carlo estimates (2.64) can then be used in a stochastic optimization routine to optimize the variational parameters $\boldsymbol{\psi}$ and thus the ELBO. The method can take advantage of adaptive learning rates (such as *Adam* that will be illustrated in Subsection 3.1.5), and of data subsampling (mini-batching) seen for SVI. The latter leads to what is called a *doubly stochastic variational approximation* (Titsias & Lázaro-Gredilla, 2014), as it involves sampling from both the variational distribution and the dataset.

Nevertheless, in practice, the variance of the score gradient estimator is large, and that would require very small step sizes leading to slow convergence. Thererore, we need to rely on methods that reduce this variance, such as *control variates* (Ross, 2022). Suppose that we want to estimate $\mathbb{E}_q[f(z)]$ via Monte Carlo by computing an empirical mean of samples from $q$. Then the function $f(z)$ can be replaced with

$$\widehat{f}(z) = f(z) - a\,(h(z) - \mathbb{E}_q[h(z)]). \tag{2.65}$$

This is called a *control variate*, a family of functions indexed by the coefficient $a$ and where $h$ is a *baseline* function. It is easy to see that $\widehat{f}(z)$ is an unbiased estimator, $\mathbb{E}_q[\widehat{f}(z)] = \mathbb{E}_q[f(z)]$, with variance

$$\text{Var}[\widehat{f}(z)] = \text{Var}[f(z)] + a^2 \,\text{Var}[h(z)] - 2a \,\text{Cov}[f(z), h(z)]. \tag{2.66}$$

By setting the derivative of $\mathrm{Var}[\widehat{f}(z)]$ w.r.t. $a$ equal to 0, we obtain that the optimal value of $a$ that minimizes the variance is

$$a^* = \frac{\mathrm{Cov}[f(z), h(z)]}{\mathrm{Var}[h(z)]}. \tag{2.67}$$

Substituting this value into equation (2.66) gives

$$\mathrm{Var}[\widehat{f}(z)] = \mathrm{Var}[f(z)] - \frac{\mathrm{Cov}[f(z), h(z)]^2}{\mathrm{Var}[h(z)]} = \big(1 - \mathrm{Corr}[f(z), h(z)]^2\big)\,\mathrm{Var}[f(z)]. \tag{2.68}$$

As a consequence, the more correlated the functions $f$ and $h$, the greater the variance reduction.

Returning to BBVI, we want to choose a control variate that preserves our goal of inference without model-specific computations. A good solution is choosing $h$ to be the score function $\nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi})$, which is already needed and has expectation zero. In this way, the control variate for the $m$-th entry of the gradient is defined by

$$\begin{aligned} f_m(\mathbf{z}) &= \nabla_{\boldsymbol{\psi}_m} \log q(\mathbf{z}|\boldsymbol{\psi}) \big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big), \\ h_m(\mathbf{z}) &= \nabla_{\boldsymbol{\psi}_m} \log q(\mathbf{z}|\boldsymbol{\psi}), \\ a_m^* &= \mathrm{Cov}[f_m(\mathbf{z}), h_m(\mathbf{z})] \,/\, \mathrm{Var}[h_m(\mathbf{z})]. \end{aligned} \tag{2.69}$$

The variance and covariances necessary for $a_m^*$ are estimated on a small number of samples from $q$, to give the estimated scalings $\hat{a_m}$. Thus, it is possible to obtain the Monte Carlo estimates of the gradients as

$$\widehat{\nabla_{\boldsymbol{\psi}_m} \mathrm{ELBO}} = \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\psi}_m} \log q(\mathbf{z}_s|\boldsymbol{\psi}) \big(\log p(\mathbf{x}, \mathbf{z}_s) - \log q(\mathbf{z}_s|\boldsymbol{\psi}) - \hat{a_m}\big). \tag{2.70}$$

Finally, the whole score-gradient BBVI algorithm with control variates is presented in Algorithm 7, which can be easily adjusted to subsample minibatches. One might also use different variance reduction tecniques, like *Rao-Blackwellization* or *importance sampling* (Ruiz et al., 2016).

The BBVI approach is strongly motivated by the idea of probabilistic programming, which allows a user to write down a probability model as a computer program that generates data and then compile it into an efficient inference executable, with systems such as `Stan` (Carpenter et al., 2017), `PyMC3` (Salvatier et al., 2016), and `Edward` (Tran et al., 2016).

**Algorithm 7:** Score gradient BBVI with control variates

**Initialize:** Variational parameters $\boldsymbol{\psi}$. Schedule $\rho_t$ appropriately

**while** *not converged* **do**

Draw $S$ samples from the variational distribution:

$$\mathbf{z}[s] \sim q(\mathbf{z}|\boldsymbol{\psi}), \quad s = 1, \ldots, S$$

**for** *each* $m = 1 : M$ **do**

**for** *each* $s = 1 : S$ **do**

$$f_m[s] = \nabla_{\boldsymbol{\psi}_m} \log q(\mathbf{z}[s] \,|\, \boldsymbol{\psi}) \left(\log p(\mathbf{x}, \mathbf{z}[s]) - \log q(\mathbf{z}[s] \,|\, \boldsymbol{\psi})\right)$$

$$h_m[s] = \nabla_{\boldsymbol{\psi}_m} \log q(\mathbf{z}[s] \,|\, \boldsymbol{\psi})$$

**end**

Estimate $\hat{a_m} = \hat{\mathrm{Cov}}[f_m, h_m] \,/\, \hat{\mathrm{Var}}[h_m]$ from a few samples

Compute the noisy gradient $\hat{g}_m = \frac{1}{S} \sum_{s=1}^{S} f_m[s] - \hat{a_m}\, h_m[s]$

**end**

Update the current variational parameters:

$$\boldsymbol{\psi}^{(t)} = \boldsymbol{\psi}^{(t-1)} + \rho_t \, \hat{g}$$

**end**

### 2.3.2 Reparameterization gradient

An alternative way to obtain a low-variance gradient estimator is the *reparameterization trick* (Kingma & Welling, 2014). It assumes that:

(i) $\log p(\mathbf{x}, \mathbf{z})$ and $\log q(\mathbf{z}|\boldsymbol{\psi})$ are differentiable with respect to $\mathbf{z}$;

(ii) we can sample $\mathbf{z} \sim q(\mathbf{z}|\boldsymbol{\psi})$ by first sampling a noise term $\boldsymbol{\epsilon} \sim v(\boldsymbol{\epsilon})$ independent of $\boldsymbol{\psi}$, and then transforming it to $\mathbf{z}$ using a deterministic and differentiable function $\mathbf{z} = r(\boldsymbol{\epsilon}, \boldsymbol{\psi})$. For example, if $\mathbf{z} \sim \mathcal{N}(\mu, \sigma^2)$ then $\mathbf{z} = r(\boldsymbol{\epsilon}, \boldsymbol{\psi}) = \mu + \sigma\boldsymbol{\epsilon}$, with $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$.

This allows us to rewrite the variational objective as an expectation with respect to $v(\boldsymbol{\epsilon})$, namely

$$\begin{aligned} \mathrm{ELBO}(\boldsymbol{\psi}) &= \mathbb{E}_{q(\mathbf{z}|\boldsymbol{\psi})}[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})] \\ &= \mathbb{E}_{v(\boldsymbol{\epsilon})}[\log p(\mathbf{x}, r(\boldsymbol{\epsilon}, \boldsymbol{\psi})) - \log q(r(\boldsymbol{\epsilon}, \boldsymbol{\psi}))]. \end{aligned} \tag{2.71}$$

At this point, since $v(\boldsymbol{\epsilon})$ is independent of $\boldsymbol{\psi}$, we can push the gradient of the ELBO inside the expectation and use the chain rule to obtain

$$\nabla_{\boldsymbol{\psi}} \text{ELBO} = \mathbb{E}_{v(\boldsymbol{\epsilon})}\big[\nabla_{\mathbf{z}}\big(\log p(\mathbf{x},\mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi})\big)\,\nabla_{\boldsymbol{\psi}}\, r(\boldsymbol{\epsilon},\boldsymbol{\psi}) - \nabla_{\boldsymbol{\psi}} \log q(\mathbf{z}|\boldsymbol{\psi})\big], \quad \text{where } \mathbf{z} = r(\boldsymbol{\epsilon},\boldsymbol{\psi}). \tag{2.72}$$

The last term vanishes since the score function has zero expectation, and

$$\nabla_{\boldsymbol{\psi}} \text{ELBO} = \mathbb{E}_{v(\boldsymbol{\epsilon})}\big[\,[\nabla_{\mathbf{z}}(\log p(\mathbf{x},\mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{\psi}))\,\nabla_{\boldsymbol{\psi}}\, r(\boldsymbol{\epsilon},\boldsymbol{\psi})]_{\mathbf{z}=r(\boldsymbol{\epsilon},\boldsymbol{\psi})}\big], \tag{2.73}$$

which is known as *reparameterization* or *pathwise* gradient. The corrispondent Monte Carlo estimates are used in a stochastic optimization as Algorithm 8 illustrates.

Note that unlike the previous BBVI method, the reparameterization approach works only for differentiable models with continuous latent variables; however, this is bypassed for models such as mixture models where discrete latent variables can be marginalized out. Since it requires calculating $\nabla_{\mathbf{z}} \log p(\mathbf{x},\mathbf{z})$, the reparameterization estimator is therefore more computationally expensive than the score estimator, but often has lower variance, intuitively because it uses more information from the log posterior. This is theoretically proven by (Xu et al., 2019) in certain cases, while (Kucukelbir et al., 2017) show empirically that in their settings the reparametrization gradient estimator exhibits much lower variance than the score estimator with control variates.

Reparametrization BBVI is made attractive by *automatic differentiation* techniques, which allow to compute derivatives in an algorithmic manner using frameworks such as `autograd`, `TensorFlow` and `Theano`. In our case this is especially useful for evaluating the gradient of the log joint model w.r.t. the latent variables and fits with the ideas of probabilistic programming and automated VI.

In this context, the most commonly used approximations for $q(\mathbf{z}|\boldsymbol{\psi})$ are Gaussian distributions. The standard option is to posit a parametric

**Algorithm 8:** Reparameterization gradient BBVI

**Initialize:** Variational parameters $\boldsymbol{\psi}$. Schedule $\rho_t$ appropriately

**while** *not converged* **do**

Draw $S$ samples from the auxiliary distribution:

$$\boldsymbol{\epsilon}_s \sim v(\boldsymbol{\epsilon}), \quad s = 1, \ldots, S$$

Compute the noisy gradient:

$$\hat{g} = \tfrac{1}{S} \textstyle\sum_{s=1}^{S} \nabla_{\mathbf{z}} \big[\log p(\mathbf{x}, r(\boldsymbol{\epsilon}_s, \boldsymbol{\psi})) - \log q(r(\boldsymbol{\epsilon}_s, \boldsymbol{\psi}))\big] \nabla_{\boldsymbol{\psi}} r(\boldsymbol{\epsilon}_s, \boldsymbol{\psi})$$

Update the current variational parameters:

$$\boldsymbol{\psi}^{(t)} = \boldsymbol{\psi}^{(t-1)} + \rho_t \, \hat{g}$$

**end**

fully-factorized mean field Gaussian approximation

$$q(\mathbf{z}|\boldsymbol{\psi}) = \prod_{j=1}^{m} \mathcal{N}(z_j \,|\, \mu_j, \sigma_j^2). \tag{2.74}$$

This family is flexible and efficient, but its assumptions do not allow it to capture posterior correlations between latent variables (Blei et al., 2017).

An extension is *full-rank* Gaussian variational Bayes (GVB), namely

$$q(\mathbf{z}|\boldsymbol{\psi}) = \mathcal{N}(\mathbf{z} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma}), \tag{2.75}$$

with the covariance matrix represented using its Cholesky decomposition $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix. We can sample $\mathbf{z} \sim \mathcal{N}(\mathbf{z} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ using a location-scale transformation, $\mathbf{z} = r(\boldsymbol{\epsilon}, \boldsymbol{\psi}) = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_m)$, in order to define the reparameterization gradient to be calculated with the help of autodifferentiation.

The full-rank GVB comes at a computational cost, but leads to a more accurate posterior approximation, generalizing the mean field Gaussian family. In fact, the off-diagonal terms in $\boldsymbol{\Sigma}$ enable to capture posterior correlations across latent variables.

In high dimensions, an efficient alternative to the Cholesky decomposition is the factor decomposition

$$\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^T + \mathbf{C}^2, \tag{2.76}$$

where $\mathbf{B}$ is the factor loading matrix of size $m \times f$, where $f \ll m$ is the number of factors, $m$ is the dimensionality of $\mathbf{z}$, and $\mathbf{C} = \text{diag}(c_1, ..., c_m)$. This *low-rank* GVB with factor covariance structure (Ong et al., 2018) is useful when $m$ is large, as the number of variational parameters reduces from $m + m(m+1)/2$ in the full-rank case to $(f+2)m$.

An approach towards more expressive variational posteriors, which go beyond Gaussian approximations, will be mentioned in Chapter 4. As an aside, note that throughout the thesis we focus only on optimizing the $\text{KL}(q \,||\, p)$ divergence. However, optimizing alternative measures opens up other variational approximation methods, such as *expectation propagation* (Minka, 2001), which is based on the reverse Kullback-Leibler divergence $\text{KL}(p \,||\, q)$.

To wrap up the discussion on reparameterization BBVI, there is the approach proposed by (Kucukelbir et al., 2017), *automatic differentiation variational inference* (ADVI). It first automatically transforms the joint density of the model so that the latent variables $\mathbf{z}$ live in the real coordinate space, and then posits a Gaussian variational approximation without constraints to apply GVB with the reparameterization trick.

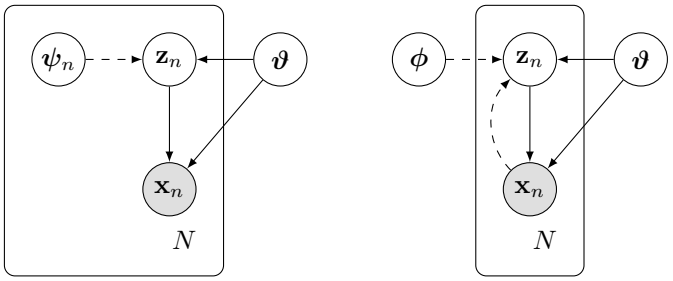
Reparameterization gradients will be seen in action in Chapter 4, since they are, along with *amortized inference*, the key idea behind *variational autoencoders*, the arrival point of this thesis.

### 2.3.3 Amortized inference

The term amortized inference refers to reusing past inferences to support future computations, mirroring human probabilistic reasoning (Gershman & Goodman, 2014).

Consider a hierarchical model where each data point $x_n$ is associated with a local latent variable $z_n$, which is assumed to be governed by its own free variational parameters $\boldsymbol{\psi}_n$. As Figure 2.6(a) illustrates, standard BBVI requires freely optimizing the parameters $\boldsymbol{\psi}_n$ for each data point, which is computationally expensive. The idea behind amortized variational inference is to learn a function $\boldsymbol{\psi}_n = f_{\boldsymbol{\phi}}(x_n)$ to predict $\boldsymbol{\psi}_n$ from the input $x_n$. In this way, as Figure 2.6(b) shows, the local variational parameters are replaced by

a function of $x_n$ whose parameters $\boldsymbol{\phi}$ are shared across all data points, i.e. inference is amortized (C. Zhang et al., 2018). At this point, one can obtain fast estimates of local parameters on future data by feeding them into the function $f_{\boldsymbol{\phi}}$, without having to run other optimizations.




**(a)** BBVI **(b)** Amortized BBVI

**Figure 2.6:** Graphical representation of a hierarchical model with mean field variational approximations, standard and amortized respectively. Dashed lines denote variational approximations and $\boldsymbol{\vartheta}$ are the global variables (parameters) of the model.

The function $f_{\boldsymbol{\phi}}$ is usually chosen to be a deep neural network and is called *inference network* or *recognition network*. To this extent, amortized VI with inference networks combines probabilistic modeling with the power of deep learning (C. Zhang et al., 2018). Examples of that are *deep Gaussian processes* (Damianou & Lawrence, 2013) and *deep latent Gaussian models* (DLGMs) (Rezende et al., 2014), which lead to *variational autoencoders* (VAEs) (Kingma & Welling, 2014).

The idea of amortization will then be taken up again in Chapter 4, but in order to better understand what follows, we first need to illustrate some concepts about the theory of neural networks.

# Chapter 3

# Artificial Neural Networks

*Artificial neural networks*, simply called neural networks in machine learning parlance, are a class of computational systems inspired by the biological neural networks that constitute human brains. In more detail, a neural net is a network model consisting of interconnected units or nodes called (artificial) *neurons*, which transmit information signals to one another, loosely modeling the neurons in a biological brain. Depending on the typologies of neurons and how they are connected, there are different types of neural networks. In this chapter we will focus on *feedforward neural networks* and then specifically on *convolutional neural networks* and *autoencoders*.

## 3.1 Deep feedforward neural networks

Feedforward neural networks (FFNNs), also known as *multilayer perceptrons* (MLPs), are the quintessential deep learning models (Goodfellow et al., 2016). The goal of a FFNN is to approximate an arbitrary function $f^*$ through a hierarchical composition of nonlinear functions. Operationally, this hierarchical structure is expressed as a network where the nodes (called *units* or *neurons*) are organized in *layers* and connected by edges, each of which is associated with a parameter that is called *weight*. The name *feedforward* comes from the fact that in this network the information flows in only one direction, from the *input nodes*, through the *hidden layers* and to

the *output nodes*; the connections between nodes do not form cycles, which instead characterize *recurrent neural networks*, not presented in this thesis. The layered structure of a FFNN is best captured with a directed acyclic graph (DAG) representation, as illustrated by Figure 3.1.

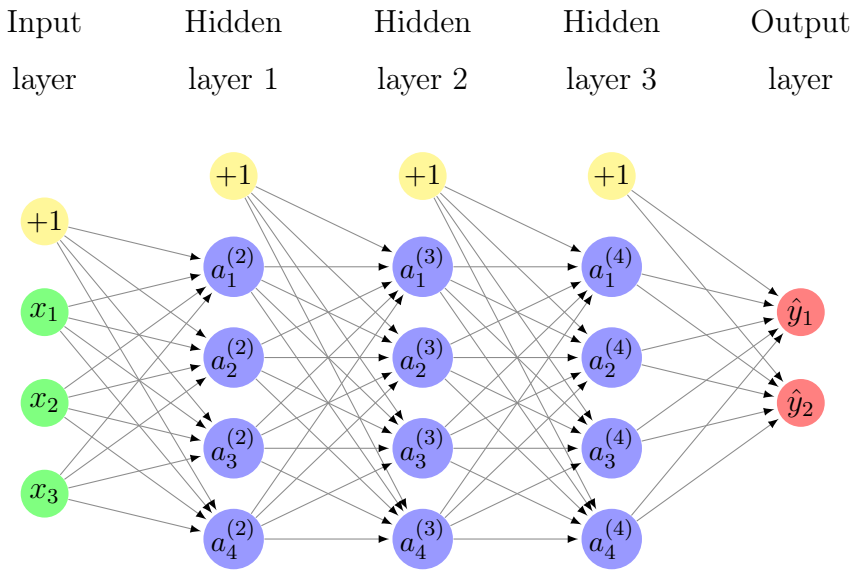



**Figure 3.1:** Structure of a deep feedforward neural network with three hidden layers; the yellow nodes represent the intercepts or *bias* units.

Referring to the latter, we then want to define the mathematical formulation of multilayer perceptrons. Adapting from (Efron & Hastie, 2016), we introduce a element-wise notation, defining:

- $L$ : number of layers in the network, which has one input layer, $L-2$ hidden layers and one output layer;
- $p_\ell$ : number of nodes in the $\ell$-th layer (excluding the bias unit);
- $a_j^{(\ell)}$ : value of the $j$-th node in the $\ell$-th layer, with the $i$-th input unit being defined as $a_i^{(1)} = x_i$ , and the $k$-th output unit as $a_k^{(L)} = \hat{y}_k$ ;
- $w_{ji}^{(\ell)}$ : weight associated with the connection between node $i$ in layer $\ell$ and node $j$ in layer $\ell + 1$;
- $b_j^{(\ell)}$ : bias associated with node $j$ in layer $\ell + 1$.

With this notation in hand, the relation between the layer $\ell$ and the layer $\ell+1$ is defined as

$$\begin{aligned} z_j^{(\ell+1)} &= b_j^{(\ell)} + \sum_{i=1}^{p_\ell} w_{ji}^{(\ell)}\, a_i^{(\ell)}, \\ a_j^{(\ell+1)} &= g^{(\ell+1)}(z_j^{(\ell+1)}), \end{aligned} \tag{3.1}$$

that is a linear combination of the nodes in the previous layer transformed by a usually nonlinear function $g^{(\ell+1)}(\cdot)$, which is the *activation function* for the $(\ell+1)$-th layer.

This formulation can be made more streamlined by adopting a vector notation. We define:

- $\mathbf{a}^{(\ell)} = \begin{bmatrix} a_1^{(\ell)} & a_2^{(\ell)} & \cdots & a_{p_\ell}^{(\ell)} \end{bmatrix}^T$;
- $\mathbf{W}^{(\ell)} = \begin{bmatrix} \mathbf{w}_1^{(\ell)} & \mathbf{w}_2^{(\ell)} & \cdots & \mathbf{w}_{p_\ell}^{(\ell)} \end{bmatrix}$, with $\mathbf{w}_i^{(\ell)} = \begin{bmatrix} w_{1i}^{(\ell)} & w_{2i}^{(\ell)} & \cdots & w_{p_{\ell+1}i}^{(\ell)} \end{bmatrix}^T$;
- $\mathbf{b}^{(\ell)} = \begin{bmatrix} b_1^{(\ell)} & b_2^{(\ell)} & \cdots & b_{p_{\ell+1}}^{(\ell)} \end{bmatrix}^T$;
- $\mathbf{W} = \begin{bmatrix} \mathbf{W}^{(1)} & \mathbf{W}^{(2)} & \cdots & \mathbf{W}^{(L-1)} \end{bmatrix}$;
- $\mathbf{b} = \begin{bmatrix} \mathbf{b}^{(1)} & \mathbf{b}^{(2)} & \cdots & \mathbf{b}^{(L-1)} \end{bmatrix}$.

