# Probabilistic Machine Learning
# The Variational Autoencoder

# Aprendizaje Automático Probabilístico
# El Autoencoder Variacional

Bachelor's Thesis

Author:
Ángel Ramos Ortiz

Supervisor:
Antonio Salmerón Cerdán

Bachelor's Degree in Mathematics



June, 2025
University of Almería

# Contents

# *Abstract*

This Bachelor's Thesis presents a theoretical and practical review of machine learning from a probabilistic perspective, with a special focus on the Bayesian approach and its integration with deep learning techniques. The central objective is a rigorous study of the Variational Autoencoder (VAE), a deep generative model that combines neural networks with approximate Bayesian inference to model complex data distributions.

The work begins within the framework of Bayesian statistics, highlighting the role of Bayes' Theorem in belief updating and uncertainty modeling. From there, variational inference is introduced as an efficient method for approximate inference in computationally demanding settings, reformulating the Bayesian problem as a functional optimization task. Key concepts such as the Evidence Lower Bound (ELBO), factorized variational families, and the Coordinate Ascent Variational Inference (CAVI) algorithm are analyzed in depth, with an emphasis on their theoretical foundations and connections to variational calculus.

The Variational Autoencoder is then presented as a deep latent variable model. Its formulation via variational inference and the reparameterization trick is detailed, demonstrating how the neural network is used to parameterize the posterior distribution. A notable contribution of this thesis is the presentation of an Approximation Theorem, which discusses the universal capabilities of these models in approximating complex probability distributions, supporting their role in generative tasks.

Finally, the practical implementation of the VAE is explored in several settings: data generation, denoising, anomaly detection, and latent space manipulation. These experiments showcase the representational power of the model and empirically validate its applicability, complementing the theoretical development with computational evidence.

Overall, this work highlights the integration of Bayesian statistics, variational methods, and deep neural networks, offering an applied and mathematically grounded perspective on modern generative learning.

# *Resumen*

Este Trabajo de Fin de Grado presenta una revisión teórica y práctica del aprendizaje automático desde una perspectiva probabilística, con especial énfasis en el enfoque bayesiano y su integración con técnicas de aprendizaje profundo. El objetivo central es el estudio riguroso del Autoencoder Variacional (VAE), un modelo generativo profundo que combina redes neuronales con inferencia estadística aproximada para modelar distribuciones complejas de datos.

Se parte del marco de la estadística bayesiana, donde se destaca el papel del Teorema de Bayes en la actualización de creencias y la modelización de incertidumbre. Con esa base, se introduce la inferencia variacional como método de inferencia aproximada eficiente en escenarios de alta complejidad computacional, transformando el problema bayesiano en uno de optimización funcional. Se analizan en detalle el *Evidence Lower Bound* (ELBO), las familias variacionales factorizadas y el algoritmo *Coordinate Ascent Variational Inference* (CAVI), profundizando en sus fundamentos teóricos y aplicando herramientas del cálculo variacional.

A continuación, se presenta el Autoencoder Variacional, enmarcado como un modelo profundo de variables latentes. Se detalla su formulación mediante inferencia variacional y el uso del truco de la reparametrización, mostrando cómo la red neuronal se utiliza para parametrizar la distribución a posteriori. Una contribución relevante del trabajo es el estudio de un Teorema de Aproximación, en el que se discuten las capacidades universales de estos modelos para aproximar distribuciones de probabilidad complejas, lo que justifica su uso en tareas generativas.

Finalmente, se realiza una implementación práctica del VAE en distintos escenarios: generación de datos, reducción de ruido, detección de anomalías y manipulación del espacio latente. Estos experimentos ilustran el poder representacional del modelo y validan empíricamente su aplicabilidad, complementando así el desarrollo teórico con evidencia computacional.

En su conjunto, este trabajo pone en valor la combinación entre estadística bayesiana, métodos variacionales y redes neuronales profundas, aportando una visión matemática y aplicada del aprendizaje generativo moderno.

# *Introduction*

In recent decades, machine learning and artificial intelligence have experienced exponential growth, driven by the massive availability of data and computational advances. However, many of the models currently used are perceived as statistical black boxes, that is, statistical models whose operation is unknown even to those who operate them.

This work is based on the idea that the best way to build machines that learn from data is through the use of probability theory. Probability allows us to face the inherent uncertainty in machine learning in its multiple forms: from the choice of the best model to the most reliable prediction or the most informed decision. In this sense, the Bayesian approach provides a coherent and mathematically solid framework so that model decisions are not only effective, but also interpretable and justifiable from a statistical point of view [15].

However, the direct application of Bayesian statistics in real problems is usually limited by the computational difficulty of obtaining posterior distributions in an exact way. This challenge has given rise to approximate inference methods, among which variational inference [9] occupies a prominent place due to its efficiency and scalability [3]. In turn, the rise of neural networks has led to the development of hybrid models that combine deep learning with probabilistic techniques, thus giving rise to a new type of generative models capable of learning complex latent representations from observed data.

One of the most representative models of this approach is the Variational Autoencoder (VAE), introduced in [11]. This model is based on the idea of learning a latent distribution that allows both reconstructing the input data and generating new samples, all framed in the context of variational inference. The VAE has proven to be a versatile tool, with applications ranging from image generation to anomaly detection, and its study allows for a deep understanding of the intersection between Bayesian statistics, optimization, and deep learning.

The main objective of this work is to explore in detail the framework of probabilistic machine learning, with special attention to the Variational Autoencoder. To this end, three specific objectives are proposed:

- To study the fundamentals of machine learning from a Bayesian perspective, with special emphasis on variational inference, developed in depth in Chapter 2.
- To analyze in depth the formulation and theoretical foundations of the Variational Autoencoder as a generative model of latent variables. For this, in Chapter 4, the theoretical development of the model is produced, presenting a probabilistic Approximation Theorem.
- To implement the model and apply it to different practical scenarios that allow observing its behavior and potential.

This approach aims not only to offer a rigorous theoretical understanding of the VAE, but also to validate its usefulness through computational experiments, using tools widely used in current deep learning research, such as the *PyTorch* library. In

Chapter 5, it is shown how these deep probabilistic models make it possible to efficiently address tasks that require a rich and flexible representation of data at a statistical level.

# 1 Preliminary Concepts

## 1.1 Bayesian Inference

Within the different schools of statistical thought, Bayesian statistics offers a coherent reasoning and formulation based on Bayes' Theorem. By making use of this result, a model is systematically updated as new data is provided. Bayesian statistics adopts a subjective view of probability that is understood as a degree of belief that changes as new evidence is observed using tools provided by probability theory. This contrasts with the frequentist interpretation, where probability is interpreted as the long-term relative frequency of the event under study. Consequently, the Bayesian interpretation becomes truly useful when modeling uncertainty in events that do not have observable or realizable long-term frequencies [15].

Let's imagine a scenario where there is a data set $\mathcal{D}$ and a model given by some parameters $\theta$. The expression provided in this case by Bayes' theorem includes the main elements of Bayesian inference:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}.$$

On the one hand, the prior density of the model $p(\theta)$ is the probability distribution that the parameters are initially believed to follow, before observing the data. This probability distribution is transformed thanks to Bayes' theorem into a distribution $p(\theta|\mathcal{D})$ known as the posterior, where the knowledge provided by the data observations is incorporated into the initial belief.

The likelihood function $p(\mathcal{D}|\theta)$ quantifies how well the model explains the data under different parameter configurations $\theta$. That is, it indicates how plausible the observed data are under given parametric values. The likelihood is not a probability distribution over $\theta$.

In the denominator, $p(\mathcal{D})$ acts as a normalizing term that makes the overall expression, when integrated over the entire parameter space, a probability distribution with respect to $\theta$. This term is called the evidence or marginal likelihood of the model, obtained as follows:

$$p(\mathcal{D}) = \int p(\mathcal{D}|\theta)p(\theta)d\theta.$$

One of the great advantages of the Bayesian approach is that the prior distribution $p(\theta)$ can reflect initial beliefs regarding the value of the parameters, which is a natural approach to a statistical problem [2].

**Definition 1.1.** *A class $\mathcal{P}$ of prior distributions for a parametric model $p(\mathcal{D},\theta)$, with $p(\theta) \in \mathcal{P}$ is said to be conjugate for $p(\mathcal{D}|\theta)$ if*

$$p(\theta|\mathcal{D}) \in \mathcal{P}.$$

The previous concept motivates some of the main criticisms of Bayesian statistics, since sometimes the prior distribution of a model is taken to be conjugate and thus facilitate the calculation of the posterior, sometimes abandoning a distribution that reflects a true initial belief about the parametric values in favor of gaining computational convenience.

In some situations it is possible to benefit from a conjugate expression and at the same time incorporate the initial belief in the prior taken. A classic example that serves to understand the difference between the Bayesian and frequentist approaches is the toss of a coin that is initially believed to be normal, but which is actually biased. In that case, we take:

$$p(x|\theta) \sim Bernoulli(\theta),$$

$$p(\theta) \sim Beta(100, 100).$$

The parameter $\theta \in (0, 1)$ is taken as the probability of showing heads. The initial belief reflects that the coin has the same probability of showing heads or tails, since the Beta distribution taken concentrates all its probability mass around 1/2 in a very accentuated way.

The Beta distribution is conjugate to the Binomial, so the posterior distribution will be another beta. This can be demonstrated very simply by using $p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta) \cdot p(\theta)$, arriving for a sample $\mathbf{x} = \{x^{(i)}\}_{i=1}^{N}$ at:

$$p(\theta|\mathbf{x}) \sim Beta(100 + \sum_{i=1}^{N} x^{(i)}, 100 + N - \sum_{i=1}^{N} x^{(i)}).$$

If the experiment is performed $N = 1000$ times and in 900 heads have been obtained, then, following the previous expression, we have that $p(\theta|\mathbf{x}) \sim Beta(1000, 200)$, whose probability mass is sharply concentrated in values very close to 0.9. Just as someone rational might think after the experiment that the coin is biased, the Bayesian approach given the evidence reflects that change in belief through a probability distribution that fits what happened.

The task of finding the posterior $p(\theta|\mathcal{D})$ is the procedure known as Bayesian inference. In the previous case, it was easy to perform the inference; this is known as exact inference. However, in more complex cases, the analytical approach will be practically unfeasible. This work is dedicated to exploring the methods of approximate inference within the variational framework. Later it will be seen that Bayesian inference is key to creating large deep models where there is room for interpretation, such as the Variational Autoencoder.

In certain part of the text, the MAP or *Maximum A Posteriori* estimator will be mentioned, which is based on taking the parametric estimation given by $\hat{\theta} = \underset{\theta \in \Theta}{\text{argmax}} p(\theta|\mathcal{D})$.

Bayesian inference is a very extensive topic in itself; one could talk about credible intervals, hypothesis contrasts focused on Bayes' theorem, and many more aspects. However, to understand this work it will be sufficient to understand the role of each

of the elements that intervene in an inference problem, already commented on previously.

## 1.2 Deep Learning

Deep learning is a branch of machine learning that is based on the use of neural networks, a type of function approximators that are especially flexible and scalable from a computational point of view. It is possible to define a neural network formally as follows:

**Definition 1.2.** *A neural network is a function $f_{\omega,b} : \mathbb{R}^n \to \mathbb{R}^m$ constructed by composing affine transformations and nonlinear functions–called activation functions– and that depends on two sets of parameters $w$ and $b$:*

$$f_{w,b}(\boldsymbol{x}) = (f^{[L]} \circ f^{[L-1]} \circ \dots f^{[1]})(\boldsymbol{x}), \qquad \forall \boldsymbol{x} \in \mathbb{R}^n,$$

*where each $f^{[l]} : \mathbb{R}^p \to \mathbb{R}^q$, for certain $p, q \in \mathbb{N}$ that change depending on $l$, is given by:*

$$f^{[l]}(\boldsymbol{a}) = \sigma^{[l]}(W^{[l]}\boldsymbol{a} + b^{[l]}), \qquad \forall \boldsymbol{a} \in \mathbb{R}^p,$$

*with $\sigma^{[l]}$ being a nonlinear function applied element by element, $W^{[l]} \in \mathbb{R}^{q \times p}$ the so-called weight matrix and $b^{[l]} \in \mathbb{R}^q$ the bias vector.*

In the statistical and deep learning community, the composition levels $l = 1, \dots, L$ are known as the layers of the neural network, with the value $L$ being the depth of the network. Following the notation of the previous definition, each of these layers is said to be composed of $p$ neurons, and it will be taken into account that the next layer is made up of $q$ neurons. Thus, the i-th neuron in layer $l+1$ has a weight vector $w_i^{[l]} \in \mathbb{R}^p$ and a bias $b_i^{[l]} \in \mathbb{R}$. When all the biases and weights of a layer are grouped together, we arrive at the matrix $W^{[l]} \in \mathbb{R}^{q \times p}$ and the vector $b^{[l]} \in \mathbb{R}^q$ given in the previous definition. These relationships between two levels or layers can be observed graphically in Figure 1.1.