At this point, the relation between two generic layers can be expressed in the compact form

$$\begin{aligned} \mathbf{z}^{(\ell+1)} &= \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell)} + \mathbf{b}^{(\ell)}, \\ \mathbf{a}^{(\ell+1)} &= g^{(\ell+1)}(\mathbf{z}^{(\ell+1)}), \end{aligned} \tag{3.2}$$

where the activation function $g^{(\ell+1)}(\cdot)$ is applied to the vector $\mathbf{z}^{(\ell+1)}$ in an element-wise fashion. Consequently, given an input $\mathbf{x}$ and the parameters $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$, the network aims to approximate an unknown function $f^*(\mathbf{x})$ through the chain relation,

$$\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta}) = g^{(L)}(\mathbf{W}^{(L-1)} g^{(L-1)}(\cdots \mathbf{W}^{(2)} g^{(2)}(\mathbf{W}^{(1)}\mathbf{x}))), \tag{3.3}$$

in which the bias terms are omitted for simplicity. We have that the neural network is generally defined *deep* if the number of hidden layers is greater than or equal to 3.

### 3.1.1 Non-linearities

Regarding the activation functions $g(\cdot)$, we notice that if only linear functions are used, then the model (3.3) is equivalent to a linear model. For this reason, in order to approximate complex relations $f^*(\mathbf{x})$, the same nonlinear activation functions are commonly used for the hidden layers, while for the output layer the transformation $g^{(L)}(\cdot)$ is chosen according to the problem of interest. If we are facing a quantitative regression problem, the single output node is the result of a linear transformation $\hat{y} = z^{(L)}$; on the other hand, if we are in a classification context with $K$ classes, the activation function for the output layer is typically the *softmax* function

$$g^{(L)}(\mathbf{z}_k^{(L)}) = \frac{e^{\mathbf{z}_k^{(L)}}}{\sum_{j=1}^{K} e^{\mathbf{z}_j^{(L)}}}. \tag{3.4}$$

With regard to activation functions for hidden layers, in the early days of neural networks common choices were the *sigmoid* function or the *hyperbolic tangent*, which are presented in Table 3.1. However, as Figure 3.2 (left) illustrates, the sigmoid function saturates at 1 for large positive inputs and at 0 for large negative inputs. The same happens to the tanh function, which has a similar shape but horizontal asymptotes $\pm 1$. Consequently, as can be seen from Figure 3.2 (right), the gradients of both functions become small and tend to zero as $|z|$ increases, being in general always smaller than one.

This makes it problematic to estimate parameters $\boldsymbol{\theta}$ via *backpropagation*, which, as will become clear later, exploits the chain rule to compute the gradient of the output w.r.t. $\boldsymbol{\theta}^{(\ell)}$, by recursively multiplying the activation functions gradients and intermediate quantities related to higher layers. If the gradients of the activation functions take on small values, it will then result in the gradient signal from higher layers being unable to propagate back to earlier layers, especially for very deep neural networks. This is known as *vanishing gradient* problem (Bengio et al., 1994), and leads to infinitesimal parameter updates, making the gradient-based learning process difficult.

One solution to this issue is to use *rectified linear unit* (ReLU), the activation function recommended for deep networks, defined in Table 3.1.

**Table 3.1:** Most popular activation functions for neural networks. Formally, ReLU and leaky ReLU are not differentiable at 0, but by convention their gradients are set respectively equal to 0 and $\alpha$ when $z = 0$.

| Name | Definition | Range | Gradient |
|---|---|---|---|
| Sigmoid | $\sigma(z) = (1 + e^{-z})^{-1}$ | $[0, 1]$ | $\sigma(z)\,(1 - \sigma(z))$ |
| Hyperbolic tangent | $\tanh(z) = 2\sigma(2z) - 1$ | $[-1, 1]$ | $1 - \tanh^2(z)$ |
| Rectified linear unit | $\mathrm{ReLU}(z) = \max(0, z)$ | $[0, \infty]$ | $\mathbb{1}_{\{z>0\}}$ |
| Leaky ReLU | $\max(0, z) + \alpha \min(0, z)$ | $[-\infty, \infty]$ | $\mathbb{1}_{\{z>0\}} + \alpha \mathbb{1}_{\{z\leq 0\}}$ |

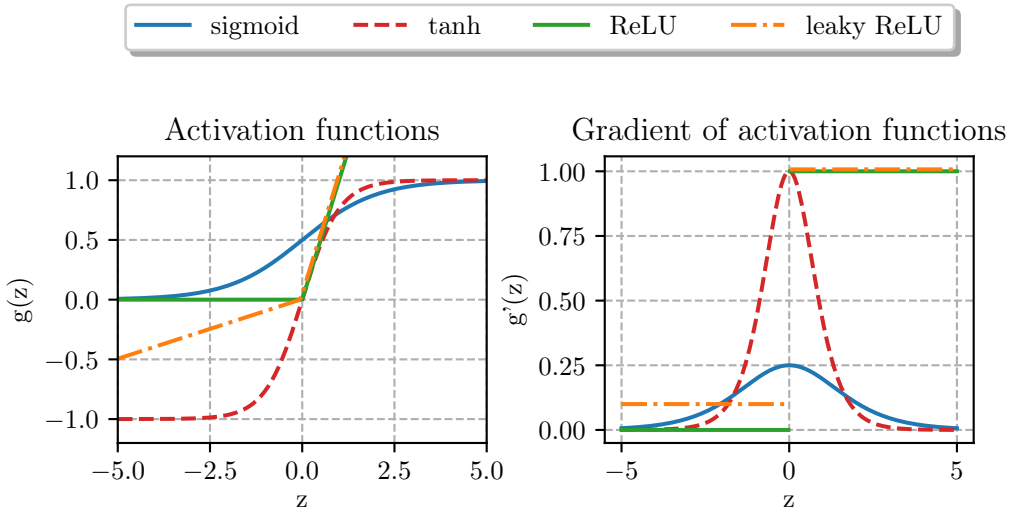



**Figure 3.2:** Plots of the most used activation functions (left) and their gradients (right); we set $\alpha = 0.1$ for leaky ReLU.

As Figure 3.2 (left) shows, ReLU (Glorot et al., 2011) is a non-saturating activation function, which introduces non-linearity through a piecewise linear transformation. In practice, (Krizhevsky et al., 2012) have shown empirically that using the rectifier activation function leads to better convergence performance. In fact, ReLU has the following major advantages:

- it reduces the likelihood of gradient vanishing. The function *activates* only those hidden units with $z > 0$, to which correspond gradients that have constant value 1 (Table 3.1), and can thus be propagated back to earlier layers without becoming increasingly small. In contrast, the

remaining hidden units (with $z \leq 0$) become exactly equal to 0, and thus have zero gradients that are not backpropagated (Figure 3.3).

- It is computationally efficient, as it only requires comparing whether a number is positive, and the gradient is equal to 0 or 1.

- The rectifier function allows the network to obtain sparse representations, since about 50% of the hidden neurons return real zeros and are therefore *inactive*, as shown by Figure 3.3. We will often see that sparse representations are more beneficial than dense ones for a number of reasons, including the fact that they are prone to disentangle the underlying factors of variation in the data.

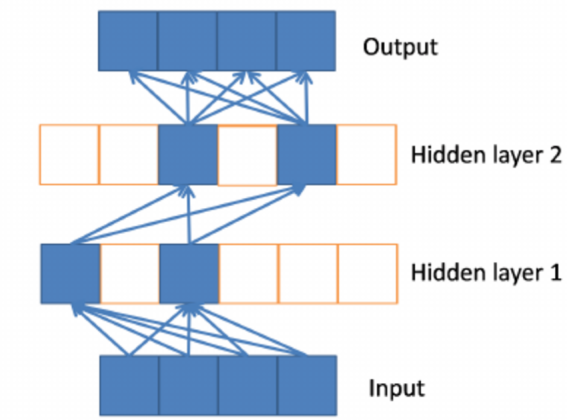



**Figure 3.3:** Sparsity in a network with ReLU activations. Active neurons are transformed linearly; the non-linearity in the network arises from the path selection (combination) associated with individual neurons being active or not (Glorot et al., 2011).

Nevertheless, ReLU neurons can end up in a state in which they are always inactive and output zero for all inputs. In this situation, the neuron is unlikely to recover, since no gradients flow backward through it, and "dies", being not able to discriminate between inputs. This is known as *dying ReLU* problem. One attempt to solve this issue is to use the *leaky ReLU* activation function, which assigns a small positive slope for $z \leq 0$, usually $\alpha = 0.01$ (Maas et al., 2013).

### 3.1.2 Why deep representations

FFNNs are generally referred to as universal approximators, in the sense that a multilayer perceptron with one hidden layer can approximate any continuous function to an arbitrary level of accuracy, given enough hidden units. This is stated in the *universal approximation theorem* enunciated by (Cybenko, 1989). Unfortunately, in the worst-case scenario, an exponential number of hidden units may be necessary in order to represent a function. This can be easily seen in the binary case: the number of all possible Boolean functions on vectors $\boldsymbol{v} \in \{0,1\}^n$ is $2^{2^n}$, and in general representing one such function requires $2^n$ bits with $O(2^n)$ degrees of freedom (Goodfellow et al., 2016). Therefore, a shallow network with only one hidden layer is sufficient to memorize any function, but the layer may be exaggeratedly wide and generalize poorly to new data.

However, many arguments, both empirical (e.g. Goodfellow et al., 2014) and theoretical (e.g. Montufar et al., 2014), have shown that using very deep neural networks feasibly reduces the number of units-per-layer required to learn a function, and leads to better generalization for a variety of tasks. The reason is that higher layers can leverage the features learned by earlier layers, allowing the network to discover hierarchical representations of high-dimensional data with multiple levels of abstraction.

Consider, for example, a face image, which arrives in the form of an array of pixel values. As illustrated by Figure 3.4, the first hidden layer of the network typically learns to detect edges at various orientations. Then, the second hidden layer can compose these edges to detect primitive shapes, here eyes, noses, ears, etc. The third hidden layer can assemble the previous representations to identify more complex shapes and objects, in this case particular types of faces.

In general, subsequent layers gradually learn to extract more abstract level features that are invariant to irrelevant variations in the data, and can be used to discriminate with respect to the problem of interest, which for this example may be facial recognition or some type of classification.

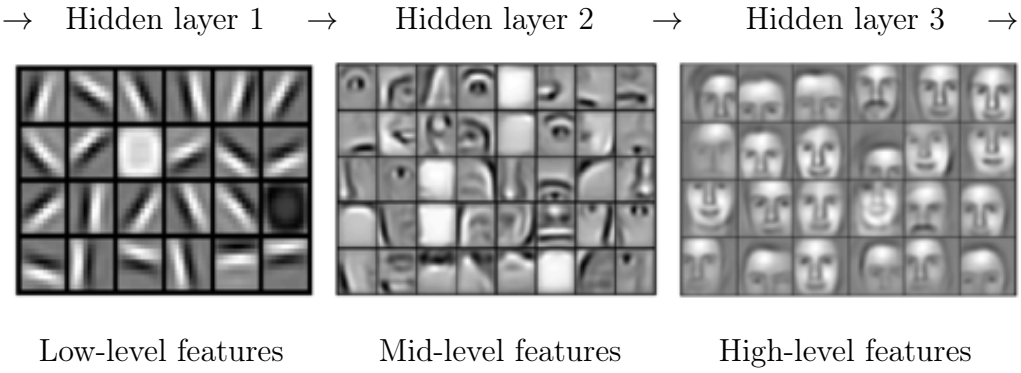



**Figure 3.4:** Visualizations of features learned by a convolutional neural net, a specific type of FFNN suitable for images; each box represents one specific hidden unit (Lee et al., 2011). Incidentally, it was ascertained that these features resemble those observed in areas V1 and V2 of the visual cortex.

As stated by (LeCun et al., 2015), the crucial aspect of deep learning is that these features are not hand-designed by engineers, but they are automatically discovered by the algorithm from the raw data. This approach is known as *representation learning* (Bengio et al., 2013), and aims to disentangle the underlying factors of variation in the data, which in face images may be pose, illumination, identity, facial expression. Deep learning methods are a natural way to accomplish this goal by obtaining abstract representations as a composition of simpler concepts. This makes deep neural networks the state-of-the-art models in a variety of problems, such as speech recognition, image recognition and natural language processing.

### 3.1.3 Backpropagation

Suppose we are in a supervised learning setting, which means we have access to a labeled training set $\left\{(x_i, y_i)_{i=1}^n\right\}$, where $x_i = (x_{i1}, \ldots, x_{ip_1})$ are the input variables and $y_i$ is the response variable, which represents noisy examples of the unknown function we want to learn, namely $y_i \approx f^*(x_i)$. The nature of the label $y_i$ is quantitative for a regression problem, while it is qualitative with $K$ modalities in a classification context.

We are therefore interested in understanding how the parameters $\boldsymbol{\theta}$ of the neural network are estimated in such a way as to minimize a certain loss function $L(\boldsymbol{\theta})$. The process first involves flowing the input $\mathbf{x}$ through the various layers of the network to produce the output $\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta})$: this is known as *forward propagation*. Next, the *backpropagation* algorithm allows to compute the gradients $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ by propagating the information backward in the network. These gradients can then be exploited in a gradient-based optimization algorithm to update the parameters.

A common choice of loss function for regression problems is the *mean squared error* (MSE):

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L_i = \frac{1}{n} \sum_{i=1}^{n} \big(y_i - f(x_i; \boldsymbol{\theta})\big)^2. \tag{3.5}$$

In contrast, for classification problems one generally uses the *cross-entropy*, that is the negative log-likelihood of the multinomial distribution,

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L_i = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log f_k(x_i; \boldsymbol{\theta}), \tag{3.6}$$

where $y_{ik} = 1$ if the $i$-th response belongs to the class $k$, and 0 otherwise.

Our goal is to find the parameters $\boldsymbol{\theta}$ (formed by the weights $\mathbf{W}$ and the biases $\mathbf{b}$) that minimize the loss function $L(\boldsymbol{\theta})$. This is done through the training procedure we outlined in broad strokes earlier, in which the crucial step is the *backpropagation* algorithm, or *backprop* (Rumelhart et al., 1986). In fact, the latter makes it possible to efficiently compute the derivatives of the loss function w.r.t. the parameters by recursively applying the chain rule for differentiation.

Suppose we have forward propagated a single input $x_i$ to obtain $f(x_i; \boldsymbol{\theta})$, having stored underway all the intermediate quantities $\mathbf{a}^{(\ell)}$ and $\mathbf{z}^{(\ell)}$ for the hidden layers. Denoting by $L_i$ the contribution of the $i$-th example to the loss function, we can define the delta terms

$$\boldsymbol{\delta}^{(l)} = \frac{\partial L_i}{\partial \, \mathbf{z}^{(\ell)}}, \tag{3.7}$$

which measure the responsibility of each node for the error in predicting the $y_i$ response (Efron & Hastie, 2016).

Applying the chain rule, we can calculate the quantities $\boldsymbol{\delta}^{(L)}$ for the output layer as

$$\begin{aligned}\boldsymbol{\delta}^{(L)} &= \frac{\partial L_i}{\partial \mathbf{z}^{(L)}} = \frac{\partial L_i}{\partial f(x_i;\boldsymbol{\theta})}\frac{\partial f(x_i;\boldsymbol{\theta})}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial L_i}{\partial f(x_i;\boldsymbol{\theta})} \odot \dot{g}^{(L)}(\mathbf{z}^{(L)}),\end{aligned} \tag{3.8}$$

where $\dot{g}^{(L)}(\mathbf{z}^{(L)})$ indicates the first derivative of $g^{(L)}(\mathbf{z}^{(L)})$ and the symbol $\odot$ denotes the element-wise product operator (or Hadamard product). The first term $\partial L_i/\partial f(x_i;\boldsymbol{\theta})$ is, for example, equal to $2\,(y_i - f(x_i;\boldsymbol{\theta}))$ when using MSE in a regression problem.

Proceeding backward recursively, we can obtain the deltas for the generic layer $\ell$ as

$$\begin{aligned}\boldsymbol{\delta}^{(\ell)} &= \frac{\partial L_i}{\partial \mathbf{z}^{(\ell)}} = \frac{\partial L_i}{\partial \mathbf{z}^{(\ell+1)}}\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{a}^{(\ell)}}\frac{\partial \mathbf{a}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}} \\ &= \boldsymbol{\delta}^{(\ell+1)}\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{a}^{(\ell)}}\frac{\partial \mathbf{a}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}} \\ &= \left((\mathbf{W}^{(\ell)})^T\boldsymbol{\delta}^{(\ell+1)}\right) \odot \dot{g}^{(\ell)}(\mathbf{z}^{(\ell)}).\end{aligned} \tag{3.9}$$

With these results it is possible to compute the derivatives of the loss function with respect to the parameters, namely

$$\frac{\partial L_i}{\partial \mathbf{W}^{(\ell)}} = \frac{\partial L_i}{\partial \mathbf{z}^{(\ell+1)}}\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell+1)}(\mathbf{a}^{(\ell)})^T, \tag{3.10}$$

$$\frac{\partial L_i}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial L_i}{\partial \mathbf{z}^{(\ell+1)}}\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell+1)}. \tag{3.11}$$

In practice, the calculation is done by automatic differentiation expressing the differentiable components that characterize the neural network as a computation graph (Baydin et al., 2018).

### 3.1.4 Stochastic gradient descent

The derivatives (3.10) and (3.11) can therefore be used in a *gradient descent* optimization algorithm to iteratively find the parameter values that minimize the loss function. Define

$$\nabla\boldsymbol{\theta}^{(\ell)} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial \boldsymbol{\theta}^{(\ell)}} \tag{3.12}$$

to be the average gradient of an iteration over all the training examples. Then, the updates for the parameters $\boldsymbol{\theta}$, consisting of $\mathbf{W}$ and $\mathbf{b}$, are

$$\boldsymbol{\theta}^{(\ell)} \leftarrow \boldsymbol{\theta}^{(\ell)} - \eta \nabla \boldsymbol{\theta}^{(\ell)}, \tag{3.13}$$

where $\eta \in (0, 1)$ is the learning rate.

In practice, for the same reasons explained in Section 2.2, it is convenient to solve our non-convex optimization problem by *stochastic gradient descent* (SGD). This involves updating the parameters each time we process a new training sample, as illustrated by Algorithm 9. Each parameter is usually randomly initialized via the Xavier initialization,

$$\theta \sim \mathcal{N}\Big(0, \frac{2}{p_1 + p_L}\Big), \quad p_1 \text{ num of inputs and } p_L \text{ num of outputs}, \tag{3.14}$$

so as to break the symmetry in the network that would occur if all weights were initialized to zero. In addition, for numerical reasons, it is good practice to standardize the input variables so that they have zero mean and unit variance: this helps to accelerate convergence.

The approach presented in Algorithm 9 can be easily extended to mini-batches by appropriately rescaling the gradient (3.12) as seen in the previous chapter. In effect, *mini-batch gradient descent* is the go-to algorithm for training large datasets, as it allows us to: (i) take advantage of vectorization to efficiently perform computations on $B$ data points; (ii) process multiple minibatches in parallel on different machines.

The batch size $B$, as well as the neural network architecture (number of layers, number of hidden units, activation functions), should be chosen in such a way as to minimize the prediction error on a validation set, so as to trade-off bias and variance.

### 3.1.5 Adaptive moment estimation

An extension that accelerates stochastic gradient descent is the method of *momentum*, which aims to move faster in the right direction leading to the minimum of the loss function, just like a ball rolling downhill. This is accomplished by smoothing the gradient through an exponentially weighted

**Algorithm 9:** Parameter estimation in a neural network

**Initialize:** parameters $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$

**while** *not converged* **do**

  **for** $i = 1 : n$ **do**

    Perform forward propagation of $x_i$, obtaining the quantities

    $$\mathbf{a}^{(\ell)}, \mathbf{z}^{(\ell)} \text{ for } \ell = 2, \ldots, L$$

    For the output layer, set

    $$\boldsymbol{\delta}^{(L)} = \frac{\partial L_i}{\partial f(x_i;\boldsymbol{\theta})} \odot \dot{g}^{(L)}(\mathbf{z}^{(L)})$$

    For the hidden layers $\ell = L-1, \ldots, 2$, set

    $$\boldsymbol{\delta}^{(\ell)} = \left((\mathbf{W}^{(\ell)})^T \boldsymbol{\delta}^{(\ell+1)}\right) \odot \dot{g}^{(\ell)}(\mathbf{z}^{(\ell)})$$

    Compute the gradient

    $$\nabla \boldsymbol{\theta}^{(\ell)} = \begin{cases} \nabla \mathbf{W}^{(\ell)} = \boldsymbol{\delta}^{(\ell+1)} (\mathbf{a}^{(\ell)})^T \\ \nabla \mathbf{b}^{(\ell)} = \boldsymbol{\delta}^{(\ell+1)} \end{cases}$$

    Update the current parameters

    $$\boldsymbol{\theta}^{(\ell)} \leftarrow \boldsymbol{\theta}^{(\ell)} - \eta \, \nabla \boldsymbol{\theta}^{(\ell)}$$

  **end**

**end**

moving average, which is accumulated by the algorithm so as to avoid unnecessary oscillations. Defining $\theta_t$ the generic scalar parameter of $\boldsymbol{\theta}$ at iteration $t$, and introducing the variable $v_t$ that plays the role of velocity, the implementation is as follows:

$$\begin{aligned} v_t &= \beta_1 \, v_{t-1} + \nabla \theta_t \\ \theta_t &= \theta_{t-1} - \eta_t \, v_t, \end{aligned} \tag{3.15}$$

where the constant $\beta_1 \in [0, 1)$ is typically set to 0.9. For $\beta_1 = 0$, we return back to standard SGD.

Another algorithm that improves SGD is RMSProp (Root Mean Square Propagation) (Tieleman & Hinton, 2012), which adapts the learning rate according to the frequency of each parameter. The idea is to first calculate

an exponentially weighted moving average of the past squared gradients as

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2)(\nabla\theta_t)^2, \tag{3.16}$$

and then scale the learning rate by the square root of this moving average, to obtain the update

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}}\nabla\theta_t. \tag{3.17}$$

The small scalar $\epsilon$ (usually $10^{-8}$) prevents division by zero, while $\beta_2$ is typically set to 0.9. RMSProp is an adaptive learning rate algorithm; the overall step size $\eta$ still need to be chosen, but the results are less sensitive to this choice.