The input data $\mathbf{x}$ in the neural network is normally assigned an input layer; this level will never be counted for the depth of the network and is said to be $l = 0$. In contrast, the final layer $L$ is the output of the neural network and its activation and number of neurons depend on the task for which the neural network is intended. These can range from classification, in which the output layer has as many units as classes and its activation produces a vector of probabilities, to regression, where a linear activation is used, which is often considered in the literature as not using any activation at all. The rest of the intermediate layers are called the hidden layers of the neural network.

It is in the hidden layers that the fascinating thing about neural networks really happens, which is that given a sufficiently large depth and layer size, they can result in certainly powerful approximators. The following result was the first to demonstrate the approximating power that these functions possess.

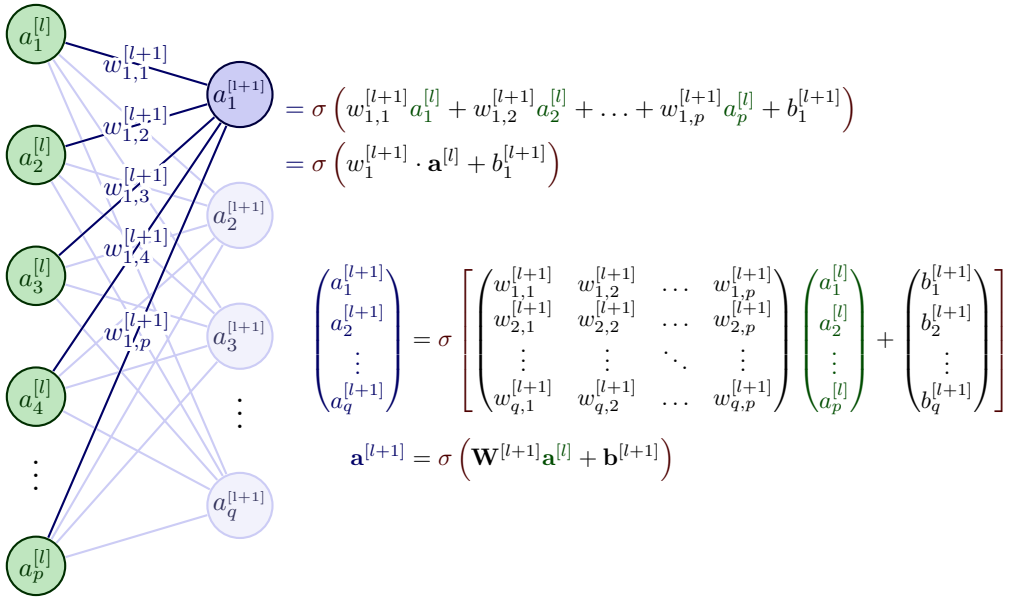



Figure 1.1: Diagram made with the *TikZ* package [22] of LaTeX which illustrates two densely connected neural network layers, with $p$ and $q$ units respectively. It shows how the activation of the first neuron of the second layer is calculated, as well as the complete activation of the layer, using matrix notation.

**Theorem 1.1.** *(Universal Approximation Theorem, [4]). Let $C([0,1]^n)$ be the set of continuous scalar functions defined in the unit hypercube of $\mathbb{R}^n$, if $\sigma$ is considered a sigmoid activation function, that is,*

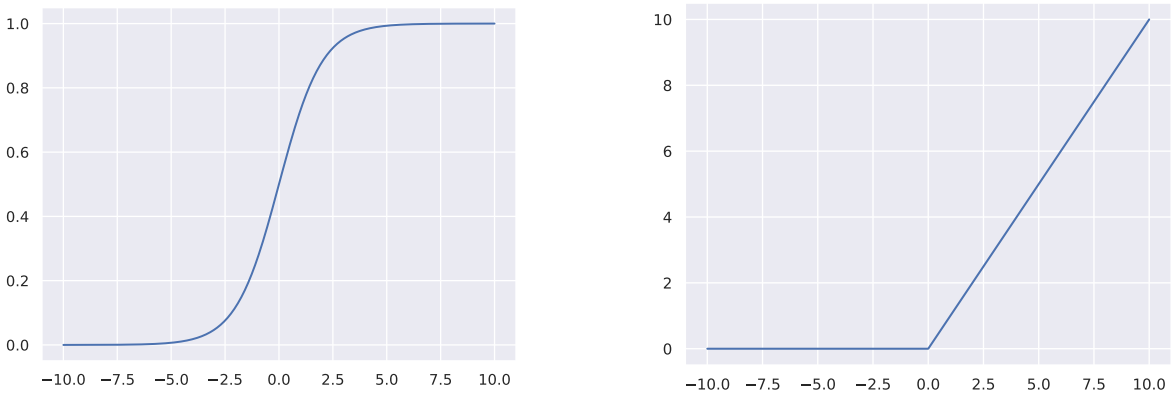
$$\sigma(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{x}}}, \qquad \forall \boldsymbol{x} \in \mathbb{R}^n,$$

*then the set $A = \{f \in C([0,1]^n) : f(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i \sigma(w_i^T \boldsymbol{x} + b_i)\}$ is dense in $C([0,1]^n)$.*

The density allows us to affirm that given any element of $C([0,1]^n)$, it will be possible to find an element of $A$ as close to it as desired, according to the infinite norm $\|\cdot\|_\infty$. This was the first universal approximation theorem, proposed in [4] and demonstrated using tools from Functional Analysis.

In simple terms, the theorem states that it is possible to approximate any real function defined in the hypercube using a single-layer neural network with enough neurons and a sigmoid activation. Later, different modifications of the result emerged that included the use of different activation functions and that had greater generality in terms of the functions to be approximated [8]. Today, the universal approximation theorem is often referenced to indicate that given any function there exists a sufficiently powerful neural network capable of approximating it as well as desired. Obviously, in practice, a level of complexity such that the approximation is perfect cannot always be reached due to computational limitations, but this result makes clear the great usefulness that neural networks can have.

The weights and biases of a neural network are the parameters that must be adjusted to achieve good performance in the problem it is applied to. To do this, op-




(a) (b)

Figure 1.2: The sigmoid activation functions $\sigma(x) = \frac{1}{1+e^{-x}}$ (a) and ReLU $f(x) = \max\{0, x\}$ (b) are two of the most common activation functions used in neural networks.

timization techniques based on the use of gradients are normally used. The neural network, given a data set $\mathcal{D}$, has a cost function $J$ that averages a loss function $L$ applied to each element of the sample; the mean squared error or cross-entropy, among others, is normally used. As will be seen when defining the *autoencoder*, which is a specific type of neural network, this loss function can often be accompanied by a regularization term that has various functions such as avoiding overfitting or forcing the network to perform a certain task.

A typical update of the parameter set given a learning rate $\alpha > 0$ is:

$$\omega \leftarrow w - \alpha \nabla_w J,$$
$$b \leftarrow b - \alpha \nabla_b J.$$

The iterative process of performing consecutive updates gives rise to the well-known gradient descent algorithm. On the other hand, backpropagation is an algorithm based on computational graphs and the chain rule of differential calculus that is vital when calculating the derivatives of the cost $J$ with respect to the parameters in a computationally efficient way and being able to perform the updates of the weights and biases. In chapters 6 and 8 of [6] you can read more about backpropagation and other optimization methods in neural networks.

## 1.3 Probabilistic Graphical Models

Probabilistic graphical models or Bayesian networks are a type of probabilistic model where the variables and their dependency relationships are organized and defined by a directed acyclic graph.

**Definition 1.3** ([18]). *A Bayesian network or directed probabilistic graphical model on a random vector $X = \{X_1, \ldots, X_n\}$ is a pair $(\mathcal{G}, \mathcal{E})$ where:*

- *$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed acyclic graph with $\mathcal{V} = \{X_1, \ldots, X_n\}$.*

- $\mathcal{P} = \{p(X_i|Pa(X_i)) : i \in \{1,\dots,n\}\}$, *where* $Pa(X_i)$ *denotes the set of parents of the variable* $X_i$ *with respect to the dependency relationships defined by* $\mathcal{E}$.

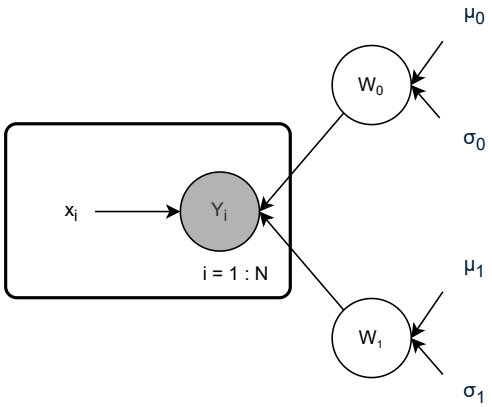



Figure 1.3: Example of a Bayesian network for a linear regression model where $Y_i|W_0, W_1, x_i = W_0 + W_1 x_i \quad \forall i \in \{1,\dots,N\}$ and we have that $W_j \sim \mathcal{N}(\mu_j, \sigma_j)$ with $j = 1, 2$. Figure created using *draw.io*.

In every Bayesian network parameterized by $\theta$, the joint distribution of the variables can be factored as a product of conditional and prior distributions:

$$p_\theta(x_1,\dots,x_n) = \prod_{j=1}^{n} p_\theta(x_j|Pa(x_j)).$$

Latent variables are those unobservable variables of the model that are not part of the data set. In what follows, when considering probabilistic models, $\mathbf{z}$ will denote the vector of latent variables. In this way, the model is defined by its joint distribution, normally $p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})$.

Probabilistic models of latent variables $p_\theta(\mathbf{x}, \mathbf{z})$ whose distributions are parameterized by the use of neural networks are called deep latent variable models. Deep latent variable models have the advantage that they can have a simple and well-known prior distribution or likelihood, such as the case of a normal, but can have quite complex marginal distributions $p_\theta(\mathbf{x})$ thanks to the expressiveness of the neural network involved. This makes these models ideal for approximating complicated underlying distributions, since the neural network can model very complex nonlinear transformations that are reflected in $\mathbf{z}$ by integrating into the latent space [12].

In the case of deep latent variable models, the expression of the evidence and the posterior distribution are normally analytically intractable and obtaining computationally efficient estimators can be complicated. Next, we will explore the variational inference methods that can obtain excellent results in this type of problem and given a large amount of data. Subsequently, the Variational Autoencoder will be introduced, a deep latent variable model that has been of great importance for generative and semi-supervised learning tasks.

# 2

# *Variational Inference*

Variational inference [9] is an approximate method of Bayesian inference that is fundamental when defining the *Variational Autoencoder* and, in general, learning processes that require large amounts of data. In this chapter, variational inference will be built step by step, from the motivations that lead to its appearance, to the typical procedure used in practice. Although this methodology differs from the one used in the VAE, it is important to understand the theoretical framework and the methods that are finally applied in this context.

One of the main problems of modern Bayesian statistics is the treatment of distributions (normally posteriors) that are difficult to compute. Faced with this challenge, two types of algorithms stand out for approximating intractable distributions: *Markov Chain Monte Carlo* (MCMC) and Variational Inference, each with its advantages and limitations.

Before discussing which method is more appropriate depending on the situation, consider the statistical model formed by latent or hidden variables $\mathbf{z} = (z_1, \dots, z_M)$ and sample observations $\mathbf{x} = \{x^{(i)}\}_{i=1}^N$. Their joint density is:

$$p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z}).$$

In this Bayesian model, the latent variables are obtained from a prior density $p(\mathbf{z})$, and are related to the observations through the likelihood of the model $p(\mathbf{x}|\mathbf{z})$. The goal of Bayesian inference is to find the posterior $p(\mathbf{z}|\mathbf{x})$, which in complex models requires the use of approximate inference techniques. To understand why, remember that the posterior distribution is normally calculated using Bayes' Theorem, so that:

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{z})p(\mathbf{x}|\mathbf{z})}{p(\mathbf{x})}.$$

The numerator, which is nothing more than the joint distribution of the latent variables and the data, is easy to obtain. The denominator, $p(\mathbf{x})$, is what is known as marginal density or evidence. In fact, it is obtained by marginalizing the joint density:

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

Here is the problem. For many models of practical interest, it is unfeasible to compute this expression, not to mention expectations with respect to it. This can be due to several factors, such as a too high dimensionality of the latent space or simply a very complex integrand that makes the expression analytically intractable. An alternative could be to resort to numerical integration. However, in scenarios of practical interest, the high dimensionality of the space and the complexity of the integrand could result in a prohibitive numerical approach, as will be seen in the example in Section 2.4.