Combining momentum and RMSProp we obtain what is the most widely used optimization algorithm for deep networks, namely Adam (Kingma & Ba, 2014), which stands for "adaptive moment estimation" as it is based on adaptive estimates of lower-order moments. Define the moving averages

$$\begin{aligned} v_t &= \beta_1\, v_{t-1} + (1 - \beta_1)\nabla\theta_t \\ s_t &= \beta_2 + (1 - \beta_2)(\nabla\theta_t)^2. \end{aligned} \tag{3.18}$$

Since if we initialize $v_0 = s_0 = 0$ the initial estimates are biased toward small values, it is appropriate to use the following bias-corrected moments:

$$\widetilde{v_t} = \frac{v_t}{1 - \beta_1^t} \qquad \widetilde{s_t} = \frac{s_t}{1 - \beta_2^t}\,. \tag{3.19}$$

At this point the update becomes:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\widetilde{s_t}} + \epsilon}\,\widetilde{v_t}, \tag{3.20}$$

where the recommended values for the constants are $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$, while a good default value for the overall learning rate is $\eta = 0.001$. The algorithm is fairly robust to this choices, albeit sometimes $\eta$ needs to be tuned.

Empirical results of (Kingma & Ba, 2014) show that Adam works well in practice and compares favorably to other optimization algorithms, without needing to costly store the Hessian as in second-order methods.

An additional technique that generally allows to improve optimization is *batch normalization* (Ioffe & Szegedy, 2015). It is a method of adaptive reparameterization that standardizes the hidden units in order to speed up learning of mini-batch gradient descent. More specifically, in any layer $\ell$ (omitted), the pre-activation vector $\mathbf{z}^{\{m\}}$ for data point $m$ is transformed, by exploiting broadcasting, as follows:

$$\begin{aligned} \widehat{\mathbf{z}}^{\{m\}} &= \frac{\mathbf{z}^{\{m\}} - \boldsymbol{\mu}_b}{\sqrt{\boldsymbol{\sigma}_b^2 + \epsilon}}, \\ \boldsymbol{\mu}_b &= \frac{1}{B} \sum_{\mathbf{z} \in b} \mathbf{z}, \\ \boldsymbol{\sigma}_b^2 &= \frac{1}{B} \sum_{\mathbf{z} \in b} (\mathbf{z} - \boldsymbol{\mu}_b)^2, \end{aligned} \tag{3.21}$$

where $b$ is the minibatch of cardinality $B$ containing data point $m$, $\boldsymbol{\mu}_b$ is the mean of $\mathbf{z}$ in that minibatch and $\boldsymbol{\sigma}_b^2$ the corresponding variance, with $\epsilon$ very small constant. In order to maintain the expressive power of the neural network, the vector $\widehat{\mathbf{z}}^{\{m\}}$ is further transformed to give:[1]

$$\widetilde{\mathbf{z}}^{\{m\}} = \boldsymbol{\gamma} \odot \widehat{\mathbf{z}}^{\{m\}} + \boldsymbol{\delta}, \tag{3.22}$$

where $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$ are learnable location and scale parameters of the considered layer, which are in addition to the other model parameters. Since the transformation (3.22) is differentiable, we can back-propagate the gradient w.r.t. $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$ in order to optimize these parameters, for example via Adam.

The reasons why batch normalization works well in practice are still under debate, but intuitively this is because it makes the distribution of every layer more stable with respect to variations in its inputs; allowing us to use higher learning rates and be less careful about weight initialization. In addition, since the noise introduced into the transformations by the minibatch statistics forces the learning process to be more robust, *batchnorm* has also a slight regularization effect.

[1] Forcing the pre-activations to take on a unit Gaussian distribution could constrain the outputs of the activation functions to the linear regime of the non-linearities (think about sigmoid function) and limit the flexibility of the neural network.

### 3.1.6 Regularization strategies

In fact, deep neural networks are highly parameterized and complex models that easily run the risk of the so-called *overfitting* phenomenon, which occurs when the model follows too closely local characteristics of the training data that are not replicated on new data. For this reason, the selection of a good model should not be based solely on the training data, but should aim to minimize the prediction error on a validation dataset not used for fitting. In other words, we need to find a trade-off between bias and variance: the variance of the model increases as the number of parameters grows, because the model tends to chase fluctuations in the data; conversely, the bias decreases as model complexity increases.

Regularization strategies aim to prevent overfitting (i.e. reduce variance) by introducing some modification in the training algorithm, in a way that allows the model to best generalize to new data sets. Specifically, in a deep learning context, we almost always find that the best model, in terms of prediction error on the validation dataset, is a large neural network that has been heavily regularized (Goodfellow et al., 2016). In this direction, we first focus on the following forms of regularization:

- early stopping;
- L2 regularization;
- dropout.

Early stopping is the simplest regularization strategy: it consists of stopping the training process when the minimum error on the validation set is reached. Indeed, for models with sufficient representational power we usually have that the training error gradually decreases over time, while the validation set error begins to rise from a certain epoch, giving life to an asymmetric U-shaped curve. Thus, by progressively storing a copy of the best parameters until the validation error increases consistently, we can obtain the model that predicts best on the validation set, and hopefully also on an additional set of unseen data known as the *test set*. This means that we

can prevent overfitting at the same time as training, without having to tune regularization parameters (or if one wants, having to tune only the number of epochs). Therefore, early stopping is an ad-hoc heuristic, but it works effectively in practice, provided an appropriate stopping criterion.

Another way to avoid overfitting are penalization methods, which constrain the model to be more regular by artificially shrinking its parameters toward zero, in order to reduce its variability by accepting some bias. This is done minimizing the regularized loss function defined as:

$$\widetilde{L}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda\,\Omega(\mathbf{W}), \tag{3.23}$$

where $\lambda \geq 0$ is a regularization parameter that weights the contribution of $\Omega(\mathbf{W})$, a norm penalty term that measures the complexity of the model. The parameter $\lambda$ should be chosen so as to minimize the validation error, with larger values of $\lambda$ corresponding to a bigger regularization effect. As can be seen, the bias parameters are not penalized.

The most common penalty for neural nets is the squared Frobenius norm,

$$\Omega(\mathbf{W}) = \frac{1}{2}\,\|\mathbf{W}\|_F^2 = \frac{1}{2}\sum_{\ell=1}^{L-1}\sum_{i=1}^{p_\ell}\sum_{j=1}^{p_{\ell+1}}\left(w_{ji}^{(\ell)}\right)^2, \tag{3.24}$$

which is known as *weight decay* and gives rise to the so-called L2 regularization, related to ridge regression. Assuming that the network has no bias parameters, the regularized loss function becomes

$$\widetilde{L}(\boldsymbol{W}) = L(\boldsymbol{W}) + \frac{\lambda}{2}\sum_{\ell=1}^{L-1}\sum_{i=1}^{p_\ell}\sum_{j=1}^{p_{\ell+1}}\left(w_{ji}^{(\ell)}\right)^2, \tag{3.25}$$

with the corresponding gradient that has generic element equal to

$$\frac{\partial \widetilde{L}}{\partial w_{ji}^{(\ell)}} = \frac{\partial L}{\partial w_{ji}^{(\ell)}} + \lambda\, w_{ji}^{(\ell)}. \tag{3.26}$$

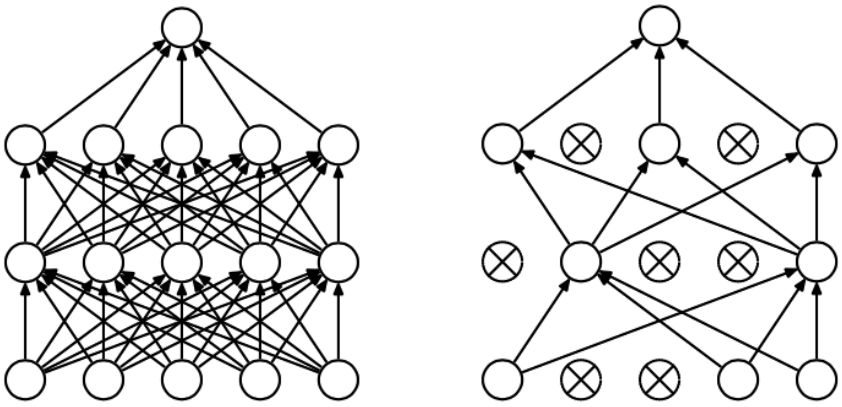
At this point, in minimizing (3.25), the generic update takes the form:

$$\begin{aligned} w_{ji}^{(\ell)} &= w_{ji}^{(\ell)} - \eta\left(\lambda\, w_{ji}^{(\ell)} + \frac{\partial L}{\partial w_{ji}^{(\ell)}}\right) \\ &= (1-\eta\lambda)\, w_{ji}^{(\ell)} - \eta\,\frac{\partial L}{\partial w_{ji}^{(\ell)}}. \end{aligned} \tag{3.27}$$

which shrinks the ordinary update because of the factor $1-\eta\lambda$.

Interestingly, it can be proved (Bishop, 2006) that the estimation with weight decay penalty is equivalent to the MAP (maximum a posteriori) estimation of a Bayesian neural network, with independent priors on the weights of the type $w_{ji}^{(\ell)} \sim \mathcal{N}(0, 1/\lambda)$. Instead, a different penalty term is the L1 norm, that is the sum of the absolute values of the weights, which encourages some weights to be exactly equal to zero.

To conclude, we present a powerful regularization method called *dropout* (Srivastava et al., 2014). It consists of randomly omitting nodes from the neural network at each iteration (sample) of the training process, as illustrated by Figure 3.5. Each node is kept in the network with a probability $p$ that can vary across layers: it is typically 0.8 for input units and 0.5 for hidden units, but can be tuned using a validation set.



**(a)** Standard neural net **(b)** After applying dropout

**Figure 3.5:** Illustration of dropout regularization. (a) Standard neural network with two hidden layers. (b) Example of a thinned net obtained after applying dropout with $p = 0.5$ to the net on the left; crossed units have been dropped out (Srivastava et al., 2014).

In practice, dropout is widely used because it drastically reduces overfitting and improves the test performance of the networks in an elegant and inexpensive way. Intuitively, the reason why this technique works well is that it prevents complex, but brittle, co-adaptations of the network units. In simple terms, each unit learns to work without relying on the presence of other specific units. This resembles the idea behind a *random forest* (Breiman,

2001), which is a bagged ensemble of decision trees.

Formally, the generic weight going out the $\ell$-th layer, in a neural net with dropout, is equal to $w_{ji}^{(\ell)}\epsilon_i^{(\ell)}$, where $\epsilon_i^{(\ell)} \sim \text{Bernoulli}(p)$. As a result, the weights starting from a unit that has been dropped out are set to zero and their gradient is not back-propagated in the network. This multiplicative noise that characterizes dropout improves the robustness of neural networks; the same thing happens, in a much slighter way, with batch normalization.

At test time, we usually turn dropout off to make deterministic predictions. To ensure that the expected input activation to a unit at test time is roughly the same as it was at train time, even though a $p$ percentage of input units is randomly missing on average, we have to multiply the weights by the probability $p$ before making predictions. Another way to achieve the same goal is to divide the post-activation units' values by $p$ during training, which is known as *inverted dropout*.

However, we can also use dropout at test time by averaging multiple networks that share parameters. This is called *Monte Carlo dropout* (Gal & Ghahramani, 2016), and is based on the following predictive distribution:

$$p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{x}, \mathbf{y}) \approx \frac{1}{S}\sum_{s=1}^{S} p(\mathbf{y}^*|\mathbf{x}^*, \hat{\mathbf{W}}_{\{s\}}, \hat{\mathbf{b}}), \tag{3.28}$$

where $\hat{\mathbf{W}}_{\{s\}}$ are the estimated weights where some network units are dropped out based on a sampled Bernoulli noise vector.

In fact, (Gal & Ghahramani, 2016) showed that deep neural networks with dropout can be viewed as a variational approximation to the probabilistic deep Gaussian process (DGP) model (Damianou & Lawrence, 2013). Specifically, for each DGP layer $\ell$, consider the parameters $\{\mathbf{W}_\ell\}_{\ell=1}^{L-1}$ and $\{\mathbf{b}_\ell\}_{\ell=1}^{L-1}$ that mimic those of FFNNs, but this time they are random arrays with standard multivariate normal priors. Define the variational distribution

$$q(\mathbf{W}, \mathbf{b}) = \prod_{\ell=1}^{L-1} q(\mathbf{W}_\ell)\, q(\mathbf{b}_\ell). \tag{3.29}$$

Each $q(\mathbf{W}_\ell)$ factorizes into Gaussian mixtures with two components, i.e.:

$$q(\mathbf{W}_\ell) = \prod_{j=1}^{d_\ell} q(\mathbf{w}_j^{(\ell)}), \quad q(\mathbf{w}_j^{(\ell)}) = p\,\mathcal{N}(\mathbf{m}_j^{(\ell)}, \sigma^2\mathbf{I}_{d_{\ell+1}}) + (1-p)\,\mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I}_{d_{\ell+1}}), \tag{3.30}$$

where $d_\ell$ stands for dimension of the $\ell$-th layer, $\mathbf{m}_j^{(\ell)}$ is a $(d_{\ell+1})$-dimensional mean vector, $\sigma^2$ is a scalar variance and $p$ mimics the dropout probability. For $q(\mathbf{b}_\ell)$ we posit a simple multivariate Gaussian $\mathcal{N}(\mathbf{m}_0^{(\ell)}, \sigma^2 \mathbf{I}_{d_{\ell+1}})$.

We have that the variational approximation can be reparameterized in terms of Bernoulli and standard Gaussian distributions to obtain a stochastic version of the ELBO, which for $\sigma^2 \to 0$ is mathematically equivalent to the objective function that neural nets with dropout aim to minimize. This implies that, in practice, Monte Carlo dropout can perform $S$ stochastic forward propagations of a new input $\mathbf{x}^*$ by randomly dropping out nodes each time, to obtain $S$ empirical samples $\{\mathbf{y}_1^*, \dots, \mathbf{y}_S^*\}$ from an approximate predictive distribution. These samples can then be averaged to obtain a final prediction of which we also have a measure of uncertainty.

What has just been described goes in the direction of Bayesian deep learning, which aims to infer the full posterior distribution over the model parameters, in contrast to the traditional approach that is only interested in point estimates leading to accurate predictions. Representing uncertainty is actually important in deep learning to provide the practitioner with a measure of model confidence, which is not offered by traditional neural nets. To address this, Bayesian neural networks were introduced, which offer a probabilistic interpretation of models and encourage regularization (MacKay, 1992). Specifically, (Neal, 1996) proposed inferring these networks with MCMC methods, while (Graves, 2011) used fully-factorized Gaussian SVI, but these techniques have often proven to be computationally prohibitive or inaccurate in practice. For this reason, the probabilistic perspective of neural networks is still an active research topic (see Murphy, 2023, §17 for good insights).

## 3.2 Convolutional neural networks

In this section, we illustrate *convolutional neural networks* (CNNs) (LeCun et al., 1989), a specialized type of FFNNs for processing data with a grid-like topology: for example a colored image $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, which is composed of two-dimensional grids of $H \times W$ pixel intensities in the $C = 3$ RGB color

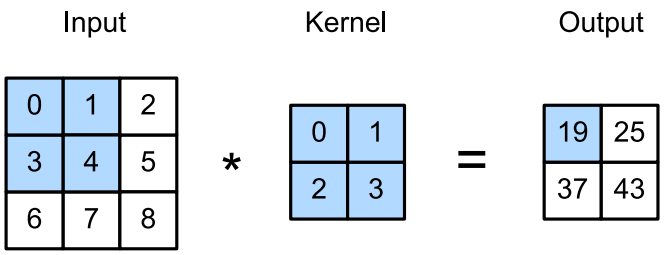



**Figure 3.6:** 2*d* convolution; image taken from (A. Zhang et al., 2021).

channels. CNNs are extremely effective in computer vision applications of image classification and object detection, as they allow spatial information in the array data to be taken into account and require fewer parameters than fully connected architectures. They are based on replacing weight matrix multiplication with a convolution operation.

### 3.2.1 Convolution

In this context, convolution is a kind of linear operation that consists of "sliding" a weight matrix over the image array and add up the results. More formally, in a two-dimensional case, let $\mathbf{X}$ be the input matrix of size $x_h \times x_w$ and $\mathbf{W}$ be a weight matrix of size $f_h \times f_w$ called *filter* or *kernel*. The output of a convolution, $\mathbf{Z} = \mathbf{W} \circledast \mathbf{X}$, is called a *feature map* and is such that:[2]

$$z_{i,j} = [\mathbf{W} \circledast \mathbf{X}](i,j) = \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} w_{u,v}\, x_{i+u,j+v}\,. \tag{3.31}$$

This can best be visualized with Figure 3.6.

Essentially, we can think the filter to detect a certain feature from the input image: the convolution applies the filter to local patches of the image and outputs a high value where the feature is detected. Indeed, in image data, local groups of pixels are often highly correlated, forming special motifs to be discovered.

According to (Goodfellow et al., 2016), convolution leverages three important ideas to improve neural networks: sparse interactions, parameter sharing and equivariant representations.

[2]To be precise, the operation we are defining is cross-correlation, but it is commonly called convolution in deep learning.

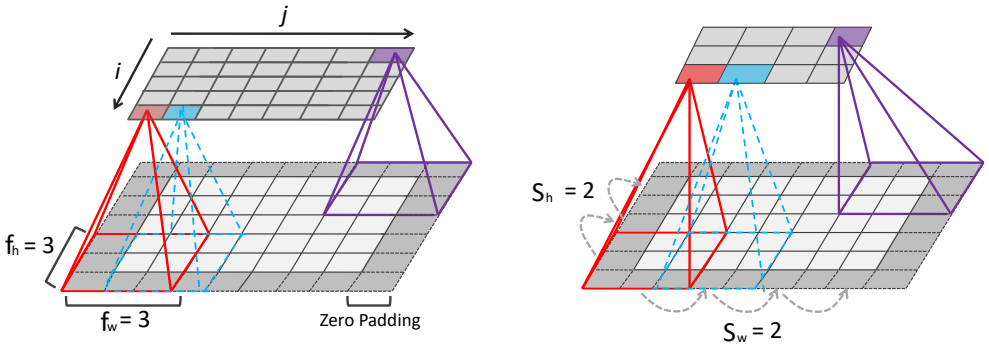
First, CNNs have sparse connections as, in the hidden layers, each output depends only on a restricted set of local inputs called the *receptive field*. In contrast, fully-connected FFNNs are dense since each output neuron can interact with all the input neurons of the previous layer. The sparsity of CNNs is induced by replacing weight matrices multiplications with convolution operations defined by kernels, which are smaller than the inputs and thus are characterized by fewer parameters than dense FFNNs.[3] This improves the statistical efficiency of the model.

Second, parameter sharing refers to the fact that a CNN shares the same parameters of a kernel across different locations of an input; unlike a traditional neural network, which uses each weight only once. The motivation behind parameter sharing is the property of natural images whereby the local statistics of an image are stationary. Accordingly, a feature detector (filter) that is useful in one part of the image is probably useful also in other parts of the image. This helps to dramatically reduce the number of parameters in the network and its memory requirements.

Finally, a property of the convolution operation caused by parameter sharing is *translational equivariance*. From a theoretical standpoint, a function is equivariant if a change in the input causes the output to change in the same way. In terms of convolution, this means that if a pattern in the input shifts location, its representation will translate the same amount in the output.

So far, we have considered a convolution that applies a $f_h \times f_w$ filter to an input of size $x_h \times x_w$ to produce an output of size $(x_h - f_h + 1) \times (x_w - f_w + 1)$, as can be verified in Figure 3.6. This is called a *valid* convolution, since it only uses valid input data. We can use an additional technique called *zero padding*, which adds a border of 0s to the input, as shown by Figure 3.7(a).

[3]The number of parameters of a fully-connected FFNN is computationally prohibitive for image data, since it is equal to $(H \times W \times C) \times D$, where $D$ is the number of hidden units. Consider for example a standard image of dimension $227 \times 227 \times 3$, then fitting a network with two hidden layers of 10 neurons already requires estimating about 15 million parameters.




**(a)** Strides of 1 **(b)** Strides of 2

**Figure 3.7:** Illustration of padding and stride in $2d$ convolutions. (a) We convolve a $5 \times 7$ input using zero padding with a $3 \times 3$ filter to get a $5 \times 7$ output (the dark border is formed by pixels equal to zero). (b) As at left, but with strides of 2. Adapted from (Géron, 2017).

Zero padding the input allows us to (i) control the filter size and the output size independently; (ii) give more importance to the input pixels in the corners. In particular, when enough zero-padding is added to keep the output size equal to the input size, we obtain the so-called *same* convolution.

Another trick of the trade is to downsample the output by skipping over some intermediate positions of the filter. This is illustrated in Figure 3.7(b), where we slide the filter with a *stride* $s = 2$. The purpose is to reduce the computational burden of convolution, but also to limit the redundancy between neighboring outputs.

In general, when we add zero padding on each side of size $p_h$ and $p_w$, and we use a stride $s_h$ for the height and a stride $s_w$ for the width, the size of the output is:

$$\left\lfloor \frac{x_h + 2p_h - f_h + s_h}{s_h} \right\rfloor \times \left\lfloor \frac{x_w + 2p_w - f_w + s_w}{s_w} \right\rfloor, \tag{3.32}$$

with $\lfloor \cdot \rfloor$ the floor function. The formula can be double-checked in Figure 3.7.

In practice, we have to deal with inputs with multiple channels (think about RGB images). Moreover, we want to detect many kinds of features by using multiple filters. These two issues are considered by defining a filter bank $\mathbf{W}$, which is a $4d$ tensor, where $\mathbf{W}_{:,:,c,d}$ is the kernel to detect feature type $d$ in input channel $c$. At this point, the definition of convolution operation can

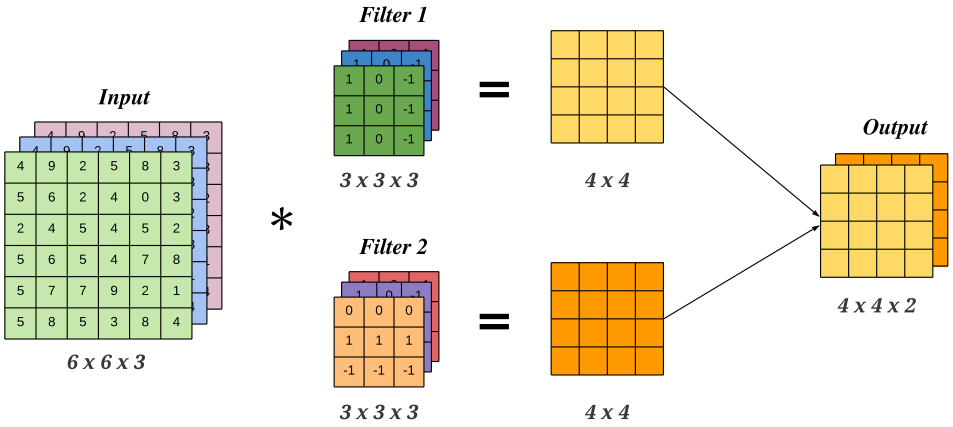



**Figure 3.8:** Convolution with 3 input channels and 2 output channels. Each $2d$ convolution uses a $3 \times 3$ kernel and a stride 1 without padding (bias is omitted), so that the $6 \times 6$ input gets mapped to the $4 \times 4$ output (Murphy, 2023).

be extended so that the $d$-th output feature map is obtained by summing the two-dimensional convolutions over the $C$ input channels, namely:

$$z_{i,j,d} = b_d + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{c=0}^{C-1} x_{si+u,sj+v,c} \, w_{u,v,c,d} \,, \tag{3.33}$$

where $b_d$ is the bias term and $s$ the stride. This is best illustrated by Figure 3.8. Software implementations usually work in minibatch mode, thus the feature maps are represented by means of a four-dimensional tensor with the last axis indexing the minibatch examples.

### 3.2.2 Pooling

The output of a convolution is generally passed through a non-linearity such as ReLU in an element-wise fashion, retaining information about the location of the input in the representation, a property known as translational *equivariance*. In some cases, however, we want to have *invariance* with respect to small translations of the input. For example, when classifying an image, we may want to recognize whether an object of interest is present in the image, regardless of its exact location. In other situations, preserving the location of features may be more crucial.

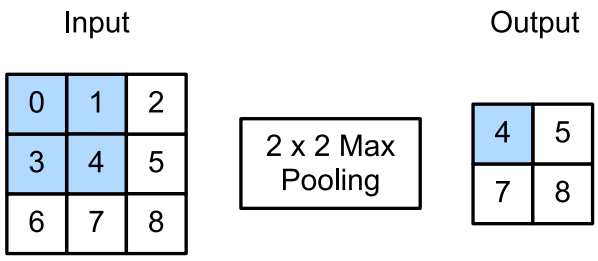



**Figure 3.9:** $2d$ max pooling with a $2 \times 2$ filter (A. Zhang et al., 2021).

The *pooling* operation aims to make the representations of a CNN translationally invariant to location, by coarse-graining the feature maps within local neighborhoods. For example, *max pooling* subsamples a feature map by calculating its maximum value within local patches, as shown by Figure 3.9; another option is *average pooling*, which performs the same process but computing the average. The same considerations about padding and stride also apply here. When dealing with multiple channels, we apply pooling to each feature channel independently: note that pooling does not require learning any parameters.

Putting everything together, as illustrated by Figure 3.10, a typical CNN is composed of alternating convolutional layers with ReLU activation and pooling layers, followed by a final fully-connected classification layer at the end. Backpropagation takes place in a manner similar to that described for traditional neural networks: in practice, using frameworks such as `TensorFlow` and `PyTorch`, we only need to specify the forward propagation structure, then the framework takes care of computing the gradient automatically. Again, the key aspect is that the convolutional filters do not have to be hand-designed, but are learned naturally by a general-purpose procedure, detecting features that gradually refer to larger areas of the image, such as those visualized in Figure 3.4.

As previously announced, deep convolutional networks have been shown to work tremendously well in computer vision tasks. A big motive behind this success is the use of GPUs (graphics processing units) during training: these were originally developed to speed up image rendering for video games, but later became beneficial for performing the tensor operations underlying

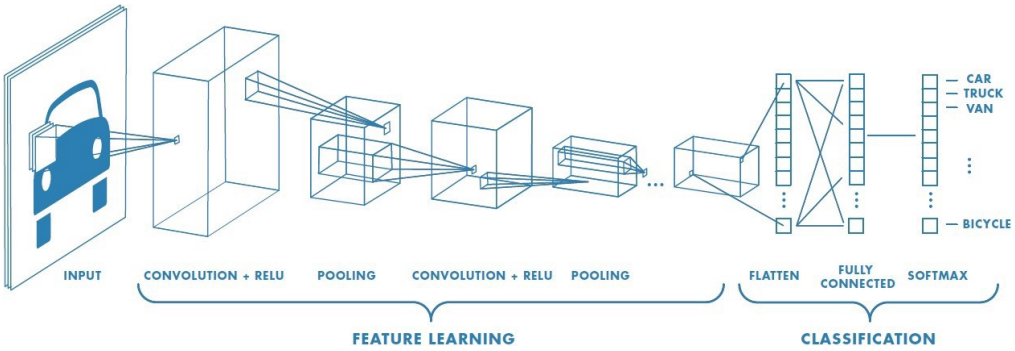



**Figure 3.10:** A simple CNN for image classification. Picture taken from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

deep CNNs, accelerating their training (Murphy, 2022). This idea, along with the introduction of ReLU after convolutional layers and dropout in dense layers, characterized the creation of AlexNet (Krizhevsky et al., 2012), a CNN architecture that marked a breakthrough in deep learning by almost halving the error rate on ImageNet, a benchmark dataset containing million of images from 1000 classes.

Another main reason behind the success of deep CNNs is precisely the availability of large labeled datasets, which serve as fuel for models with thousands of parameters and put more pressure on them to generalize well preventing overfitting. However, we do not always have all the labeled data we wish to solve a given problem. One solution to this is to retrieve the parameters *pre-trained* on a separate data-rich task and then *fine-tune* them on our data-poor problem. This approach is called *transfer learning* as it transfers knowledge across similar vision tasks. Another approach is *dataset augmentation*, which is a regularization technique that randomly distorts input images in order to create additional fake training examples.

## 3.3 Autoencoders

Supervised learning, however, is not the only paradigm for learning representations: there is also unsupervised learning, which aims to discover patterns

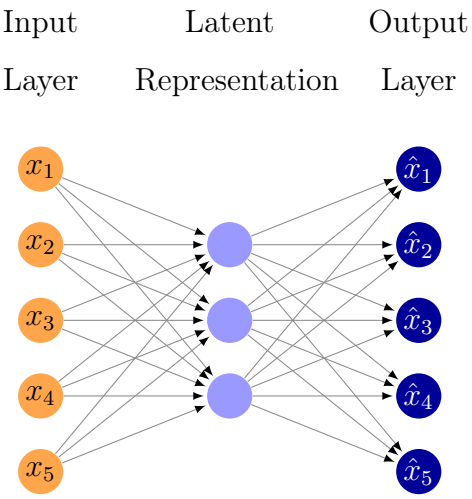


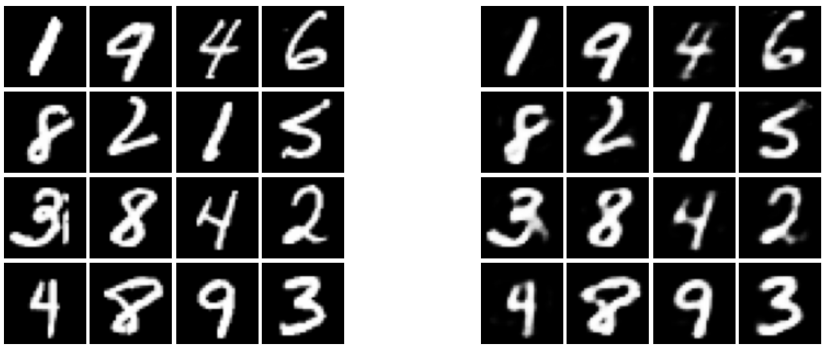

**Figure 3.11:** A simple autoencoder with one hidden layer.

from unlabeled data. In this context, an important model is the *autoencoder*, a particular type of neural network that attemps to reconstuct its inputs learning useful representations of data in its latent space. Specifically, this model is composed of two parts: an *encoder* function $\mathbf{h} = f_e(\mathbf{x})$ that maps inputs into a latent space (*code*), and a *decoder* that produces a reconstruction $\hat{\mathbf{x}} = f_d(\mathbf{h})$. This is illustrated by Figure 3.11.

### 3.3.1 Undercomplete autoencoders

We can think an autoencoder as a nonlinear extension of *principal component analysis* (PCA), which is a dimensionality reduction technique that projects the inputs $\mathbf{x} \in \mathbb{R}^{d_x}$ into a low dimensional subspace $\mathbf{h} \in \mathbb{R}^{d_h}$, characterized by the $d_h$ orthogonal directions of greatest variation in the data. In particular, PCA learns a linear transformation $\mathbf{h} = \mathbf{W}^T\mathbf{x}$, to be mapped back to the original space as $\hat{\mathbf{x}} = \mathbf{W}\mathbf{h}$, in a way that reduces the information loss with respect to the reconstruction error $L(\mathbf{W}) = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2$.

Similarly, an autoencoder involves the nonlinear mappings $\mathbf{h} = f_e(\mathbf{x})$ and $\hat{\mathbf{x}} = f_d(\mathbf{h})$, which are implemented by a neural network with parameters $\boldsymbol{\theta}$ that is trained to minimize the reconstruction error $L(\boldsymbol{\theta}) = \big\|\mathbf{x} - f_d\big(f_e(\mathbf{x})\big)\big\|_2^2$. The latter can also be equal to the cross-entropy loss (3.6), depending on the type of inputs.



**(a)** Original images **(b)** Reconstructions

**Figure 3.12:** Results of applying a shallow autoencoder to MNIST data. (a) Some original test digits. (b) Corresponding reconstruction images by the autoencoder.

To provide a practical example, we applied a simple autoencoder to the MNIST dataset http://yann.lecun.com/exdb/mnist/. This is a collection of images of handwritten digits, of which some test samples are shown in Figure 3.12(a). We flattened each $28 \times 28$ training image to obtain a 784 dimensional input vector $\mathbf{x}_n$ consisting of values scaled to the $[0, 1]$ range. Then, we fitted a fully-connected autoencoder with one hidden layer of 50 units, ReLU activation for the hidden layer and sigmoid function for the output layer. In particular, the neural network is trained with stochastic gradient descent in order to minimize the binary cross-entropy loss $L(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\{\mathbf{x}_i \cdot \log \hat{\mathbf{x}}_i + (1 - \mathbf{x}_i) \cdot \log(1 - \hat{\mathbf{x}}_i)\}$. We used Adam with learning rate $\eta = 0.001$, batch size $B = 16$ and 20 epochs to obtain some reconstruction results showed in Figure 3.12(b).

We can also consider architectures with much more hidden neurons, however, we should keep in mind that the ultimate goal is not learning how to copy the input to the output exactly. Think again of a shallow autoencoder (with one hidden layer). If the number of hidden neurons $d_h$ is greater than or equal to the number of inputs $d_x$, the autoencoder easily learns the identity function $f_d(f_e(\mathbf{x})) = \mathbf{x}$ with zero reconstruction error, without discovering useful latent properties of the data. For this reason, we should somehow constrain the autoencoder not to learn the trivial identity function.

One way is to use a compressed representation with $d_h < d_x$ hidden neurons, reducing the dimensionality with a powerful nonlinear extension of PCA that learns the most salient features of the data. This is known as *undercomplete autoencoder*, and is the type we have considered so far in the example, characterized by the bottleneck of Figure 3.11. It can be extended to CNN autoencoders, where the encoder performs a convolution and the deconvolution, to form a hourglass structure that forces the network to learn a useful representation of the data.

### 3.3.2 Regularized autoencoders

Nevertheless, interesting features can also be learned using an *overcomplete* autoencoder with $d_h >> d_x$ on which some form of regularization is imposed. This leads to the so-called *regularized autoencoders*, of which we review the three best-known types: sparse autoencoders, denoising autoencoders and contractive autoencoders.

Sparse autoencoders (Ng, 2011) limit the reconstruction ability of the network by adding a sparsity penalty $\Omega(\mathbf{h})$ to the output $\mathbf{h}$ of the hidden unit activations. One possibility for this penalty may be the L1 norm, which encourages some units to be exactly equal to zero, but more typically a form of Kullback-Leibler divergence is used. Specifically, let $\overline{h}_j$ be the average output of the hidden unit $j$ over a minibatch, then the penalty is based on the divergence $\text{KL}(\rho||\overline{h}_j)$, where $\rho$ is a constant to which we want the $\overline{h}_j$ to be close (typically $\rho = 0.05$). For example, for two Bernoulli distributions with means $\rho$ and $\overline{h}_j$, we have $\text{KL}(\rho||\overline{h}_j) = \rho \log \frac{\rho}{\overline{h}_j} + (1-\rho) \log \frac{1-\rho}{1-\overline{h}_j}$.
Putting it all together, a sparse autoencoder is trained to minimize the loss

$$L_{SAE}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \sum_{j=1}^{d_h} \text{KL}(\rho||\overline{h}_j), \tag{3.34}$$

where, as usual, $\lambda$ is a regularization parameter. This training objective constrains the hidden units' values and makes the autoencoder discover interesting structure in the data.

Another type are denoising autoencoders (DAEs) (Vincent et al., 2008), which learn to reconstruct a clean version of an artificially corrupted input.

This is done by modifying the training objective as:

$$L_{DAE}(\boldsymbol{\theta}) = \mathbb{E}_{c(\tilde{\mathbf{x}}|\mathbf{x})}\big[L(\boldsymbol{\theta}; \mathbf{x}, \tilde{\mathbf{x}})\big], \tag{3.35}$$

where $\mathbb{E}_{c(\tilde{\mathbf{x}}|\mathbf{x})}[\cdot]$ averages over examples $\tilde{\mathbf{x}}$ drawn from a corruption process $c(\tilde{\mathbf{x}}|\mathbf{x})$. In practice, (3.35) is optimized from stochastic gradients obtained by adding some form of noise to the inputs. DAEs are especially successful at reconstructing corrupted images, since the training procedure teaches them to optimally undo the effect of corruption.

Finally, we have contractive autoencoders (CAEs) (Rifai et al., 2011), which aim to reduce the number of effective degrees of freedom of the representation by shrinking the derivative of the encoder around any input. Thus, the objective function becomes:

$$L_{CAE}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \left\| J(\mathbf{x}) \right\|_F^2, \tag{3.36}$$

where the penalty term is the Frobenius norm of $J(\mathbf{x}) = \frac{\partial f_e}{\partial \mathbf{x}}(\mathbf{x})$, the Jacobian of the encoder, which can be calculated analytically. In contrast to DAEs, this objective has the effect of penalizing the sensitivity of the learned features to small input variations, rather than the sensitivity of the reconstruction (Bengio et al., 2013). However, a possible issue is that the representation is encouraged to be robust only to infinitesimal input variations. This is solved by penalizing also higher order derivatives (Rifai et al., 2011).

These autoencoders are proven to be successful at learning to represent manifolds, low-dimensional regions where the data density concentrates that are embedded in the original higher-dimensional input space (see Bengio et al., 2013 for further details). More generally, in a first deep learning era, it was believed that the autoencoders we presented were also crucial in a supervised learning context. In particular, the modus operandi for deep neural nets was *greedy layerwise unsupervised pre-training*, which consisted of stacking the features learned by sequential autoencoders and then fine-tuning the model on the data with backprop. With the advent of large collections of labeled data, the unsupervised pre-training phase proved to be no longer necessary. Nevertheless, autoencoders remain important tools for applications such as semantic hashing, image processing and anomaly detection.

# Chapter 4

# Variational Autoencoders

In this chapter, we bring together all the work done previously to illustrate variational autoencoders (VAEs). These models were introduced independently by two research groups (Kingma & Welling, 2014; Rezende et al., 2014), and can be thought of as a probabilistic version of autoencoders. In fact, VAEs and standard autoencoders share similiarities in architectures, but they are built with a different statistical formulation and for different purposes. As an autoencoder extracts embeddings of the input space, a VAE is a generative model that not just reconstructs original instances, but also learns to generalize beyond the training examples already seen, producing realistic-looking synthetic data.

## 4.1 Probabilistic formulation

We start by describing the probabilistic framework. We consider the data $\mathbf{x}$ to come from the following generative process:

$$\mathbf{z} \sim p_{\boldsymbol{\vartheta}}(\mathbf{z}), \tag{4.1}$$

$$\mathbf{x} \,|\, \mathbf{z} \sim p\big(\mathbf{x} \,|\, f_{\boldsymbol{\vartheta}}(\mathbf{z})\big), \tag{4.2}$$

where $\mathbf{z}$ are latent variables and $f_{\boldsymbol{\vartheta}}(\mathbf{z})$ is a deep neural network. Therefore, the generative process is to first sample some values $\mathbf{z}$ from the prior distribution $p_{\boldsymbol{\vartheta}}(\mathbf{z})$ and then generate $\mathbf{x}$ from the conditional distribution $p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})$.

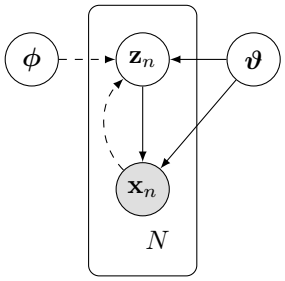



**Figure 4.1:** Directed acyclic graph representation of amortized inference in a VAE. Dashed lines indicate the variational approximation $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, solid lines denote the generative model $p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})\, p_{\boldsymbol{\vartheta}}(\mathbf{z})$. The variational parameters $\boldsymbol{\phi}$ are optimized jointly with the generative parameters $\boldsymbol{\vartheta}$.

The prior $p_{\boldsymbol{\vartheta}}(\mathbf{z})$ and the likelihood $p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})$ are assumed to be member of parametric families of distributions.

However, when we observe some data $\mathbf{x}$, we know neither the underlying latent variables $\mathbf{z}$ nor the true generative parameters $\boldsymbol{\vartheta}$. Thus, we are interested in performing posterior inference by computing the density $p_{\boldsymbol{\vartheta}}(\mathbf{z}|\mathbf{x})$, which, as usual, is intractable due to the marginal likelihood

$$p_{\boldsymbol{\vartheta}}(\mathbf{x}) = \int p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})\, p_{\boldsymbol{\vartheta}}(\mathbf{z})\, d\mathbf{z}. \tag{4.3}$$

For this reason, we need to resort to variational inference, also because we hypothesize to be in a machine learning context where dataset are too large for MCMC methods.

Specifically, VAEs are based on amortized inference, which was discussed in Subsection 2.3.3. As illustrated by Figure 4.1, this method amortizes the inference computations by using a shared variational approximation $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ to the posterior $p_{\boldsymbol{\vartheta}}(\mathbf{z}|\mathbf{x})$, so that the number of variational parameters does not grow with the size of the data. In particular, the variational approximation is called *recognition* (*inference*) family and has the form $q_{\boldsymbol{\phi}}\big(\mathbf{z}|f_{\boldsymbol{\phi}}(\mathbf{x})\big)$, where $f_{\boldsymbol{\phi}}(\mathbf{x})$ is chosen to be a neural network as well.

VAEs consists of combining the recognition family $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ and the generative model $p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})\, p_{\boldsymbol{\vartheta}}(\mathbf{z})$, with the two components that support each other during the joint optimization of the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\vartheta}$. The

recognition family provides the generative model with an approximation of its posterior; reversely, the second allows to learn useful representations of the data. To this extent, the recognition family can be viewed as approximately making the inverse path of the generative model.[1]

This idea was first proposed in the Helmholtz machine (Dayan et al., 1995), which used a wake-sleep algorithm to infer latent variables in a fashion similar to the EM algorithm. However, unlike VAEs, this method did not optimize a single objective function.

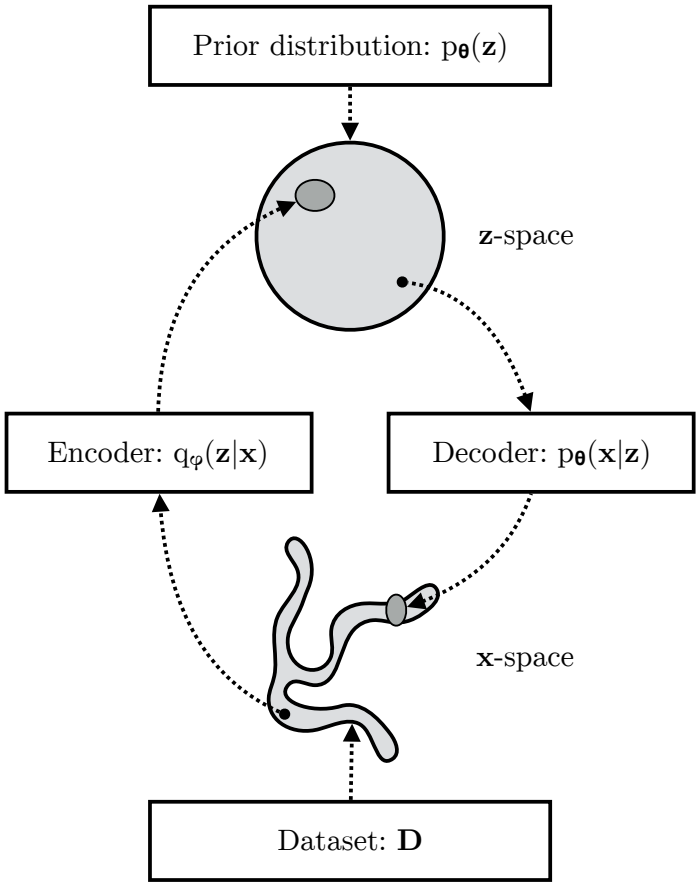



**Figure 4.2:** A VAE learns stochastic mappings between an observed **x**-space with complicated distribution, and a latent **z**-space with simple prior distribution (spherical in this case). Image taken from (Kingma & Welling, 2019).

[1]The approximation $q_{\phi}(\mathbf{z}|\mathbf{x})$ is often called "recognition model" in the literature, but we preferred to avoid this term to make precise the separation between inference and modeling from a statistical perspective.

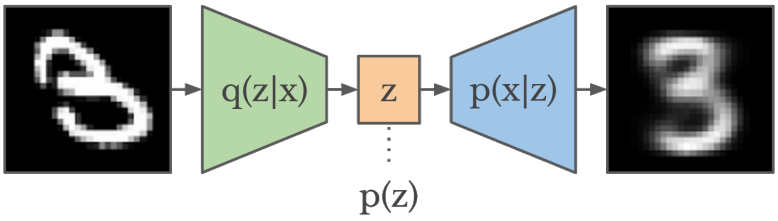



**Figure 4.3:** Schematic illustration of a VAE (Murphy, 2023).

The connection with autoencoders becomes clear when looking at Figures 4.2 and 4.3. A variational autoencoder is therefore composed of (Kingma & Welling, 2014):

(i) a probabilistic encoder $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ which, given an input $\mathbf{x}$, produces a distribution over all the possible values of the latent variables $\mathbf{z}$;

(ii) a probabilistic decoder $p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})$ which, given a $\mathbf{z}$ (with prior $p_{\boldsymbol{\vartheta}}(\mathbf{z})$), produces a distribution over the possible values of $\mathbf{x}$.