Traditionally, MCMC has dominated the paradigm of approximate inference. In these methods, a Markov chain is constructed with respect to the latent variables, whose stationary distribution, normally reached in an asymptotic regime, is the true

posterior $p(\mathbf{z}|\mathbf{x})$. It is a random sampling method in which samples from the chain are collected and approximated with an empirical estimate of the results.

MCMC methods have made it possible to expand Bayesian statistics to other domains where they have proven to be valuable tools. Their great advantage is that, given infinite resources and time, exact results can be obtained. However, this same thing makes them computationally expensive, thus limiting their application to large-scale problems such as neural networks, learning models that require enormous amounts of data [3]. For example, in models like the VAE, variational inference is a good alternative to MCMC when performing approximate Bayesian inference.

Variational inference, instead of using sampling techniques, aims to solve the inference problem by transforming it into an optimization problem. Thus, a family of approximate densities $\mathcal{F}$ is proposed and an element is sought

$$q^*(\mathbf{z}) = \underset{q(\mathbf{z})\in\mathcal{F}}{\operatorname{argmin}} \quad KL(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x})).$$

That is, the density belonging to $\mathcal{F}$ that minimizes the Kullback-Leibler divergence operator is sought [1].

**Definition 2.1** ([15])**.** *The Kullback-Leibler divergence operator comes from Information Theory and measures how different two density functions $p(\boldsymbol{x})$ and $q(\boldsymbol{x})$ are on the same variable or random vector $\boldsymbol{X}$. It is given by the following expression:*

$$KL(q(\boldsymbol{x})\|p(\boldsymbol{x})) = \mathbb{E}_{q(\boldsymbol{z})}\left[\log\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})}\right] = \int q(\boldsymbol{x})\log\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})}d\boldsymbol{x}.$$

*It is responsible for measuring how much additional information is necessary if p is used to model data generated by q.*

The KL-divergence is always non-negative and is zero, and therefore minimum, if and only if $q = p$. The most intuitive way to conceive of this operator is as a distance between two probability distributions. However, caution should be exercised because, as can be seen from its definition, the operator is not symmetrical and is not a metric as such.

While it is true that in the inference problem it might seem logical to use the form $KL(p(\mathbf{z}|\mathbf{x})\|q(\mathbf{z}))$ by virtue of Definition 2.1, using $KL(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x}))$ is what is habitual because its expression allows the calculation to be performed in a much more direct way and to obtain equally good results, since the minimum in both cases is $q(\mathbf{z}) = p(\mathbf{z}|\mathbf{x})$.

In short, the inference problem has been transformed into an optimization problem. MCMC methods produce asymptotically exact samples in a guaranteed way, but variational inference, which can find a close distribution (which is normally not exact), tends to be faster in computational terms and is capable of dealing with large-scale data.

[1] It is common for the variational density to be denoted with dependence on the sample, i.e., $q(\mathbf{z}|\mathbf{x})$, because the update of the parameters is usually a function of the observations. However, in this section, it has been decided to omit this dependence to favor clarity in the calculations. In section 4, on the other hand, this dependence is made explicit due to its relevance in the analysis.

### *Why variational?*

It has been possible to know that the objective of variational inference is to optimize an operator over a space of probability densities, that is, over a space of functions. The latter is the main objective of variational calculus.

Variational calculus, unlike standard calculus, studies the infinitesimal variations introduced in functionals, that is, applications that take functions as input values. The KL-divergence defined above is a functional operator. Shannon entropy is also a functional operator, for example:

$$H(p) = -\int p(\mathbf{x}) \log p(\mathbf{x}) d\mathbf{x}.$$

The concept of a functional derivative arises from Euler's first ideas about the calculus of variations, indicating how the value of a functional changes with infinitesimal changes in the input function. In this case, it plays a fundamental role in how optimization techniques will be obtained in the paradigm of variational inference. Even so, once the family of functionals $\mathcal{F}$ is specified, it is possible to do so in such a way that the family is parametric and with it all its elements can be expressed as a function of parameters, thus managing to perform a search for a set of optimal parameters instead of functions [1].

## 2.1 The optimization problem and the ELBO

As previously mentioned, to perform variational inference, a family of densities $\mathcal{F}$ is postulated and the objective is to find:

$$q^*(\mathbf{z}) = \underset{q(\mathbf{z})\in\mathcal{F}}{\operatorname{argmin}} \quad KL(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x})).$$

The family $\mathcal{F}$ will determine the complexity of the optimization. Therefore, it must be flexible enough to obtain a good approximation of the posterior, but at the same time be computable in a reasonable amount of time.

Given $q \in \mathcal{F}$, the following development can be done:

$$\begin{aligned}
KL(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x})) &= \int q(\mathbf{z}) \log \frac{q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})} d\mathbf{z} \\
&= \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{z}|\mathbf{x})] \\
&= \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x},\mathbf{z})] + \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x})] \\
&= \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x}|\mathbf{z})] + \log p(\mathbf{x}).
\end{aligned}$$

From the above it is extracted that the expression of the KL-divergence depends on the evidence $p(\mathbf{x})$ and thus, in the case of interest, the operator becomes intractable.

Remember that this is the reason why variational inference is used in the first place. This does not mean that the initial approach is incorrect, optimizing the KL-divergence operator makes sense considering its definition, but it is not possible to compute it. The use of an alternative operator is thus proposed, which will be obtained by applying the so-called Jensen's inequality to the logarithm of the evidence.

**Theorem 2.1** (Jensen's Inequality [2]). *Let* $(\Omega, \mathcal{A}, P)$ *be a probability space,* $X$ *a real and integrable random variable, and* $f$ *a convex function. Then*

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)].$$

Applying Jensen's inequality, using the logarithm as a convex function $f$, a lower bound of the evidence is found, which receives the name of ELBO[2].