In its most common setting, the generative model $p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})\, p_{\boldsymbol{\vartheta}}(\mathbf{z})$ that composes a VAE is a deep latent Gaussian model (Rezende et al., 2014), with standard multivariate normal prior

$$p_{\boldsymbol{\vartheta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z} \,|\, \mathbf{0}, \mathbf{I}), \tag{4.4}$$

and Gaussian likelihood

$$p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z}) = \prod_{n=1}^{N} \mathcal{N}\big(\mathbf{x}_n \,|\, \boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2)\big). \tag{4.5}$$

The means and the variances in the likelihood derive from the outputs $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}$ of the decoding neural network $f_{\boldsymbol{\vartheta}}(\mathbf{z})$. An alternative for binary observations is to use the Bernoulli likelihood.

In a similar way, the recognition family is often defined to be:

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\big(\mathbf{z} \,|\, \boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2)\big), \tag{4.6}$$

where $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}$ are outputs of the encoding neural network $f_{\boldsymbol{\phi}}(\mathbf{x})$. Under this assumption, a VAE architecture is constructed as shown in Figure 4.4. We see that an input $\mathbf{x}$ is encoded into the means $\boldsymbol{\mu}$ and the standard deviations $\boldsymbol{\sigma}$ of a three-dimensional Gaussian approximation $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, from

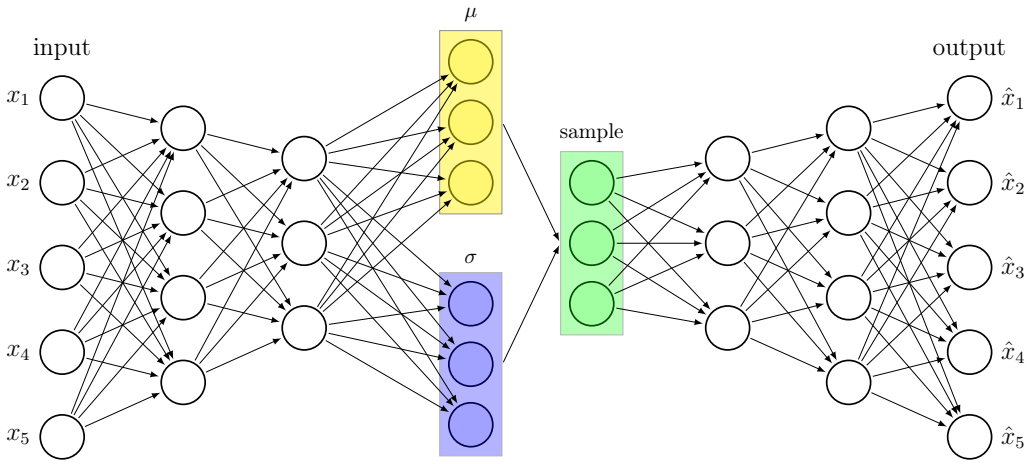



**Figure 4.4:** Illustration of a simple VAE that employs a Gaussian recognition family $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ with means $\boldsymbol{\mu}$ and standard deviations $\boldsymbol{\sigma}$.

which a vector $\mathbf{z}$ of latent variables is sampled. Subsequently, this sample $\mathbf{z}$ is decoded into the reconstruction $\hat{\mathbf{x}}$.

The image gives us an even better understanding of the difference between a VAE and a standard autoencoder: while the second embeds inputs into point estimates of latent variables, the first embeds inputs into a distribution of latent variables. This is the reason why VAEs are able to generate novel data instances by just decoding random samples from the prior $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

The training objective used is the evidence lower bound (ELBO) which, adapting the formulas of Section 1.1, is defined as:

$$\begin{aligned} \text{ELBO}(\boldsymbol{\vartheta}, \boldsymbol{\phi}; \mathbf{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})} \left[ \log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \right] && (4.7) \\ &= \log p_{\boldsymbol{\vartheta}}(\mathbf{x}) - \text{KL}\left( q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \,||\, p_{\boldsymbol{\vartheta}}(\mathbf{z}|\mathbf{x}) \right) && (4.8) \\ &\leq \log p_{\boldsymbol{\vartheta}}(\mathbf{x}). && (4.9) \end{aligned}$$

According to Equation (4.10), the maximization of the ELBO concurrently optimizes the two aspects we care about (Kingma & Welling, 2019): (i) it approximately maximizes the marginal likelihood $p_{\boldsymbol{\vartheta}}(\mathbf{x})$, improving the generative model; (ii) it minimizes the KL divergence between the recognition family $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ and the true posterior $p_{\boldsymbol{\vartheta}}(\mathbf{z}|\mathbf{x})$, improving the variational approximation. This is a crucial consideration since it allows to define a training

scheme for VAEs that jointly optimizes the variational parameters $\boldsymbol{\phi}$ and the generative parameters $\boldsymbol{\vartheta}$.

In addition, we can rewrite the ELBO as:

$$\begin{aligned}\text{ELBO}(\boldsymbol{\vartheta}, \boldsymbol{\phi}; \mathbf{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[-\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) + \log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z})] \\ &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[-\log\left(\frac{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\vartheta}}(\mathbf{z})}\, p_{\boldsymbol{\vartheta}}(\mathbf{z})\right) + \log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z})\right] \\ &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})\right] - \text{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \,||\, p_{\boldsymbol{\vartheta}}(\mathbf{z})\right).\end{aligned} \tag{4.10}$$

The last line of the objective function can be interpreted as the expected log likelihood plus a regularizer. The first term is the negative expectation of the reconstruction error, while the second term encourages variational densities close to the prior.

Therefore, unlike the autoencoders seen in Section 3.3, VAEs are trained to optimally decode random latent realizations into sensible outputs. The KL divergence from the prior encourages the latent space to be organized smoothly.

## 4.2 The reparameterization trick

We now discuss how to compute the ELBO and its gradient by means of the reparameterization trick introduced in Subsection 2.3.2. We have that unbiased gradients of the ELBO w.r.t. the generative parameters $\boldsymbol{\vartheta}$ are easily obtained as Monte Carlo estimates:

$$\begin{aligned}\nabla_{\boldsymbol{\vartheta}} \text{ELBO}(\mathbf{x}) &= \nabla_{\boldsymbol{\vartheta}}\, \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})] \\ &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\nabla_{\boldsymbol{\vartheta}}(\log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}))] \\ &\simeq \frac{1}{S}\sum_{s=1}^{S} \nabla_{\boldsymbol{\vartheta}} \log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}^s), \quad \text{where} \;\; \mathbf{z}^s \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}).\end{aligned} \tag{4.11}$$

Unbiased gradients of the ELBO w.r.t. the variational parameters $\boldsymbol{\phi}$ are more difficult to obtain since the expectation is taken w.r.t. a distribution indexed by $\boldsymbol{\phi}$. Namely, we have

$$\begin{aligned}\nabla_{\boldsymbol{\phi}} \text{ELBO}(\mathbf{x}) &= \nabla_{\boldsymbol{\phi}}\, \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})] \\ &\neq \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\nabla_{\boldsymbol{\phi}}(\log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}))].\end{aligned} \tag{4.12}$$

Assuming to have continuous latent variables $\mathbf{z}$, we can use a change of variables to compute ELBO's reparameterization gradients that, as discussed in Subsection 2.3.2, exhibit reduced variance. This reparameterization trick expresses the random variable $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ as a differentiable (deterministic) transformation $g$ of a noise variable $\boldsymbol{\epsilon}$, i.e.:

$$\mathbf{z} = g(\boldsymbol{\phi}, \mathbf{x}, \boldsymbol{\epsilon}), \quad \text{with} \quad \boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}), \tag{4.13}$$

where the auxiliary distribution $p(\boldsymbol{\epsilon})$ is independent of $\boldsymbol{\phi}$ and $\mathbf{x}$.

In this way, we can rewrite the gradients as expectations w.r.t. the noise distribution $p(\boldsymbol{\epsilon})$ and compute them through Monte Carlo estimates. Defining the function $f(\mathbf{z}) = \log p_{\boldsymbol{\vartheta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, we have:

$$\begin{aligned} \nabla \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] &= \nabla \mathbb{E}_{p(\boldsymbol{\epsilon})}[f(\mathbf{z})|_{\mathbf{z}=g(\boldsymbol{\phi},\mathbf{x},\boldsymbol{\epsilon})}] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[\nabla f(\mathbf{z})|_{\mathbf{z}=g(\boldsymbol{\phi},\mathbf{x},\boldsymbol{\epsilon})}] \\ &\simeq \frac{1}{S}\sum_{s=1}^{S} \nabla f(\mathbf{z}^s), \quad \text{where} \quad \mathbf{z}^s = g(\boldsymbol{\phi}, \mathbf{x}, \boldsymbol{\epsilon}^s), \quad \boldsymbol{\epsilon}^s \sim p(\boldsymbol{\epsilon}). \end{aligned} \tag{4.14}$$

This allows us to "externalize" the randomness in the latent variables and backpropagate the gradients through the $\mathbf{z}$ nodes, as is well explained by Figure 4.5.

Thus, we can use frameworks like `TensorFlow` and `PyTorch` to express a VAE as a computation graph, and automatically compute noisy but unbiased gradients of the ELBO through stochastic backpropagation. These noisy gradients are then passed to a stochastic optimization method like Adam: Algorithm 10 summarizes how the parameters are learned in a VAE. Since it involves sampling both noise values $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ and minibatches of data, this is a *doubly stochastic* algorithm.

In this vein, given a minibatch $\mathcal{M} = \{\mathbf{x}^{(b)}\}_{b=1}^{B}$ of $B$ data points randomly drawn from the full dataset with $N$ observations, and $S$ samples $\boldsymbol{\epsilon}^{(b,s)} \sim p(\boldsymbol{\epsilon}),\ s = 1, \dots, S,$ for every data point $\mathbf{x}^{(b)}$ in the minibatch, from Equation (4.10) we can construct the ELBO for a single data point as:

$$\begin{aligned} &\widetilde{\text{ELBO}}(\boldsymbol{\vartheta}, \boldsymbol{\phi}; \mathbf{x}^{(b)}) = -\text{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(b)}) \,||\, p_{\boldsymbol{\vartheta}}(\mathbf{z})\right) + \frac{1}{S}\sum_{s=1}^{S} \log p_{\boldsymbol{\vartheta}}(\mathbf{x}^{(b)}|\mathbf{z}^{(b,s)}), \\ &\text{where} \quad \mathbf{z}^{(b,s)} = g(\boldsymbol{\phi}, \mathbf{x}^{(b)}, \boldsymbol{\epsilon}^{(b,s)}). \end{aligned} \tag{4.15}$$

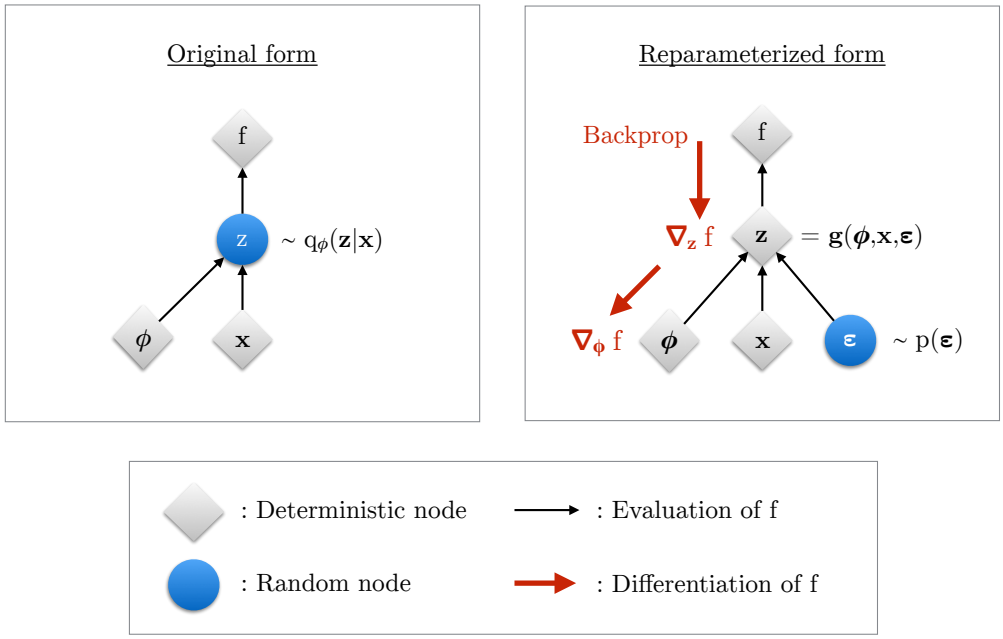



**Figure 4.5:** Illustration of how the reparameterization trick allows to compute gradients of the objective $f$ w.r.t to the variational parameters $\boldsymbol{\phi}$. On the left, the computation graph in its original form shows that the gradients cannot be directly backpropagated because it is not possible to differentiate w.r.t. the sampling operation $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$. As shown by the graph on the right, we can "move" the stochasticity into a noise source $\boldsymbol{\epsilon}$ by reparameterizing $\mathbf{z}$ with a deterministic and differentiable transformation $\mathbf{z} = g(\boldsymbol{\phi}, \mathbf{x}, \boldsymbol{\epsilon})$. This allows the gradients to flow backward through the $\mathbf{z}$ nodes (Kingma & Welling, 2019).

Equation (4.15) holds since the KL divergence can be often computed in closed form (e.g. for two Gaussian distributions, as we will see in an example). Things can be easily extended to whole minibatches, in a way that provides a stochastic approximation of the ELBO over the entire dataset, that is:

$$\mathrm{ELBO}(\boldsymbol{\vartheta}, \boldsymbol{\phi}; \mathbf{x}) \simeq \frac{N}{B} \sum_{b=1}^{B} \widetilde{\mathrm{ELBO}}(\boldsymbol{\vartheta}, \boldsymbol{\phi}; \mathbf{x}^{(b)}). \tag{4.16}$$

As usual, this quantity is used to monitor the training progress of the model.

**Algorithm 10:** Parameter estimation in a variational autoencoder

**Initialize:** variational parameters $\boldsymbol{\phi}$, generative parameters $\boldsymbol{\vartheta}$

**while** *not converged* **do**

Sample a minibatch $\mathcal{M} = \{\mathbf{x}^{(b)}\}_{b=1}^{B}$ of $B$ random data points

Sample $S$ noise vectors $\boldsymbol{\epsilon}^{(b,s)}$ for each data point in the minibatch

Compute $\mathbf{z}^{(b,s)} = g(\boldsymbol{\phi}, \mathbf{x}^{(b)}, \boldsymbol{\epsilon}^{(b,s)})$, for $b = 1, \ldots, B,\ s = 1, \ldots, S$

Compute the noisy gradients of the ELBO:

$$\hat{g}_{\boldsymbol{\vartheta}} = \frac{N}{B} \sum_{b=1}^{B} \Big\{ \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\vartheta}} \log p_{\boldsymbol{\vartheta}}(\mathbf{x}^{(b)}, \mathbf{z}^{(b,s)}) \Big\}$$

$$\hat{g}_{\boldsymbol{\phi}} = \frac{N}{B} \sum_{b=1}^{B} \Big\{ \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\phi}} \big( \log p_{\boldsymbol{\vartheta}}(\mathbf{x}^{(b)}, \mathbf{z}^{(b,s)}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}^{(b,s)} | \mathbf{x}^{(b)}) \big) \Big\}$$

Update $\boldsymbol{\vartheta}$ and $\boldsymbol{\phi}$ using stochastic gradient descent

**end**

Regarding the recognition family, the log density $\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ is computed by using the change of variables formula,

$$\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \log p(\boldsymbol{\epsilon}) - \log |\det(\partial \mathbf{z} / \partial \boldsymbol{\epsilon})|, \tag{4.17}$$

where $\partial \mathbf{z} / \partial \boldsymbol{\epsilon}$ is the Jacobian

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \begin{pmatrix} \frac{\partial z_1}{\partial \epsilon_1} & \cdots & \frac{\partial z_1}{\partial \epsilon_j} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_j}{\partial \epsilon_1} & \cdots & \frac{\partial z_j}{\partial \epsilon_j} \end{pmatrix}. \tag{4.18}$$

The most common choice for the probabilistic encoder is a factorized Gaussian:

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\big(\mathbf{z} \,|\, \boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2)\big) = \prod_j \mathcal{N}(z_j \,|\, \mu_j, \sigma_j^2). \tag{4.19}$$

In this case, the reparameterization has the form:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{4.20}$$

where $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}$ are outputs of the encoding neural network. We can compute the Jacobian of the transformation as

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \mathrm{diag}(\boldsymbol{\sigma}), \tag{4.21}$$

whose determinant is equal to the product of the diagonal terms $\sigma_j$.

The factorized Gaussian can be extended to a Gaussian with full covariance:

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma}). \tag{4.22}$$

Following (Kingma & Welling, 2019), a reparameterization is:

$$\mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon}, \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{4.23}$$

where $\mathbf{L}$ is a lower (or upper) triangular matrix with non-zero entries on the diagonal, which defines the Cholesky decomposition $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T$ of the covariance. The Jacobian of the transformation is $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \mathbf{L}$. Since $L$ is a triangular matrix, its determinant is the product of its main diagonal elements, thus

$$\log |\det(\partial \mathbf{z}/\partial \boldsymbol{\epsilon})| = \sum_j \log |L_{jj}|. \tag{4.24}$$

One way to construct a matrix $\mathbf{L}$ with the desired properties (triangularity and non-zero entries on the diagonal) is to define it as follows:

$$\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\boldsymbol{\sigma}), \quad (\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}') = f_{\boldsymbol{\phi}}(\mathbf{x}), \tag{4.25}$$

where $f_{\boldsymbol{\phi}}(\mathbf{x})$ is the encoding network and $\mathbf{M}$ is a masking matrix with zeros on and above the diagonal, and ones below the diagonal. With this construction, $\mathbf{L}$ is triangular with diagonal entries given by $\boldsymbol{\sigma}$, thus we have

$$\log |\det(\partial \mathbf{z}/\partial \boldsymbol{\epsilon})| = \sum_j \sigma_j. \tag{4.26}$$

More generally, we can replace $\boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon}$ with a chain of differentiable and nonlinear transformations: as long as the Jacobian of each step is triangular, the log determinant remains simple to compute. This is the basic idea behind *inverse autoregressive flows*, which allows to construct more flexible variational approximations that go beyond Gaussian posteriors. See (Kingma & Welling, 2019) for further details.

## 4.3 A simple VAE

To fix the ideas, we now propose a simple example of a VAE composed of a shallow encoder and a shallow decoder. Keep in mind Figure 4.4 during the construction.

Let the prior over the latent variables be the isotropic multivariate Gaussian $p_{\boldsymbol{\vartheta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z} \,|\, \mathbf{0}, \mathbf{I})$. We assume $p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z})$ to be the Bernoulli likelihood for the $D-$dimensional binary data $\mathbf{x}$. Thus, we have:

$$\begin{gathered} \log p_{\boldsymbol{\vartheta}}(\mathbf{x}|\mathbf{z}) = \sum_{d=1}^{D} x_d \cdot \log \hat{x}_d + (1 - x_d) \cdot \log(1 - \hat{x}_d), \\ \text{where } \hat{\mathbf{x}} = \text{sigmoid}(\mathbf{W}_5 \cdot \text{relu}(\mathbf{W}_4 \, \mathbf{z} + \mathbf{b}_4) + \mathbf{b}_5), \end{gathered} \tag{4.27}$$

with the generative parameters $\boldsymbol{\vartheta} = \{\mathbf{W}_4, \mathbf{b}_4, \mathbf{W}_5, \mathbf{b}_5\}$ that are the weights and the biases of the one-hidden-layer decoding neural network.

We assume the variational approximation $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ to be Gaussian with diagonal covariance, namely:

$$\begin{aligned} \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) = \log \mathcal{N}\big(\mathbf{z} \,|\, \boldsymbol{\mu}^{(i)}, \text{diag}(\boldsymbol{\sigma}^{2(i)})\big), \quad \text{where} \\ \boldsymbol{\mu}^{(i)} = \mathbf{W}_2 \, \mathbf{h} + \mathbf{b}_2, \\ \log \boldsymbol{\sigma}^{(i)} = \mathbf{W}_3 \, \mathbf{h} + \mathbf{b}_3, \\ \mathbf{h} = \text{relu}(\mathbf{W}_1 \, \mathbf{x}^{(i)} + \mathbf{b}_1), \end{aligned} \tag{4.28}$$

with $\mathbf{x}^{(i)}$ a random data point and $\boldsymbol{\phi} = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \mathbf{W}_3, \mathbf{b}_3\}$ the variational parameters, i.e. the weights and the biases of the one-hidden-layer encoder. As previosly explained, we sample from the encoder $\mathbf{z}^{(i,s)} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})$ using the reparameterization trick $\mathbf{z}^{(i,s)} = g(\boldsymbol{\phi}, \mathbf{x}^{(i)}, \boldsymbol{\epsilon}^{(i,s)}) = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(i,s)}$, where $\boldsymbol{\epsilon}^{(i,s)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

Since both the prior $p_{\boldsymbol{\vartheta}}(\mathbf{z})$ and the posterior approximation $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})$ are Gaussian, the KL divergence that defines the objective (4.15) can be integrated analytically. Let $J$ be the dimensionality of the latent space $\mathbf{z}$, and let $\mu_j^{(i)}$ and $\sigma_j^{(i)}$ respectively denote the $j$-th element of the mean and of the standard deviation at data point $i$. Then, taking from (Kingma & Welling, 2014), we have:

$$\begin{aligned} \int q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \log p(\mathbf{z}) \, d\mathbf{z} = -\frac{J}{2} \log 2\pi - \frac{1}{2} \sum_{j=1}^{J} \big((\mu_j^{(i)})^2 + (\sigma_j^{(i)})^2\big), \quad \text{and} \\ \int q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \, d\mathbf{z} = -\frac{J}{2} \log 2\pi - \frac{1}{2} \sum_{j=1}^{J} \big(1 + \log(\sigma_j^{(i)})^2\big). \end{aligned} \tag{4.29}$$

We can compute the KL divergence as:

$$
\begin{aligned}
-\mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(b)})\,||\,p_{\boldsymbol{\vartheta}}(\mathbf{z})\right) &= \int q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})(\log p(\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}))\, d\mathbf{z} \\
&= \frac{1}{2}\sum_{j=1}^{J}\left(1 + \log(\sigma_j^{(i)})^2 - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right).
\end{aligned}
\tag{4.30}
$$

Thus, the ELBO estimator for data point $i$ becomes equal to:

$$
\begin{aligned}
&\frac{1}{2}\sum_{j=1}^{J}\left(1 + \log(\sigma_j^{(i)})^2 - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right) + \frac{1}{S}\sum_{s=1}^{S}\log p_{\boldsymbol{\vartheta}}(\mathbf{x}^{(i)}|\mathbf{z}^{(i,s)}), \\
&\qquad \text{where} \quad \mathbf{z}^{(i,s)} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(i,s)}, \quad \boldsymbol{\epsilon}^{(i,s)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}).
\end{aligned}
\tag{4.31}
$$

### 4.3.1 Application to MNIST

We applied the VAE just presented to the MNIST dataset by using `PyTorch`. Most of the pixels in the scaled images are 0 or 1, so it is reasonable to assume approximately a Bernoulli distribution.

We employed 400 neurons in the hidden layer for both the encoder and the decoder, initially choosing a two-dimensional latent space. After initializing the parameters via Xavier initialization, we trained the VAE with Adam optimization (learning rate $\eta = 0.001$) for 20 epochs, using minibatches of size $B = 100$ and $S = 1$ noise samples per datapoint.

Figure 4.6 shows the manifold learned by the model, which is a projection of the high-dimensional data into the two-dimensional latent space. The latent space is organized smoothly: this allows us to generate novel sensible images and interpolate between two existing digits. Moreover, the kind of nonlinear probabilistic PCA carried out by the VAE the factors of variation in the data and learn meaningful representations.

Figure 4.7(a) shows some random digits generated by the model. With the same training setting, we also tried increasing the dimensionality of the latent space to 20 units, obtaining some generative results reported in 4.7(b). We observe that increasing the number of units in the latent space produces higher-quality digits, designed in an original style.

One usually use deeper generative models: all the considerations and the practicalities of the previous chapter apply.

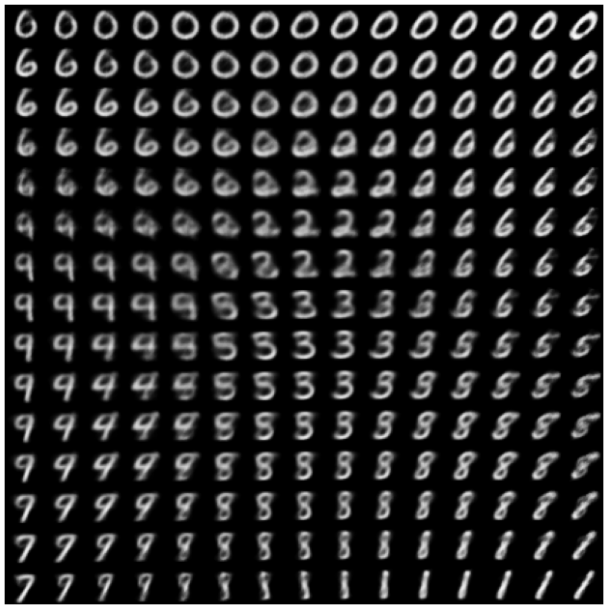
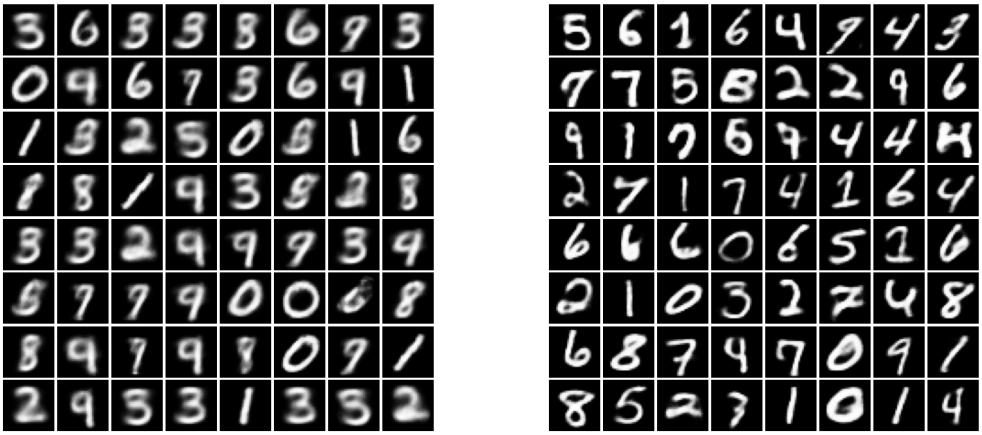


**Figure 4.6:** Visualization of the data manifold learned by a VAE with two-dimensional latent space. Since the prior of the latent space is Gaussian, we produced latent values $\mathbf{z}$ by passing an equispaced grid of coordinates on the unit square through the inverse CDF (cumulative distribution function) of a Gaussian. For each of these values $\mathbf{z}$, we plotted the corresponding image generated by $p_{\vartheta}(\mathbf{x}|\mathbf{z})$.



**(a)** 2-D latent space **(b)** 20-D latent space

**Figure 4.7:** Random samples generated by two VAEs trained on MNIST with different dimensionalities of latent space.

## 4.4 Convolutional VAE: application to celebrity faces

To conclude, we apply a convolutional VAE to the *CelebFaces Attributes* dataset https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html (Z. Liu et al., 2015). It is often abbreviated as "CelebA", and contains a variety of images of celebrity faces, as shown by some random samples in Figure 4.8.

According to (Murphy, 2023), we considered the following architecture. The encoder is formed by convolutional layers, which have the following progression in the number of hidden channels: $(32, 64, 128, 256, 512)$, and the following progression in terms of spatial size: $(32, 16, 8, 4, 2)$. Each convolutional layer is followed by a ReLU activation and by a batch normalization layer, there are not pooling layers.

The last $2 \times 2 \times 512$ convolutional layer is flattened and passed through a linear layer to compute the means and the diagonal variances from which to sample the latent vector $\mathbf{z}$, which has dimensionality 256. We therefore uses a factorized Gaussian as variational approximation. The structure of the decoder exactly mirrors that of the encoder, this time with deconvolutional layers. The Gaussian likelihood is employed within the ELBO computation.

That being said, we trained the convolutional VAE for 5 epochs with Adam optimization and batch size 256. We took advantage of the software implementation of (Murphy, 2023). Some celebrity faces generated by the model are shown in Figure 4.9: they appear a little blurry, but all in all realistic-looking.

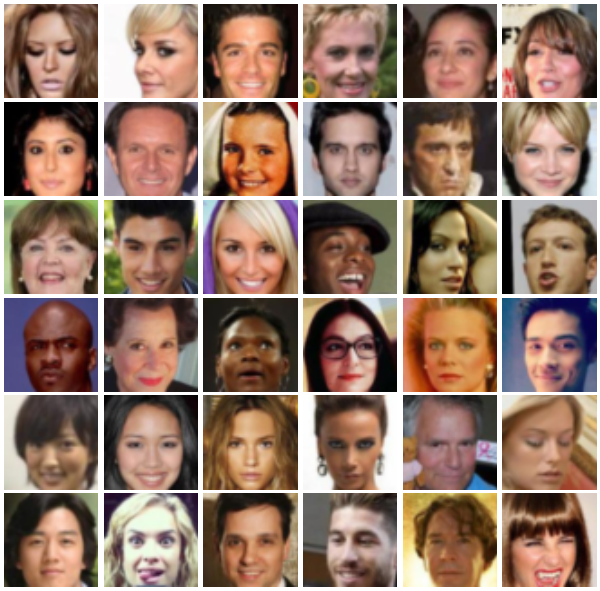


**Figure 4.8:** Random test images from the CelebA dataset.

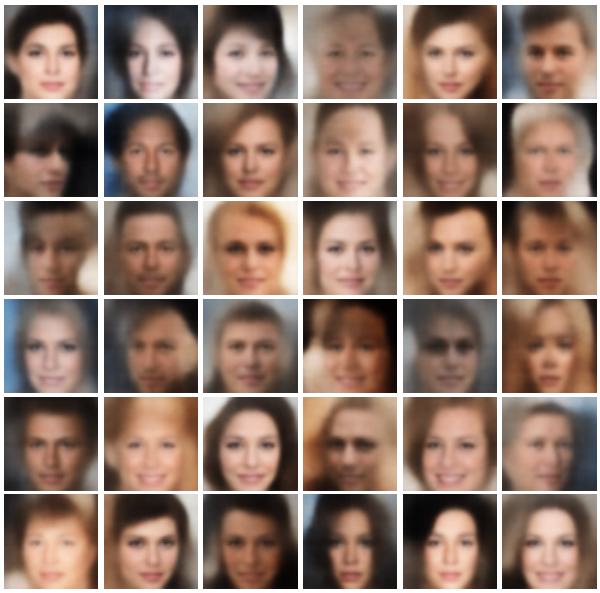


**Figure 4.9:** Random images generated by a convolutional VAE trained on CelebA.

# Conclusion

In this thesis, we focused on recent developments in variational inference. We saw how these methods can be married with ideas from deep learning to give life to the example of variational autoencoders, models that learn how to sample from complicated data distributions. Along the way, we provided parallels with other statistical methods.

After describing the foundations of variational Bayes, we showed how these techniques can be innovated in order to cope with large datasets and complex models. Stochastic optimization allows variational inference to scale up to massive data, while score and reparameterization gradients render VI applicable with little effort to a broad class of models.

Secondly, we discussed the aspects that characterize neural networks. In this context, we presented various types of autoencoders, which are used to extract useful features of data. Finally, we saw with practical applications how variational autoencoders are able to generate new sensible data. This is possible because a VAE is a probabilistic version of an autoencoder that embeds inputs into a distribution of latent variables.

We introduced only fairly basic types of variational autoencoders, however there is a boundless world behind these methods, which has more than 900 papers about them (Murphy, 2023).

# Appendix A

# Probability distributions

## Categorical

The categorical distribution is a discrete distribution that describes the possible results of a random variable $x$ that can take on one of $K$ possible categories, with the probability of each category separately specified. Denote $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)$, where $\pi_k$ is the probability of the $k$-th category, $k \in \{1, \ldots, K\}$ with $\sum_{k=1}^{K} \pi_k = 1$. Then we have that the probability mass function $f$ is

$$p(x = k \,|\, \boldsymbol{\pi}) = \pi_k, \quad k \in \{1, \ldots, K\}. \tag{A.1}$$

In this thesis, we adopt another formulation by considering the sample space to be the set of 1-of-$K$ encoded random vectors $\mathbf{x}$. This means $\mathbf{x}$ is $K$-dimensional and has exactly one element with the value 1, while the others have the value 0: the particular element having the value 1 indicates which category has been chosen. Thus, the probability mass function is given by

$$\mathrm{Cat}(\mathbf{x}|\boldsymbol{\pi}) = \prod_{k=1}^{K} \pi_k^{x_k}. \tag{A.2}$$

It holds

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\pi}. \tag{A.3}$$

This distribution is the most general one over a $K$-way event: any other

discrete distribution is a special case. The categorical is the generalization of the Bernoulli distribution for a categorical random variable, i.e. for a discrete variable with more than two possible outcomes. On the other hand, the categorical is a special case of the multinomial distribution, in that it gives the probabilities of potential outcomes of a single drawing rather than multiple drawings. The conjugate prior of the categorical distribution is the Dirichlet distribution.

## Dirichlet

The Dirichlet is a continuous multivariate distribution over $K$ random variables $0 \leq \pi_k \leq 1$, where $k = 1, \ldots, K,$ subject to the constraints

$$0 \leq \pi_k \leq 1, \quad \sum_{k=1}^{K} \pi_k = 1. \tag{A.4}$$

Denoting $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)$ and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)$,

$$\mathrm{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}) = C(\boldsymbol{\alpha}) \prod_{k=1}^{K} \pi_k^{\alpha_k - 1}, \tag{A.5}$$

where the normalization constant is

$$C(\boldsymbol{\alpha}) = \frac{\Gamma(\widehat{\alpha})}{\Gamma(\alpha_1) \cdots \Gamma(\alpha_K)}, \quad \widehat{\alpha} = \sum_{k=1}^{K} \alpha_k \tag{A.6}$$

$\Gamma(\cdot)$ is the Gamma function, such that

$$\Gamma(a) = \int_0^\infty x^{a-1} e^{-x} \, dx, \tag{A.7}$$

with its logarithmic derivative

$$\psi(a) \equiv \frac{d}{da} \log \Gamma(a) \tag{A.8}$$

known as the *digamma* function. The parameters $\alpha_k$ are subject to the constraint $\alpha_k \geq 0$ in order to ensure that the distribution can be normalized. Thus we have that

$$\mathbb{E}[\pi_k] = \frac{\alpha_k}{\widehat{\alpha}} \tag{A.9}$$

$$\mathbb{E}[\log \pi_k] = \psi(\alpha_k) - \psi(\widehat{\alpha}). \tag{A.10}$$

The entropy is given by

$$\mathbb{H}[\boldsymbol{\pi}] = -\sum_{k=1}^{K}(\alpha_k - 1)\left\{\psi(\alpha_k) - \psi(\widehat{\alpha})\right\} - \log C(\boldsymbol{\alpha}) \tag{A.11}$$

The Dirichlet forms the conjugate prior for the categorical and the multinomial distributions, representing a generalization of the Beta distribution. Suppose that $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ are i.i.d observations from a categorical distribution over $K$ classes, where for each $K$-dimensional vector $\mathbf{x}_n$, the element $x_{nk}$ is 1 if the observation $n$ belongs to class $k$ and 0 otherwise.

$$p(\mathbf{X}|\boldsymbol{\pi}) = \prod_{n=1}^{N} \mathrm{Cat}(\mathbf{x}_n|\boldsymbol{\pi}), \tag{A.12}$$

where $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)$ is the probability vector, which is assumed a priori to follow a Dirichlet distribution with hyperparameters $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)$

$$p(\boldsymbol{\pi}) = \mathrm{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}). \tag{A.13}$$

Then the posterior distribution of the parameters $\boldsymbol{\pi}$ is a Dirichlet

$$p(\boldsymbol{\pi}|\mathbf{X}) = \mathrm{Dir}\left(\boldsymbol{\pi} \,|\, \boldsymbol{\alpha} + (N_1, \ldots, N_K)\right), \tag{A.14}$$

where $N_k$ is the number of observations in category $k$

$$N_k = \sum_{n=1}^{N} x_{nk}. \tag{A.15}$$

Intuitively, the hyperparameters $\boldsymbol{\alpha}$ can be interpreted as pseudocounts, which represent the prior number of observations in each category.

## (Multivariate) Gaussian

The multivariate Gaussian is a continuous distribution over a $D$-dimensional random vector $\mathbf{x} \in \mathbb{R}^D$. The distribution is governed by a mean vector $\boldsymbol{\mu} \in \mathbb{R}^D$ and a covariance matrix $\boldsymbol{\Sigma} \in \mathbb{S}_{++}^{D\times D}$ which must be symmetric and positive definite.

$$\mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}\right) = \frac{1}{(2\pi)^{D/2}\,|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{ -\frac{1}{2}\left(\mathbf{x} - \boldsymbol{\mu}\right)^T \boldsymbol{\Sigma}^{-1}\left(\mathbf{x} - \boldsymbol{\mu}\right) \right\} \tag{A.16}$$

with

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}, \quad \text{Cov}(\mathbf{x}) = \boldsymbol{\Sigma}. \tag{A.17}$$

Considering a vector $\mathbf{c} \in \mathbb{R}^D$ and a symmetric matrix $\mathbf{A} \in \mathbb{S}^{D\times D}$, it holds the following result for the expectation of the quadratic form

$$\mathbb{E}\left[(\mathbf{c}-\mathbf{x})^T\mathbf{A}(\mathbf{c}-\mathbf{x})\right] = (\mathbf{c}-\boldsymbol{\mu})^T\mathbf{A}(\mathbf{c}-\boldsymbol{\mu}) + \text{Tr}\left(\mathbf{A}\boldsymbol{\Sigma}\right). \tag{A.18}$$

The inverse of the covariance matrix $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ is the precision matrix, which is also symmetric and positive definite. The conjugate prior for $\boldsymbol{\mu}$ is the Gaussian, the conjugate prior for $\boldsymbol{\Lambda}$ is the Wishart, and the conjugate prior for $(\boldsymbol{\mu}, \boldsymbol{\Lambda})$ is the Gaussian-Wishart.

## Gaussian-Wishart

The Gaussian-Wishart distribution comprises the product of a Gaussian distribution for $\boldsymbol{\mu}$, whose precision is proportional to $\boldsymbol{\Lambda}$, and a Wishart distribution over $\boldsymbol{\Lambda}$.

$$p(\boldsymbol{\mu}, \boldsymbol{\Lambda} \,|\, \boldsymbol{\mu}_0, \beta, \mathbf{W}, \nu) = \mathcal{N}\left(\boldsymbol{\mu} \,|\, \boldsymbol{\mu}_0, (\beta\boldsymbol{\Lambda})^{-1}\right) \mathcal{W}\left(\boldsymbol{\Lambda}|\mathbf{W}, \nu\right) \tag{A.19}$$

For the particular case of a scalar $x$, this is equivalent to the Gaussian-Gamma distribution.

It holds the following result for the expectation of the quadratic form

$$\mathbb{E}_{\boldsymbol{\mu},\boldsymbol{\Lambda}}\left[(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Lambda}(\mathbf{x}-\boldsymbol{\mu})\right] = D\,\beta^{-1} + \nu\,(\mathbf{x}-\boldsymbol{\mu}_0)^T\mathbf{W}(\mathbf{x}-\boldsymbol{\mu}_0), \tag{A.20}$$

which can be proved using (A.18), the cyclic property of the trace and the linearity of the trace operator.

The Gaussian-Wishart is the conjugate prior distribution for a multivariate Gaussian in which both the mean $\boldsymbol{\mu}$ and the precision $\boldsymbol{\Lambda}$ are unknown, and is also called the Normal-Wishart distribution. Consider the Gaussian-Wishart conjugate prior with hyperparameters $\boldsymbol{\mu}_0, \beta_0, \mathbf{W}_0, \nu_0$

$$p(\boldsymbol{\mu}, \boldsymbol{\Lambda}) = p(\boldsymbol{\mu}|\boldsymbol{\Lambda})\, p(\boldsymbol{\Lambda}) = \mathcal{N}\left(\boldsymbol{\mu} \,|\, \boldsymbol{\mu}_0, (\beta_0\boldsymbol{\Lambda})^{-1}\right) \mathcal{W}\left(\boldsymbol{\Lambda}|\mathbf{W}_0, \nu_0\right). \tag{A.21}$$

Suppose that $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ are i.i.d observations from a multivariate Gaussian distribution with mean $\boldsymbol{\mu} \in \mathbb{R}^D$ and precision matrix $\boldsymbol{\Lambda} \in \mathbb{S}_{++}^{D \times D}$. The likelihood is

$$p(\mathbf{X} \,|\, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = \prod_{n=1}^{N} \mathcal{N}\left(\mathbf{x}_n \,|\, \boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}\right). \tag{A.22}$$

Then the posterior distribution of the parameters $(\boldsymbol{\mu}, \boldsymbol{\Lambda})$ is a Gaussian-Wishart

$$p(\boldsymbol{\mu}, \boldsymbol{\Lambda} \,|\, \mathbf{X}) = \mathcal{N}\left(\boldsymbol{\mu} \,|\, \boldsymbol{\mu}_N, (\beta_N \boldsymbol{\Lambda})^{-1}\right) \mathcal{W}\left(\boldsymbol{\Lambda} | \mathbf{W}_N, \nu_N\right) \tag{A.23}$$

where

$$\begin{aligned}
&\beta_N = \beta_0 + N \\
&\boldsymbol{\mu}_N = \frac{1}{\beta_N}\left(\beta_0 \, \boldsymbol{\mu}_0 + N \overline{\mathbf{x}}\right) \\
&\mathbf{W}_N^{-1} = \mathbf{W}_0^{-1} + \sum_{i=1}^{N} \left(\mathbf{x}_i - \overline{\mathbf{x}}\right)\left(\mathbf{x}_i - \overline{\mathbf{x}}\right)^T + \frac{\beta_0 N}{\beta_0 + N}\left(\overline{\mathbf{x}} - \boldsymbol{\mu}_0\right)\left(\overline{\mathbf{x}} - \boldsymbol{\mu}_0\right)^T \\
&\nu_N = \nu_0 + N,
\end{aligned} \tag{A.24}$$

defined with $\overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}_n$ the sample mean.

Moreover, it is possible to define the posterior predictive distribution for a new observation $\widehat{\mathbf{x}}$ by means of

$$p(\widehat{\mathbf{x}} \,|\, \mathbf{X}) = \iint p(\widehat{\mathbf{x}} \,|\, \boldsymbol{\mu}, \boldsymbol{\Lambda}, \mathbf{X}) \, p(\boldsymbol{\mu}, \boldsymbol{\Lambda} \,|\, \mathbf{X}) \, d\boldsymbol{\mu} \, d\boldsymbol{\Lambda}. \tag{A.25}$$

Integrating out the variables $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ gives that the posterior predictive distribution is a multivariate Student's t with mean $\boldsymbol{\mu}_N$, precision $\mathbf{L}_N$ and $\nu_N + 1 - D$ degrees of freedom, i.e.

$$p(\widehat{\mathbf{x}} \,|\, \mathbf{X}) = \mathrm{St}\left(\widehat{\mathbf{x}} \,|\, \boldsymbol{\mu}_N, \mathbf{L}_N, \, \nu_N + 1 - D\right), \tag{A.26}$$

$$\text{where} \quad \mathbf{L}_N = \frac{(\nu_N + 1 - D)\beta_N}{(1 + \beta_N)} \, \mathbf{W}_N$$

## (Multivariate) Student's t

The Student's t-distribution for a $D$-dimensional random vector $\mathbf{x}$ corresponds to marginalizing out the precision matrix of a multivariate Gaussian with respect to a conjugate Wishart prior. It can therefore be viewed as an infinite mixture of (multivariate) Gaussians having the same mean but different covariance matrices, and takes the form

$$\mathrm{St}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}, \nu) = \frac{\Gamma(\nu/2 + D/2)}{\Gamma(\nu/2)} \frac{|\boldsymbol{\Lambda}|^{1/2}}{(\nu\pi)^{D/2}} \left[1 + \frac{\boldsymbol{\Delta}^2}{\nu}\right]^{-\nu/2 - D/2} \tag{A.27}$$

$$\begin{aligned} \mathbb{E}[\mathbf{x}] &= \boldsymbol{\mu} \quad \text{for } \nu > 1 \\ \mathrm{Cov}(\mathbf{x}) &= \frac{\nu}{\nu - 2} \boldsymbol{\Lambda}^{-1} \quad \text{for } \nu > 2 \end{aligned} \tag{A.28}$$

where $\nu > 0$ are the degrees of freedom of the distribution and $\boldsymbol{\Delta}^2$ is the squared Mahalanobis distance defined by $\boldsymbol{\Delta}^2 = (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Lambda} (\mathbf{x} - \boldsymbol{\mu})$.
In the limit $\nu \to \infty$, the t-distribution reduces to a Gaussian with mean $\boldsymbol{\mu}$ and precision $\boldsymbol{\Lambda}$. Student's t-distribution provides a generalization of the Gaussian whose maximum likelihood estimates are robust to outliers.