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int q(\mathbf{z}) \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})} d\mathbf{z} = \log\left(\mathbb{E}_{q(\mathbf{z})}\left[\frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})}\right]\right) \\ &\leq \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})}\right] = \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x},\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})].\end{aligned}$$

The ELBO can be defined as a functional $\mathcal{L}$ that depends on the variational density $q$:

$$\mathcal{L}(q) = \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x},\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})]. \tag{2.1}$$

By developing it, it is possible to appreciate why it fits perfectly in the optimization problem:

$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{z}|\mathbf{x})] + \log p(\mathbf{x}) - \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] \\ &= -KL(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x})) + \log p(\mathbf{x}).\end{aligned}$$

Remember that the KL-divergence is positive, so the previous equality shows that minimizing it is equivalent to maximizing the functional $\mathcal{L}$ [3]. In short, the optimization problem will be solved by maximizing the ELBO with the expression (2.1). Furthermore, from the ELBO, beyond the interpretation of the KL-divergence, it is possible to obtain intuitions about what is being done when solving this optimization problem. In fact, another way to develop its expression is the following:

$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x}|\mathbf{z})] + \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] \\ &= \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{x}|\mathbf{z})] - KL(q(\mathbf{z})||p(\mathbf{z})).\end{aligned}$$

Starting from the maximization of the ELBO, two conclusions are obtained from the right member of the previous expression: from the expected likelihood, it is deduced that the optimization will incentivize densities that concentrate their probability mass in configurations of $\mathbf{z}$ that explain the observations $\mathbf{x}$; the last term, on the other hand, indicates that the optimal density will be close to the prior distribution of the latent variables [3].

[2] An acronym that comes from *Evidence Lower Bound*.

[3] Note that $\log p(x)$ does not depend on the elements of the variational family with respect to which the optimization is carried out

## 2.2 Factorized Approximations

To completely define the optimization problem, it only remains to describe the variational family $\mathcal{F}$. The complexity of the optimization task will depend to a large extent on the complexity of the family.

A traditional approach is to use the *mean-field variational family* [3].

**Definition 2.2.** *The mean-field variational family $\mathcal{F}$ is a family of multivariate probability density functions where the marginal variables are independent of each other and each is governed by a different factor. Given a generic $q \in \mathcal{F}$:*

$$q(\mathbf{z}) = \prod_{j=1}^{M} q_j(z_j).$$

In this way, each variational factor $q_j$ is associated with $z_j$. When optimizing, the goal is to find each of these factors, which in turn can come from different distributions.

The mean-field variational family seems like a good candidate a priori since it seems to be expressive enough to capture the marginal distribution of each of the components of the latent variable vector $\mathbf{z}$. However, it has a major limitation: independence. At the same time that the assumption of independence between the variables will facilitate the optimization work, it fails to capture possible correlations between the latent variables.

## 2.3 CAVI

The quintessential algorithm used to perform approximate inference with the mean-field family is variational inference via coordinate ascent [3], an algorithm known as CAVI for its acronym in English.

The idea behind CAVI is to iteratively optimize each factor $q_k$ of the mean-field variational density until an optimum of the ELBO is reached, while the other factors remain fixed. In order to find the optimal expression of each variational factor, the following result is proposed where the maximum is found constructively and then it is proven that it is indeed the maximum. In the theorem, $q_{-k} = \prod_{j \neq k} q_j(z_j)$ will be denoted.

**Theorem 2.2** ([3])**.** *Let's consider a probabilistic model with an observation vector $\mathbf{x}$ and an initial configuration of its latent variables $\mathbf{z}$ on which we want to perform posterior variational inference using the factorized mean-field distribution $q(\mathbf{z}) = \prod_{j=1}^{M} q_j(z_j)$. Considering the variable $z_k$ and its complete conditional $p(z_k|z_{-k}, \mathbf{x})$, where all $q_j(z_j)$ with $j \neq k$ remain fixed, then the optimal update of the k-th variational factor is given by*

$$q_k^*(z_k) \propto \exp\{\mathbb{E}_{-k}[\log p(z_k|\mathbf{z}_{-k}, \mathbf{x})]\},$$

*where the previous expectation has been taken with respect to $q_{-k}$.*

Proof:

A probabilistic model and a variational density $q$ verifying the hypotheses of the statement are considered. Then it is possible to develop the entropy of $q$ in the following way:

$$H(q) = -\mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] = -\mathbb{E}_{q(\mathbf{z})}[\log \prod_{j=1}^{M} q_j(z_j)] = -\sum_{j=1}^{M} \mathbb{E}_j[\log q_j(z_j)].$$

In addition, the joint distribution of the observed variables and the latent variables can be expressed in a more convenient way using the chain rule

$$p(\mathbf{z}, \mathbf{x}) = p(\mathbf{x})p(\mathbf{z}|\mathbf{x}) = p(\mathbf{x}) \prod_{j=1}^{M} p(z_j|\mathbf{z}_{1:j-1}, \mathbf{x}).$$

Using these expressions, the ELBO is developed:

$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}_{q(\mathbf{z})}[\log p(\mathbf{z}, \mathbf{x})] - \mathbb{E}_{q(\mathbf{z})}[\log q(\mathbf{z})] \\ &= \log p(\mathbf{x}) + \sum_{j=1}^{M} (\mathbb{E}_{q(\mathbf{z})}[p(z_j|\mathbf{z}_{1:j-1}, \mathbf{x})] - \mathbb{E}_j[\log q_j(z_j)]).\end{aligned}$$

Without loss of generality, $z_k$ is taken as the last latent variable and the ELBO is considered as a function of the k-th variational factor, resulting in:

$$\mathcal{L}(q_k(z_k)) = \mathbb{E}_{q(\mathbf{z})}[p(z_k|\mathbf{z}_{-k}, \mathbf{x})] - \mathbb{E}_k[\log q_k(z_k)] + const,$$

where $const$ is a constant with respect to $q_k$. Thus,

$$\begin{aligned}\mathcal{L}(q_k(z_k)) &= \int q(\mathbf{z}) \log p(z_k|\mathbf{z}_{-k}, \mathbf{x}) d\mathbf{z} - \int q_k(z_k) \log q_k(z_k) dz_k + const \\ &= \int q_k(z_k) q_{-k}(\mathbf{z}_{-k}) \log p(z_k|\mathbf{z}_{-k}, \mathbf{x}) d\mathbf{z}_{-k} dz_k - \int q_k(z_k) \log q_k(z_k) dz_k + const \\ &= \int q_k(z_k) \mathbb{E}_{-k}[\log p(z_k|\mathbf{z}_{-k}, \mathbf{x})] dz_k - \int q_k(z_k) \log q_k(z_k) dz_k + const.\end{aligned}$$