## Wishart

The Wishart distribution is the conjugate prior for the precision matrix of a multivariate Gaussian.

$$\mathcal{W}(\boldsymbol{\Lambda}|\mathbf{W}, \nu) = B(\mathbf{W}, \nu) \, |\boldsymbol{\Lambda}|^{(\nu - D - 1)/2} \exp\left(-\frac{1}{2} \mathrm{Tr}\left(\mathbf{W}^{-1} \boldsymbol{\Lambda}\right)\right), \tag{A.29}$$

The normalization constant is defined by

$$B(\mathbf{W}, \nu) \equiv |\mathbf{W}|^{-\nu/2} \left(2^{\nu D/2} \, \pi^{D(D-1)/4} \prod_{i=1}^{D} \Gamma\left(\frac{\nu + 1 - i}{2}\right)\right)^{-1} \tag{A.30}$$

Then we have that

$$\mathbb{E}[\boldsymbol{\Lambda}] = \nu \mathbf{W} \tag{A.31}$$

$$\mathbb{E}[\log|\boldsymbol{\Lambda}|] = \sum_{i=1}^{D} \psi\left(\frac{\nu + 1 - i}{2}\right) + D \log 2 + \log|\mathbf{W}| \tag{A.32}$$

and the entropy is given by

$$\mathbb{H}[\mathbf{\Lambda}] = -\log B(\mathbf{W}, \nu) - \frac{(\nu - D - 1)}{2}\mathbb{E}[\log|\mathbf{\Lambda}|] + \frac{\nu D}{2} \tag{A.33}$$

where $\mathbf{W}$ is a $D \times D$ symmetric, positive definite matrix, $\Gamma(\cdot)$ is the Gamma function and $\psi(\cdot)$ is the digamma function defined by (A.8). The parameter $\nu$ is called the number of degrees of freedom of the distribution and is restricted to $\nu \geq D - 1$ to ensure that the Gamma function in the normalization factor is well-defined. In one dimension, the Wishart reduces to the Gamma distribution with parameters $\alpha = \nu/2$ and $\beta = 1/2W$.

# Appendix B

# Python code

## B.1 CAVI algorithm for the BGMM

```
import numpy as np
import scipy
from scipy.special import psi, gammaln, multigammaln, logsumexp
import matplotlib.pyplot as plt
import timeit
import random
from sklearn import cluster



class CAVI_bgmm:


    def __init__(
        self, data, num_components,
        init_method = "random", seed = random.seed()
    ):


        self.data = data
        self.N = data.shape[0]
        self.D = data.shape[1]
        self.K = num_components
```

```
        self.clock_start = timeit.default_timer()

        # Priors hyperparameters

        self.alpha0 = 1/self.K
        self.beta0 = 1
        self.m0 = np.zeros(self.D)
        self.W0 = np.eye(self.D)
        self.nu0 = self.D

        # Variational parameters initialization

        if init_method == "random":
        # responsibilities are initialized uniformly at random
            resp = np.random.rand(self.N,self.K)
            resp /= resp.sum(axis=1)[:,np.newaxis]
            self.log_resp = np.log(resp)

        elif init_method == "kmeans":
        # responsibilities are initialized using K-means
            epsilon = 1e-6
            resp = np.zeros((self.N,self.K)) + epsilon
            label = (cluster.KMeans(n_clusters=self.K, n_init=1,
                random_state=seed).fit(self.data).labels_)
            resp[np.arange(self.N), label] = 1-epsilon*(self.K-1)
            self.log_resp = np.log(resp)

        self._compute_resp_statistics()

        # other parameters are initialized on the basis of
            responsibilities
        self.alpha = np.zeros(self.K)
        self.beta = np.zeros(self.K)
```

```
        self.m = np.zeros((self.K,self.D))
        self.W_chol = np.zeros((self.K,self.D,self.D))
        self.nu = np.zeros(self.K)
        self._update_alpha()
        self._update_beta()
        self._update_m()
        self._update_W()
        self._update_nu()


    def _compute_resp_statistics(self):
        """
        evaluate the statistics of dataset in formulae (1.45),(1.46) and
            (1.47),
        which depend on the responsibilities values
        """
        self.N_k = np.sum(np.exp(self.log_resp),0)

        self.xbar_k = np.dot(np.exp(self.log_resp).T, self.data) / self.
            N_k[:,np.newaxis]

        self.S_k = np.zeros((self.K,self.D,self.D))
        for k in range(self.K):
            diff = self.data - self.xbar_k[k]
            self.S_k[k] = np.dot(np.exp(self.log_resp[:,k])*diff.T, diff)
                 / self.N_k[k]


    def _update_alpha(self):
        # (1.52)
        self.alpha = self.alpha0 + self.N_k

    def _update_beta(self):
        # (1.54)
```

```
        self.beta = self.beta0 + self.N_k


    def _update_m(self):
        # (1.55)
        self.m = (self.beta0 * self.m0 + self.N_k[:, np.newaxis] * self.
            xbar_k) / self.beta[:, np.newaxis]


    def _update_W(self):
        # (1.56)
        for k in range(self.K):
            diff = self.xbar_k[k] - self.m0
            cov_chol = scipy.linalg.cholesky(
                scipy.linalg.inv(self.W0) + self.N_k[k] * self.S_k[k] +
                self.beta0 * self.N_k[k] / self.beta[k] * np.outer(diff,
                    diff),
                lower=True
            )
            # compute the Cholesky decomposition of precisions
            self.W_chol[k] = scipy.linalg.solve_triangular(
                cov_chol, np.eye(self.D), lower=True
            ).T



    def _update_nu(self):
        # (1.57)
        self.nu = self.nu0 + self.N_k



    def _squared_mahalanobis_chol(self, nu, data, means, precisions_chol)
        :
        """
        compute the squared Mahalanobis distance through the Cholesky
            decomposition
        and multiplies it by *nu*
```

```
        """
        qf = np.zeros((data.shape[0], means.shape[0]))
        for k in range(means.shape[0]):
            diff = np.dot(data, precisions_chol[k]) - np.dot(means[k],
                precisions_chol[k])
            qf[:,k] = nu[k] * np.sum(np.square(diff), axis=1)
        if qf.shape[0] == means.shape[0]: qf = np.diag(qf)
        return qf


    def _log_det_chol(self, precisions_chol):
        """
        compute the logarithmic determinant of the precision matrix
            through the
        Cholesky decomposition

        https://math.stackexchange.com/questions/3158303/using-cholesky-
            decomposition-to-compute-covariance-matrix-determinant
        """
        return [2 * np.sum(np.log(np.diag(precisions_chol[k]))) for k in
            range(np.shape(precisions_chol)[0])]


    def _log_pitilde(self):
        # (1.59), psi() is the digamma function (A.8)
        return psi(self.alpha) - psi(np.sum(self.alpha))

    def _log_Lambda(self):
        # (1.60)
        return (np.sum([psi(0.5*(self.nu-d)) for d in range(self.D)],
            axis=0
            ) + self.D*np.log(2.0) + self._log_det_chol(self.W_chol))
```

```
def _update_log_responsibilities(self):
    """
    update the logarithms of responsibilities, given the values of
        the other
    variational parameters

    we work with logarithms for numerical stability, since
        responsibilities
    are probabilities that may be very small in some cases
    """
    # (1.42)
    self.log_resp = (
        self._log_pitilde()
        + 0.5*self._log_Lambda()
        - 0.5*self.D*np.log(2*np.pi)
        - 0.5*(self.D/self.beta +
        self._squared_mahalanobis_chol(self.nu, self.data, self.m,
            self.W_chol))
        )
    # normalization (1.44) using the log-sum-exp trick for numerical
        stability
    # https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/
    self.log_resp -= logsumexp(self.log_resp, axis=1)[:, np.newaxis]



def _log_dirichlet_constnorm(self):
    # (A.6)
    return gammaln(np.sum(self.alpha)) - np.sum(gammaln(self.alpha))

def _log_wishart_constnorm(self):
    # (A.30)
    return -(0.5 * self.nu * self._log_det_chol(self.W_chol)
            + 0.5 * self.nu * self.D * np.log(2.0)
```

```
            + multigammaln(0.5*self.nu, self.D)
            )

    def _entropy_wishart(self):
        # (A.33)
        return -(self._log_wishart_constnorm()
            +0.5*(self.nu-self.D-1)*self._log_Lambda() - 0.5*self.nu*self
                .D
            )



    def compute_elbo(self):
        # compute the lower bound (1.64) ignoring constants
        return (self._elbo1() + self._elbo2() + self._elbo3() + self.
            _elbo4()
                - self._elbo5() - self._elbo6() - self._elbo7())

    def _elbo1(self):
        # (1.65)
        return 0.5 * np.sum(self.N_k * (
            self._log_Lambda()
            - self.D/self.beta
            - self.nu * [np.trace(np.dot(self.S_k[k],
                np.dot(self.W_chol[k], self.W_chol[k].T))) for k in range
                    (self.K)
            ]
            - self._squared_mahalanobis_chol(self.nu, self.xbar_k, self.m
                , self.W_chol)
            - self.D * np.log(2*np.pi))
            )

    def _elbo2(self):
        # (1.66)
        return np.sum(np.dot(np.exp(self.log_resp), self._log_pitilde()))
```

```
def _elbo3(self):
    # (1.67)
    return (self.alpha0-1) * np.sum(self._log_pitilde())

def _elbo4(self):
    # (1.68)
    return (0.5*np.sum(
        self._log_Lambda() - self.D*self.beta0/self.beta
        - self.beta0 * self._squared_mahalanobis_chol(self.nu, self.m
            , np.tile(self.m0,(self.K,1)), self.W_chol)
        + (self.nu0-self.D-1) * self._log_Lambda()
        - self.nu * [np.trace(np.dot(scipy.linalg.inv(self.W0),
        np.dot(self.W_chol[k], self.W_chol[k].T))) for k in range(
            self.K)
        ])
    )

def _elbo5(self):
    # (1.69)
    return np.sum(np.exp(self.log_resp) * self.log_resp)

def _elbo6(self):
    # (1.70)
    return np.sum((self.alpha-1) * self._log_pitilde()) + self.
        _log_dirichlet_constnorm()

def _elbo7(self):
    # (1.71)
    return np.sum(0.5*self._log_Lambda()
        + 0.5*self.D*np.log(self.beta)
        - self._entropy_wishart())
```

```
    def density_multivariate_t(self, x, mean, precision_chol, df):
        # compute the density of the multivariate t
        logdet = self._log_det_chol(precision_chol)
        qf = self._squared_mahalanobis_chol(1/df,x,mean,precision_chol)

        logdensity = [
            gammaln(0.5*(df[k]+self.D)) - gammaln(0.5*df[k]) - 0.5*self.D
                *np.log(df[k]*np.pi)
            + 0.5*logdet[k] - 0.5*(df[k]+self.D)*np.log(1+qf[:,k])
            for k in range(self.K)
            ]
        return logdensity

    def avg_log_predictive(self, test_data):
        # (1.73)
        logdensity = self.density_multivariate_t(
            x=test_data,
            mean=self.m,
            precision_chol=[np.sqrt((self.nu[k]+1-self.D)*self.beta[k
                ]/(1+self.beta[k])) *
                self.W_chol[k] for k in range(self.K)],
            df=self.nu+1-self.D
        ) # logsumexp trick
        # https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/
        ppd = (-np.log(np.sum(self.alpha)) +
            logsumexp(np.log(self.alpha)[:,np.newaxis] + logdensity, axis
                =0)
        )
        return np.mean(ppd)



    def fit(self, convergence_metric = "elbo", test_data=None, num_stop =
         3, max_iter = 500, tol = 1e-3):
```

```
        """
        fit the Bayesian Gaussian mixture model via CAVI and, in case of
        convergence, return one dictionary with final results and one
            with history
        """

        results = {}
        history = {
            "resp": [], "cluster": [], "weights": [], "means": [],
            "covariances": [], "metric": [], "clock": [],
            "init_clock": 0, "init_metric": 0
        }

        iter = 0
        stop = 0
        improvement = tol + 1

        if convergence_metric == "elbo":
            history["init_metric"] = self.compute_elbo()
        elif convergence_metric == "predictive density":
            history["init_metric"] = self.avg_log_predictive(test_data)
        history["init_clock"] = timeit.default_timer() - self.clock_start



        while True:

            # Variational E-step
            self._update_log_responsibilities()

            # Variational M-step
            self._compute_resp_statistics()
            self._update_alpha()
            self._update_beta()
            self._update_m()
```

```
            self._update_W()
            self._update_nu()



            history["resp"].append(np.exp(self.log_resp))
            history["cluster"].append(self.cluster_assignments(np.exp(
                self.log_resp)))
            history["weights"].append(self.posterior_mixture_weights())
            history["means"].append(self.m)
            history["covariances"].append(self.S_k)

            if convergence_metric == "elbo":
                history["metric"].append(self.compute_elbo())
                print(self.compute_elbo())
            elif convergence_metric == "predictive density":
                history["metric"].append(self.avg_log_predictive(
                    test_data))
                print(self.avg_log_predictive(test_data))

            history["clock"].append(timeit.default_timer() - self.
                clock_start)

            # convergence metric improvement
            if iter > 1:
                improvement = history["metric"][iter] - history["metric"
                    ][iter-1]

            # if the convergence metric decreases for three consecutive
                iterations, we declare convergence
            if improvement < 0:
                stop += 1
            else:
                stop = 0
```

```
            # Convergence criterion
            if abs(improvement) < tol or stop >= num_stop:

                cond = (stop >= num_stop) * num_stop

                print("Converged at iteration {}".format(iter - cond))
                print(convergence_metric, history["metric"][iter - cond])
                plt.xlabel("Iterations")
                plt.ylabel(convergence_metric)
                plt.plot(history["metric"])
                plt.show()

                results["resp"] = history["resp"][iter - cond]
                results["cluster"] = history["cluster"][iter - cond]
                results["weights"] = history["weights"][iter - cond]
                results["means"] = history["means"][iter - cond]
                results["covariances"] = history["covariances"][iter -
                    cond]

                break


            iter += 1
            if iter % 10 == 0: print("iter", iter)

            if iter > max_iter:
                print("Maximum iteration reached, not converged")
                break

        return results, history


    def cluster_assignments(self,resp):
```

```
        # assign each observation to the mixture component that maximizes
            the
        # responsibility for the specific observation
        return np.argmax(resp, axis=1)

    def posterior_mixture_weights(self):
        # (1.62)
        return (self.alpha0 + self.N_k) / (self.K * self.alpha0 + self.N)
```

### B.1.1 Fit a BGMM to the STL-10 dataset via CAVI

```
from __future__ import print_function

import sys
import os, sys, tarfile, errno
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import random
from tqdm import tqdm

if sys.version_info >= (3, 0, 0):
    import urllib.request as urllib
else:
    import urllib
try:
    from imageio import imsave
except:
    from scipy.misc import imsave

from cavi_bgmm import CAVI_bgmm
```

```
# image shape
HEIGHT = 96
WIDTH = 96
DEPTH = 3

# size of a single image in bytes
SIZE = HEIGHT * WIDTH * DEPTH

# path to the directory with the data
DATA_DIR = "./stl10data"

# url of the binary data
DATA_URL = "http://ai.stanford.edu/~acoates/stl10/stl10_binary.tar.gz"

# path to the binary train file with unlabeled image data
UNLAB_DATA_PATH = DATA_DIR + "/stl10_binary/unlabeled_X.bin"

def download_and_extract():
    """
    Download and extract the STL-10 dataset
    :return: None
    """
    dest_directory = DATA_DIR
    if not os.path.exists(dest_directory):
        os.makedirs(dest_directory)
    filename = DATA_URL.split("/")[-1]
    filepath = os.path.join(dest_directory, filename)
    if not os.path.exists(filepath):
        def _progress(count, block_size, total_size):
            sys.stdout.write("\rDownloading %s %.2f%%" % (filename,
                float(count * block_size) / float(total_size) * 100.0))
            sys.stdout.flush()
        filepath, _ = urllib.urlretrieve(DATA_URL, filepath, reporthook=
            _progress)
```

```
        print("Downloaded", filename)
        tarfile.open(filepath, "r:gz").extractall(dest_directory)

def read_single_image(image_file):
    """
    This method uses a file as input instead of the path - so the
    position of the reader will be remembered outside of context of this
        method.
    :param image_file: the open file containing the images
    :return: a single image
    """
    # read a single image, count determines the number of uint8"s to read
    image = np.fromfile(image_file, dtype=np.uint8, count=SIZE)
    # force into image matrix
    image = np.reshape(image, (3, 96, 96))
    # transpose to standard format
    # You might want to comment this line or reverse the shuffle
    # if you will use a learning algorithm like CNN, since they like
    # their channels separated.
    image = np.transpose(image, (2, 1, 0))
    return image

def read_all_images(path_to_data):
    """
    :param path_to_data: the file containing the binary images from the
        STL-10 dataset
    :return: an array containing all the images
    """

    with open(path_to_data, "rb") as f:
        # read whole file in uint8 chunks
        everything = np.fromfile(f, dtype=np.uint8)

        # We force the data into 3x96x96 chunks, since the
```

```
        # images are stored in "column-major order", meaning
        # that "the first 96*96 values are the red channel,
        # the next 96*96 are green, and the last are blue."
        # The -1 is since the size of the pictures depends
        # on the input file, and this way numpy determines
        # the size on its own.

        images = np.reshape(everything, (-1, 3, 96, 96))

        # Now transpose the images into a standard image format
        # readable by, for example, matplotlib.imshow
        # You might want to comment this line or reverse the shuffle
        # if you will use a learning algorithm like CNN, since they like
        # their channels separated.
        images = np.transpose(images, (0, 3, 2, 1))
        return images

def save_image(image, name):
    imsave("%s.jpg" % name, image, format="jpg")

def save_images(images):
    i = 0
    for image in tqdm(images, position=0):
        directory = DATA_DIR + "/" + "unlabeled_img" + "/"
        try:
            os.makedirs(directory, exist_ok=True)
        except OSError as exc:
            if exc.errno == errno.EEXIST:
                pass
        filename = directory + str(i)
        save_image(image, filename)
        i = i+1

def plot_image(image):
```

```
    """
    :param image: the image to be plotted in a 3-D matrix format
    :return: None
    """
    plt.imshow(image)
    plt.xticks(())
    plt.yticks(())
    plt.show()


def plot_image_color_histograms(image, bins, max_freq):
    """
    plot the RGB color histograms of a image
    :image: input image
    :bins: number of bins into which to divide the pixel intensities (the
         same for each color channel)
    :max_freq: maximum pixel counts (to adjust the plots)
    """

    colors = ("red", "green", "blue")
    channel_ids = (0, 1, 2)

    for channel_id, color in zip(channel_ids, colors):
        histogram, bin_edges = np.histogram(
            image[:, :, channel_id], bins=bins, range=(0, 256)
        )
        plt.subplot(3, 1, 1 + channel_id)
        plt.xlim([0, 256])
        plt.ylim([0, max_freq])
        plt.plot(bin_edges[0:-1], histogram, color=color)
        plt.xlabel("Pixel intensity")
        plt.ylabel("Pixel counts")

    plt.show()
```

```
def histogram_values(images, bins):
    """
    concatenate the pixel counts of the three RGB histograms, providing a
    3*bins-dimensional representation of each image

    :images: image dataset
    :bins: number of bins into which to divide the pixel intensities (the
         same for each color channel)
    """

    hist_values = np.zeros((len(images), 3*bins))
    channel_ids = (0, 1, 2)

    for i in range(len(images)):
        for ch in channel_ids:
            hist_values[i, range(ch*bins, (ch+1)*bins)], _ = np.histogram
                (
                images[i, :, :, ch], bins=bins, range=(0, 256)
            )

    return hist_values


def main():

    # download data if needed
    download_and_extract()

    images = read_all_images(UNLAB_DATA_PATH)
    # test to check if the whole dataset is read correctly
    # print(images.shape)

    # save images to disk
```

```
# save_images(images)


M = np.shape(images)[0]

N = 10000

bins = 64

num_cluster = 30


random.seed(770)


# Randomly select N training images

idx = random.sample(range(M), 2*N)

train_images = images[idx[:N]]

# test_images = images[idx[N:]]


# Plot a sample image and its color histograms

# plot_image(train_images[71])

# plot_image_color_histograms(train_images[71], bins, 1000)


###

# Obtain the data histograms to which to fit the Bayesian Gaussian
    mixture


train_hist_values = histogram_values(train_images, bins)

# test_hist_values = histogram_values(test_images, bins)


bgmm = CAVI_bgmm(train_hist_values, num_cluster, init_method="kmeans"
    ).fit()[0]


# Plot the nine most representative images from each of the mixture
    clusters


max_resp = np.amax(bgmm["resp"], axis=1)

df_estimates = pd.DataFrame({"Idx": range(N), "Maxresp": max_resp, "
    Cluster": bgmm["cluster"]})
```

```
    sorted_df = df_estimates.groupby(["Cluster"]).apply(lambda x: x.
        sort_values(["Maxresp"], ascending = False)).reset_index(drop=
        True)
    images_to_plot = sorted_df.groupby(["Cluster"]).head(9)

    for k in range(num_cluster):
        idx_cluster = images_to_plot.loc[images_to_plot["Cluster"] == k,
            "Idx"]
        for i in range(idx_cluster.shape[0]):
            splot = plt.subplot(3, 3, 1+i)
            plt.imshow(train_images[idx_cluster.iloc[i]])
            plt.xticks(())
            plt.yticks(())
        plt.show()



if __name__ == "__main__":
    main()
```