Now, we derive with respect to the factor $q_k$ and set it to zero. This step involves the use of functional derivatives, where the derivative of a functional $F[g] = \int g(z) f(z) dz$ with respect to the function $g$ is $f(z)$:

$$\frac{\partial \mathcal{L}}{\partial q_k}(q_k^*) = \mathbb{E}_{-k}[\log p(z_k|\mathbf{z}_{-k}, \mathbf{x})] - \log q_k^*(z_k) - 1 = 0,$$

where by clearing and taking exponentials we arrive at

$$q_k^*(z_k) \propto \exp\{\mathbb{E}_{-k}[\log p(z_k|\mathbf{z}_{-k}, \mathbf{x})]\}.$$

It must now be shown that $q_k^*$ maximizes the ELBO using the previous development of it and the proportionality found.

$$\begin{aligned}\mathcal{L}(q_k(z_k)) &= \mathbb{E}_k[\mathbb{E}_{-k}[\log p(\mathbf{z},\mathbf{x})]] - \mathbb{E}_k[\log q_k(z_k)] + const \\ &= -\left(\mathbb{E}_k[\log q_k(z_k)] - \mathbb{E}_k\left[\log\left(\exp\left\{\mathbb{E}_{-k}\left[\log p(\mathbf{z},\mathbf{x})\right]\right\}\right)\right]\right) + const \\ &= -KL(q_k(z_k)||q_k^*(z_k)) + const.\end{aligned}$$

Note here that if we want to maximize $\mathcal{L}$ with respect to the variational factor $q_k(z_k)$, then this is equivalent to minimizing the expression $KL(q_k(z_k)||q_k^*(z_k))$ which is achieved when $q_k = q_k^*$.

■

As an immediate result, the following corollary is obtained.

**Corolary 2.1.** *Under the same conditions as Theorem 2.2, the optimal k-th variational factor can be expressed as*

$$q_k^*(z_k) \propto \exp\{\mathbb{E}_{-k}[\log p(\boldsymbol{x}, z_k, \mathbf{z}_{-k})]\} = \exp\{\mathbb{E}_{-k}[\log p(\boldsymbol{x}, \boldsymbol{z})]\}.$$

Note that in the expressions obtained, the expectations do not depend on $q_k$, but on the fixed variational factors. Therefore, updating $q_k$ using either of the two expressions is valid, in addition to guiding the ELBO to its optimum as has been shown.

Based on these results, the CAVI algorithm is presented:

**Algorithm 1** CAVI [3]

**Input:** A probabilistic model $p(\mathbf{x},\mathbf{z})$, a data set $\mathbf{x}$.
Define and initialize the variational factors $q_j(z_j)$.
**Repeat**
  **for** $j \in \{1,\dots,M\}$ **:**
    **take** $q_j(z_j) \propto \exp\{\mathbb{E}_{-j}[\log p(z_j|\mathbf{z}_{-j},\mathbf{x})]\}$
  **end**
  Compute ELBO.
**until:** *the ELBO converges*
**Output:** The variational density $q(\mathbf{z}) = \prod_{j=1}^{M} q_j(z_j)$.

As a curiosity, it is inevitable to note that CAVI is an algorithm that shares some similarity with Gibbs sampling, an approximate inference method of the MCMC family. In this method, the complete conditional of each variable $p(z_j|z_{-j},\mathbf{x})$ is also used, keeping the other variables fixed to obtain a Markov chain, which in an asymptotic regime will coincide with the desired distribution.

Finally, it is worth noting that the ELBO is generally a non-convex objective function, which means that CAVI can only guarantee convergence to a local optimum. Consequently, it is sensitive to initializations. This will be observed in more detail in the next practical example proposed in [3].

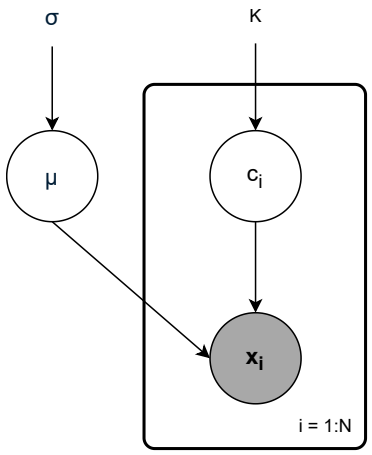



Figure 2.1: Two-dimensional mixture model. Figure created with *draw.io*.

## 2.4 Example with Gaussian Mixtures

Consider a bivariate Gaussian mixture of $K$ components. Each component is a normal distribution with independent means $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$, where $\boldsymbol{\mu}_k \in \mathbb{R}^2$ for each $k \in \{1, \ldots, K\}$. The common prior distribution for these means is an isotropic normal distribution centered at the origin and with a diagonal covariance matrix, that is, $p(\boldsymbol{\mu}_k) \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$ for all $k \in \{1, \ldots, K\}$, with the variance $\sigma^2 \in \mathbb{R}^+$ being a hyperparameter of the model.

To generate an observation $\mathbf{x}_i$ belonging to a random sample $\{\mathbf{x}_i\}_{i=1}^N$ of the model, the latent vector $\mathbf{c}_i$ is also considered, which for each $i \in \{1, \ldots, N\}$ is a categorical random variable (generalized Bernoulli) considered a priori equiprobable for each component. Using the variable $\mathbf{c}_i$, which will act as an indicator of which cluster the i-th sample belongs to, the mean $\boldsymbol{\mu}_{c_i}$ of that cluster is taken and the model's likelihood is defined as $\mathcal{N}(\boldsymbol{\mu}_{c_i}, \mathbf{I})$.