## B.2 Maximum likelihood EM algorithm for the GMM

```
"""
Not efficient, it's only to do a (small) comparison with the CAVI
    Bayesian mixture
"""

import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
```

```
class EM_gmm:

    def __init__(self, data, num_components):

        self.data = data
        self.N = data.shape[0]
        self.D = data.shape[1]
        self.K = num_components

        # parameters are randomly initialized
        self.mu_k = np.random.randn(self.K, self.D)
        A = np.random.rand(self.K, self.D, self.D)
        self.sigma_k = [np.dot(A[k], A[k].T) for k in range(self.K)]
        self.mixing_coef_k = np.repeat(1/self.K, self.K)

        self.N_k = np.zeros(self.K)
        self.resp = np.zeros((self.N, self.K))


    def _e_step(self):
        # Formula (1.24)
        for n in range(self.N):
            for k in range(self.K):
                self.resp[n,k] = (
                    self.mixing_coef_k[k] *
                    multivariate_normal.pdf(self.data[n], self.mu_k[k],
                        self.sigma_k[k])
                )
        # normalization
        self.resp /= self.resp.sum(axis = 1)[:,np.newaxis]


    def _m_step(self):
        # (1.28)
```

```
        self.N_k = np.sum(self.resp,0)
        # (1.27)
        self.mu_k = np.dot(self.resp.T, self.data) / self.N_k[:,np.
            newaxis]
        # (1.29)
        for k in range(self.K):
            diff = self.data - self.mu_k[k]
            self.sigma_k[k] = np.dot(self.resp[:,k]*diff.T, diff) / self.
                N_k[k]
        # (1.32)
        self.mixing_coef_k = self.N_k / self.N


    def log_likelihood(self):
        # (1.21)
        logL = 0
        for n in range(self.N):
            logL += np.log(np.sum(
                [self.mixing_coef_k[k] *
                multivariate_normal.pdf(self.data[n], self.mu_k[k], self.
                    sigma_k[k])
                for k in range(self.K)]
                ))
        return logL


    def fit(self, max_iter = 500, tol = 1e-3):
        """
        fit the Gaussian mixture model by means of the maximum likelihood
             EM
        algorithm and, in case of convergence, return a dictionary with
            final results
```

```
        in this case convergence is assessed by evaluating the log
            likelihood,
        also monitoring the parameters values works though
        """

        results = {}
        logL = []
        iter = 0

        while True:

            self._e_step()
            self._m_step()

            logL.append(self.log_likelihood())

            # log likelihood improvement
            improvement = logL[iter] - logL[iter-1] if iter > 0 else logL
                [iter]

            # Convergence criterion
            if iter > 0 and improvement < tol:

                print('Converged at iteration {}'.format(iter))
                print("log-likelihood", logL[-1])
                plt.plot(logL)
                plt.ylabel("log-likelihood")
                plt.xlabel("Iterations")
                plt.show()

                results["resp"] = self.resp
                results["cluster"] = self.cluster_assignments(self.resp)
                results["weights"] = self.mixing_coef_k
                results["means"] = self.mu_k
```

```
                results["covariances"] = self.sigma_k

                break


            iter += 1
            if iter % 10 == 0: print("iter", iter)

            if iter > max_iter:
                print('Maximum iteration reached, not converged')
                break

        return results


    def cluster_assignments(self,resp):
        # assign each observation to the mixture component that maximizes
            the
        # responsibility for the specific observation
        return np.argmax(resp, axis=1)
```

## B.3 Collapsed Gibbs sampler for the BGMM

```
import numpy as np
import random
from scipy.special import logsumexp
from scipy.stats import multivariate_t
import matplotlib.pyplot as plt
import timeit
from sklearn import cluster


class Gibbs_bgmm:
```

```
def __init__(
    self, data, num_components,
    init_method = "random", seed = random.seed()
):

    self.data = data
    self.N = data.shape[0]
    self.D = data.shape[1]
    self.K = num_components

    self.clock_start = timeit.default_timer()

    # Priors hyperparameters

    self.alpha0 = 1/self.K
    self.beta0 = 1
    self.m0 = np.zeros(self.D)
    self.W0 = np.eye(self.D)
    self.nu0 = self.D

    # Latent variables inizialization

    if init_method=="random":
    # observations are assigned randomly to one of the K components
        self.assignments = np.random.randint(0, self.K, self.N)

    elif init_method=="kmeans":
    # assignments are initialized using K-means
        self.assignments = (cluster.KMeans(n_clusters=self.K, n_init
            =1, random_state=seed).fit(self.data).labels_)

    # Parameters inizialization
```

```
        self.N_k = np.zeros(self.K)
        self.m_numerator = np.full((self.K, self.D), self.beta0*self.m0)
        self.Winv_tmp = np.full((self.K, self.D, self.D),
            np.linalg.inv(self.W0) + self.beta0*np.outer(self.m0, self.m0
                )
        )

        self.beta = np.full(self.K, self.beta0)
        self.m = np.zeros((self.K, self.D))
        self.Winv = np.zeros((self.K, self.D, self.D))
        self.nu = np.full(self.K, self.nu0)

        for k in range(self.K):
            for n in np.where(self.assignments == k)[0]:
                self._add_statistics_nk(n, k)


    def _add_statistics_nk(self, n, k):
        # add sufficient statistics to new cluster
        self.N_k[k] += 1
        self.m_numerator[k] += self.data[n]
        self.Winv_tmp[k] += np.outer(self.data[n], self.data[n])

        self.beta[k] += 1
        self.m[k] = self.m_numerator[k] / self.beta[k]
        self.Winv[k] = self.Winv_tmp[k] - self.beta[k] * np.outer(self.m[
            k], self.m[k])
        self.nu[k] += 1

        self.assignments[n] = k


    def _remove_statistics_n(self, n):
        # remove sufficient statistics from old cluster
```

```
        k = self.assignments[n]
        self.assignments[n] = -1

        self.N_k[k] -= 1
        self.m_numerator[k] -= self.data[n]
        self.Winv_tmp[k] -= np.outer(self.data[n], self.data[n])

        self.beta[k] -= 1
        self.m[k] = self.m_numerator[k] / self.beta[k]
        self.Winv[k] = self.Winv_tmp[k] - self.beta[k] * np.outer(self.m[
            k], self.m[k])
        self.nu[k] -= 1


    def _log_ppd_n(self, n):

        logdensity = [multivariate_t.logpdf(
                self.data[n],
                loc = self.m[k],
                shape = (1+self.beta[k]) / ((self.nu[k]+1-self.D)*self.
                    beta[k]) * self.Winv[k],
                df = self.nu[k]+1-self.D
            )
            for k in range(self.K)
        ]

        return logdensity


    def avg_log_predictive(self, test_data):
        # (1.73)
        logdensity = [multivariate_t.logpdf(
            test_data,
            loc=self.m[k],
```

```
            shape=(1+self.beta[k]) / ((self.nu[k]+1-self.D)*self.beta[k])
                 * self.Winv[k],
            df=self.nu[k]+1-self.D
            )
            for k in range(self.K)
        ]
        # https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/
        ppd = (-np.log(np.sum(self.alpha0 + self.N_k)) +
            logsumexp(np.log(self.alpha0 + self.N_k)[:,np.newaxis] +
                logdensity, axis=0)
        )
        return np.mean(ppd)


    def sampler(self, test_data, num_iter=200, burnin=50):

        history = { "clock": [], "avg_log_ppd": [], "init_clock": 0, "
            init_pred": 0 }

        history["init_pred"] = self.avg_log_predictive(test_data)
        history["init_clock"] = timeit.default_timer() - self.clock_start

        for iter in range(num_iter):

            obs_idx = list(range(self.N))
            random.shuffle(obs_idx)

            for n in obs_idx:

                self._remove_statistics_n(n)

                log_prob_zn = np.log(self.N_k + self.alpha0) + self.
                    _log_ppd_n(n)
```

```
                prob_zn = np.exp(log_prob_zn - logsumexp(log_prob_zn))  #
                     normalization

                k = np.random.choice(range(self.K), size=1, p=prob_zn) #
                    sample

                self._add_statistics_nk(n,k)

            history["avg_log_ppd"].append(self.avg_log_predictive(
                test_data))
            history["clock"].append(timeit.default_timer() - self.
                clock_start)

            print("iter", iter)


        return history
```

## B.4 Demo of a simple VAE

```
# Copyright: adapted from Arun Pandey, KU Leuven
# Based on work done for the course Data Mining and Neural Networks
# by Johan Suykens, KU Leuven

# Import necessary libraries for this course ----------------------
import torch
import torch.nn.functional as nn
import torch.autograd as autograd
import torch.optim as optim
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

```
import os
from tqdm import tqdm
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets, transforms

# Set hyper-parameters
mb_size = 100 # mini-batch size
Z_dim = 2  # latent-space dimension
h_dim = 400
c = 0
lr = 1e-3  # learning rate
max_epochs = 20



def mnist_dataloader(path_to_data='mnist'):
    """MNIST dataloader with (28, 28) images."""
    all_transforms = transforms.Compose([transforms.ToTensor()])
    train_data = datasets.MNIST(path_to_data, train=True, download=True,
        transform=all_transforms)
    train_loader = DataLoader(train_data, batch_size=mb_size, shuffle=
        True)
    return train_loader



def xavier_init(size):
    """Xavier inizialization"""
    in_dim = size[0]
    xavier_stddev = 1. / np.sqrt(in_dim / 2.)
    return Variable(torch.randn(*size) * xavier_stddev, requires_grad=
        True)

# Load Data
mnist = mnist_dataloader()
```

```
_, channels, x, y = next(iter(mnist))[0].size()
X_dim = channels * x * y

# Q(z|X)

Wxh = xavier_init(size=[X_dim, h_dim])
bxh = Variable(torch.zeros(h_dim), requires_grad=True)

Whz_mu = xavier_init(size=[h_dim, Z_dim])
bhz_mu = Variable(torch.zeros(Z_dim), requires_grad=True)

Whz_var = xavier_init(size=[h_dim, Z_dim])
bhz_var = Variable(torch.zeros(Z_dim), requires_grad=True)



def Q(X):
    h = nn.relu(X @ Wxh + bxh.repeat(X.size(0), 1))
    z_mu = h @ Whz_mu + bhz_mu.repeat(h.size(0), 1)
    z_var = h @ Whz_var + bhz_var.repeat(h.size(0), 1)
    return z_mu, z_var



def sample_z(mu, log_var):
    eps = Variable(torch.randn(mb_size, Z_dim))  # sample from unit
        gaussian
    return mu + torch.exp(log_var / 2) * eps  # re-parameterization trick



# P(X|z)

Wzh = xavier_init(size=[Z_dim, h_dim])
bzh = Variable(torch.zeros(h_dim), requires_grad=True)

Whx = xavier_init(size=[h_dim, X_dim])
```

```
bhx = Variable(torch.zeros(X_dim), requires_grad=True)


def P(z):
    h = nn.relu(z @ Wzh + bzh.repeat(z.size(0), 1))
    X = torch.sigmoid(h @ Whx + bhx.repeat(h.size(0), 1))
    return X



# TRAINING
params = [Wxh, bxh, Whz_mu, bhz_mu, Whz_var, bhz_var,
          Wzh, bzh, Whx, bhx]

solver = optim.Adam(params, lr=lr)  # Adam optimizer

for it in range(max_epochs):  # Epochs
    avg_loss = 0
    for _, (X, _) in enumerate(tqdm(mnist, desc="Iter-{}".format(it))):

        X = X.view(mb_size, -1)

        # Forward
        z_mu, z_var = Q(X)
        z = sample_z(z_mu, z_var)

        # Sampling from random z
        X_sample = P(z)

        # Loss
        # E[log P(X|z)]
        recon_loss = nn.binary_cross_entropy(X_sample, X, reduction='sum'
            ) / mb_size
        # https://github.com/y0ast/VAE-TensorFlow/issues/3
```

```
        # D_KL(Q(z|X) || P(z)); calculate in closed form as both dist.
            are Gaussian
        kl_loss = torch.mean(0.5 * torch.sum(torch.exp(z_var) + z_mu ** 2
             - 1. - z_var, 1))
        loss = recon_loss + kl_loss

        # Backward
        loss.backward()

        # Update
        solver.step()

        # Housekeeping
        for p in params:
            if p.grad is not None:
                data = p.grad.data
                p.grad = Variable(data.new().resize_as_(data).zero_())
        avg_loss += loss / len(mnist)

    print('Loss: {:.4}'.format(avg_loss.item()))

    # plot sometimes
    if it % 2 == 0:
        samples = P(z).data.numpy()[:64]

        plt.close()
        fig = plt.figure(figsize=(8, 8))
        gs = gridspec.GridSpec(8, 8)
        gs.update(wspace=0.05, hspace=0.05)

        for i, sample in enumerate(samples):
            ax = plt.subplot(gs[i])
            plt.axis('off')
            ax.set_xticklabels([])
```

```
            ax.set_yticklabels([])
            ax.set_aspect('equal')
            plt.imshow(sample.reshape(28, 28), cmap='Greys_r')

        if not os.path.exists('out/'):
            os.makedirs('out/')
        plt.savefig('out/{}_{}.png'.format(it, str(c).zfill(3)),
            bbox_inches='tight')
        c += 1

    # plot the manifold

    if Z_dim == 2 and it == max_epochs-1:

        nx = ny = 15
        x_values = np.linspace(.05, .95, nx)
        y_values = np.linspace(.05, .95, ny)

        canvas = np.empty((28*ny, 28*nx))
        for i, yi in enumerate(x_values):
            for j, xi in enumerate(y_values):
                z_mu = np.array([[norm.ppf(xi), norm.ppf(yi)]]).astype('
                    float32')
                x_mean = P(torch.from_numpy(z_mu))
                canvas[(nx-i-1)*28:(nx-i)*28, j*28:(j+1)*28] = x_mean.
                    data.numpy()[0].reshape(28, 28)

        plt.close()
        plt.figure(figsize=(8, 10))
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(canvas, origin="upper", cmap="gray")
```

```
        plt.savefig('out/manifold.png', bbox_inches='tight')


print("Done")
```

# References


Amari, S. (1998). Natural gradient works efficiently in learning. *Neural Computation*, *10*(2), 251–276.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, *18*, 1–43.

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, *35*(8), 1798–1828.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157–166.

Bernardo, J. M., & Smith, A. (2000). *Bayesian theory*. Chichester: John Wiley and Sons Ltd.

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Berlin, Heidelberg: Springer-Verlag.

Blei, D. M., Kucukelbir, A., & McAuliffe, J. D. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, *112*(518), 859–877.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, *3*, 993–1022.

Bottou, L. (1991). Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, *91*(8).

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., ... Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, *76*(1).

Casella, G., & Berger, R. L. (2002). *Statistical inference* (Vol. 2). Duxbury Pacific Grove, CA.

Casella, G., & Robert, C. P. (1996). Rao-blackwellisation of sampling schemes. *Biometrika*, *83*(1), 81–94.

Coates, A., Ng, A., & Lee, H. (2011). An analysis of single-layer networks in unsupervised feature learning. *Journal of Machine Learning Research - Proceedings Track*, *15*, 215-223.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303–314.

Damianou, A., & Lawrence, N. D. (2013). Deep gaussian processes. In *Artificial intelligence and statistics* (pp. 207–215).

Dayan, P., Hinton, G. E., Neal, R. M., & Zemel, R. S. (1995). The helmholtz machine. *Neural computation*, *7*(5), 889–904.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B*, *39*, 1-38.

Efron, B., & Hastie, T. (2016). *Computer age statistical inference: Algorithms, evidence, and data science*. Cambridge University Press.

Gal, Y., & Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning* (pp. 1050–1059).

Gelman, A., Carlin, J., Stern, H., Dunson, D., Vehtari, A., & Rubin, D. (2013). *Bayesian data analysis (3rd ed.)*. Chapman and Hall/CRC.

Geman, S., & Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*(6), 721-741.

Géron, A. (2017). Hands-on machine learning with scikit-learn and tensorflow: Concepts. *Tools, and Techniques to build intelligent systems*.

Gershman, S., & Goodman, N. (2014). Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society* (Vol. 36).

Ghahramani, Z., & Beal, M. (2001). Propagation algorithms for variational bayesian learning. *Adv. Neural Inform. Process. Syst*, *13*.

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 315–323).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Goodfellow, I., Bulatov, Y., Ibarz, J., Arnoud, S., & Shet, V. (2014). Multi-digit number recognition from street view imagery using deep convolutional neural networks. In *International conference on learning representations.*

Graves, A. (2011). Practical variational inference for neural networks. *Advances in neural information processing systems*, *24*.

Hinton, G. E., & Van Camp, D. (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on computational learning theory* (pp. 5–13).

Hoffman, M. D., Blei, D. M., Wang, C., & Paisley, J. (2013). Stochastic variational inference. *The Journal of Machine Learning Research*, *14*(1), 1303–1347.

Hoffman, M. D., & Gelman, A. (2014). The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *The Journal of Machine Learning Research*, *15*, 1593–1623.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448–456).

Jaakkola, T. S., & Jordan, M. I. (2000). Bayesian parameter estimation via variational methods. *Statistics and Computing*, *10*(1), 25–37.

Jordan, M. I. (1999). *Learning in graphical models*. MIT Press.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., & Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine learning*, *37*(2), 183–233.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint*.

Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. In *2nd international conference on learning representations, ICLR.*

Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, *12*(4), 307–392.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., & Blei, D. M. (2017). Automatic differentiation variational inference. *Journal of machine learning research*.

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, *22*(1), 79–86.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2011). Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, *54*(10), 95–103.

Liu, J. S. (1994). The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. *Journal of the American Statistical Association*, *89*(427), 958-966.

Liu, Z., Luo, P., Wang, X., & Tang, X. (2015, December). Deep learning face attributes in the wild. In *Proceedings of international conference on computer vision (iccv).*

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *in icml workshop on deep learning for audio, speech and language processing.*

MacKay, D. J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, *4*(3), 448–472.

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Proceedings of the seventeenth conference on uncertainty in artificial intelligence* (p. 362–369).

Montufar, G. F., Pascanu, R., Cho, K., & Bengio, Y. (2014). On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, *27*.

Murphy, K. P. (2007). *Conjugate bayesian analysis of the gaussian distribution* (Technical Report). Vancouver, CA: University of British Columbia.

Murphy, K. P. (2022). *Probabilistic machine learning: An introduction.* MIT Press.

Murphy, K. P. (2023). *Probabilistic machine learning: Advanced topics*. MIT Press.

Neal, R. (1996). *Bayesian learning for neural networks*. Berlin, Heidelberg: Springer-Verlag.

Neal, R., & Hinton, G. (2000). A view of the EM algorithm that justifies incremental, sparse, and other variants. *Learning in graphical models*, *89*.

Ng, A. (2011). Sparse autoencoder. *CS294A Lecture notes Stanford*, *72*(2011), 1–19.

Ong, V. M.-H., Nott, D. J., & Smith, M. S. (2018). Gaussian variational approximation with a factor covariance structure. *Journal of Computational and Graphical Statistics*, *27*(3), 465–478.

Ormerod, J. T., & Wand, M. P. (2010). Explaining variational approximations. *The American Statistician*, *64*(2), 140–153.

Parisi, G. (1988). *Statistical field theory*. Addison-Wesley.

Peterson, C. (1987). A mean field theory learning algorithm for neural networks. *Complex systems*, *1*, 995–1019.

Plataniotis, K., & Hatzinakos, D. (2000). Gaussian mixtures and their applications to signal processing..

Raftery, A. E., & Lewis, S. M. (1992). [practical markov chain monte carlo]: Comment: One long run with diagnostics: Implementation strategies for markov chain monte carlo. *Statistical Science*, *7*(4), 493 – 497.

Ranganath, R., Gerrish, S., & Blei, D. (2014). Black box variational inference. In *Artificial intelligence and statistics* (pp. 814–822).

Ranganath, R., Tang, L., Charlin, L., & Blei, D. (2015). Deep exponential families. In *Artificial intelligence and statistics* (pp. 762–771).

Ranganath, R., Wang, C., David, B., & Xing, E. (2013). An adaptive learning rate for stochastic variational inference. In *International conference on machine learning* (pp. 298–306).

Rezende, D. J., Mohamed, S., & Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning* (pp. 1278–1286).

Rifai, S., Mesnil, G., Vincent, P., Muller, X., Bengio, Y., Dauphin, Y., & Glorot, X. (2011). Higher order contractive auto-encoder. In *Joint european conference on machine learning and knowledge discovery in databases* (pp. 645–660).

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, *22*(3), 400–407.

Robert, C. P., & Casella, G. (2004). *Monte carlo statistical methods (springer texts in statistics)* (Second ed.). Springer-Verlag, New York.

Ross, S. M. (2022). *Simulation (6th ed.)*. Academic Press.

Ruiz, F. J. R., Titsias, M. K., & Blei, D. M. (2016). Overdispersed black-box variational inference. In *Uncertainty in artificial intelligence* (p. 647–656).

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Salvatier, J., Wiecki, T. V., & Fonnesbeck, C. (2016). Probabilistic programming in python using pymc3. *PeerJ Computer Science*, *2*, e55.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

Stephens, M. (2000). Dealing with label switching in mixture models. *Journal of the Royal Statistical Society: Series B*, *62*, 795–809.

Teh, Y. W., Jordan, M. I., Beal, M. J., & Blei, D. M. (2006). Hierarchical dirichlet processes. *Journal of the American Statistical Association*, *101*(476), 1566–1581.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

Titsias, M., & Lázaro-Gredilla, M. (2014). Doubly stochastic variational bayes for non-conjugate inference. In *International conference on machine learning* (pp. 1971–1979).

Titterington, D. M., & Wang, B. (2006). Convergence properties of a general algorithm for calculating variational bayesian estimates for a normal mixture model. *Bayesian Analysis*, *1*(3), 625–650.

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., & Blei, D. M. (2016). Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint*.

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on machine learning* (pp. 1096–1103).

Wainwright, M. J., Jordan, M. I., et al. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, *1*(1–2), 1–305.

Wang, C., & Blei, D. M. (2013). Variational inference in nonconjugate models. *Journal of Machine Learning Research*, *14*, 1005–1031.

Winn, J., & Bishop, C. M. (2005). Variational message passing. *Journal of Machine Learning Research*, *6*, 661–694.

Xu, M., Quiroz, M., Kohn, R., & Sisson, S. A. (2019). Variance reduction properties of the reparameterization trick. In *The 22nd international conference on artificial intelligence and statistics* (pp. 2711–2720).

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.

Zhang, C., Bütepage, J., Kjellström, H., & Mandt, S. (2018). Advances in variational inference. *IEEE transactions on pattern analysis and machine intelligence*, *41*(8), 2008–2026.