In summary, the model has the dependencies represented in Figure 2.1 and the likelihood and prior distributions for its latent variables are given by:

$$\begin{aligned} \boldsymbol{\mu}_k &\sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I}), \qquad k = 1, \ldots, K, \\ \mathbf{c}_i &\sim Cat\left(\frac{1}{K}, \ldots, \frac{1}{K}\right), \qquad i = 1, \ldots, N, \\ \mathbf{x}_i | \mathbf{c}_i, \boldsymbol{\mu} &\sim \mathcal{N}(\boldsymbol{\mu}_{c_i}, \mathbf{I}), \qquad i = 1, \ldots, N. \end{aligned}$$

**Note:** In the definition of the model, the standard subscript and superscript notation applied throughout the document has been broken. This has been done in favor of greater clarity for the reader.

Returning to the model, its joint distribution can be developed as follows:

$$p(\boldsymbol{\mu}, \mathbf{c}, \mathbf{x}) = p(\boldsymbol{\mu}, \mathbf{c}) p(\mathbf{x} | \boldsymbol{\mu}, \mathbf{c}) = p(\boldsymbol{\mu}) p(\mathbf{c}) p(\mathbf{x} | \boldsymbol{\mu}, \mathbf{c}).$$

Then the evidence is:

$$p(\mathbf{x}) = \sum_{\mathbf{c}} \int p(\boldsymbol{\mu}) p(\mathbf{c}) p(\mathbf{x}|\boldsymbol{\mu}, \mathbf{c}) d\boldsymbol{\mu} = \sum_{c} p(\mathbf{c}) \int p(\boldsymbol{\mu}) \prod_{i=1}^{N} p(\mathbf{x}_i|\boldsymbol{\mu}, \mathbf{c}_i) d\boldsymbol{\mu}.$$

In the previous integral, each $\mu_k \in \mathbb{R}^2$ appears in all the factors of the product, so the integral is performed over a space of dimension $2K$. In addition, it is summed over the $K^N$ possible configurations of the set of vectors $\mathbf{c}$. Therefore, the time complexity of evaluating this expression is at least $\mathcal{O}(K^N)$, and if the integral cannot be calculated analytically, its numerical evaluation would have an additional exponential cost in $K$, being completely ruled out for large values of $N$ and $K$.

With the goal of performing variational inference, the following candidate from the mean-field variational family is proposed which seems appropriate for this scenario:

$$q(\boldsymbol{\mu}, \mathbf{c}) = \prod_{k=1}^{K} q(\boldsymbol{\mu}_k; \mathbf{m}_k, s_k^2) \prod_{i=1}^{N} q(\mathbf{c}_i; \boldsymbol{\varphi}_i),$$

where,

$$\begin{aligned} q(\boldsymbol{\mu}_k; \mathbf{m}_k, s_k^2) &\sim \mathcal{N}(\mathbf{m}_k, s_k^2 \mathbf{I}), \qquad k = 1, \dots, K, \\ q(\mathbf{c}_i; \boldsymbol{\varphi}_i) &\sim Cat(\boldsymbol{\varphi}_i), \qquad i = 1, \dots, N, \end{aligned}$$

where $s_k^2 \in \mathbb{R}^+, \mathbf{m}_k \in \mathbb{R}^2$ and $\boldsymbol{\varphi}_i \in \mathbb{R}^K$.

The following ELBO remains as a function of the variational parameters $\mathbf{m}, \mathbf{s}^2$ and $\boldsymbol{\varphi}$ given by:

$$\begin{aligned} \mathcal{L}(\mathbf{m}, \mathbf{s}^2, \boldsymbol{\varphi}) &= \mathbb{E}_{q(\boldsymbol{\mu}, \mathbf{c})}[\log p(\mathbf{c}, \boldsymbol{\mu}, \mathbf{x}) - \log q(\boldsymbol{\mu}, \mathbf{c})] \\ &= \sum_{k=1}^{K} \mathbb{E}_{q(\boldsymbol{\mu}_k)}[\log p(\boldsymbol{\mu}_k)] + \sum_{n=1}^{N} (\mathbb{E}_{q(\mathbf{c}_i)}[\log p(\mathbf{c}_i)] + \mathbb{E}_{q(\boldsymbol{\mu}, \mathbf{c}_i)}[\log p(\mathbf{x}_i|\mathbf{c}_i, \boldsymbol{\mu})]) \\ &- \sum_{i=1}^{N} \mathbb{E}_{q(\mathbf{c}_i)}[\log q(\mathbf{c}_i)] - \sum_{k=1}^{K} \mathbb{E}_{q(\boldsymbol{\mu}_k)}[\log q(\boldsymbol{\mu}_k)]. \end{aligned}$$

The last equality can now be calculated analytically. For reasons of length, these calculations are not included in the text, the code in Appendix A can be consulted to see the value of each term.

Using the expression from Corollary 2.1, the parameter updates for each CAVI iteration are obtained:

$$\begin{aligned} q^*(\mathbf{c}_i) &\propto \exp\{\mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\mathbf{c}_i, \boldsymbol{\mu}, \mathbf{x}_i)]\} \\ &= \exp\{\mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\mathbf{c}_i)] + \mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\boldsymbol{\mu})] + \mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\mathbf{x}_i|\mathbf{c}_i, \boldsymbol{\mu})]\} \\ &\propto \exp\{\log p(\mathbf{c}_i) + \mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\mathbf{x}_i|\mathbf{c}_i, \boldsymbol{\mu})]\} \propto \exp\{\mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\mathbf{x}_i|\mathbf{c}_i, \boldsymbol{\mu})]\}. \end{aligned}$$

On the other hand, using the expression of the normal that governs the model's likelihood and the indicator vector $\mathbf{c}_i$, the following development is obtained, where

the sum is «fictitious», with only one of the terms being non-zero due to the entries $c_i^{(k)}$:

$$\mathbb{E}_{q(\boldsymbol{\mu})}[\log p(\mathbf{x}_i|\mathbf{c}_i,\boldsymbol{\mu})] = \sum_{k=1}^{K} c_i^{(k)} \mathbb{E}[-\frac{1}{2}\|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2] + const = \sum_{k=1}^{K} \mathbf{c}_i^{(k)}(-\frac{1}{2}\|\mathbf{x}_i - \boldsymbol{m}_k\|^2 - \frac{1}{2}s_k^2).$$

Remember that the $\boldsymbol{\varphi}_i$ are vectors whose components are probabilities and that they must sum to 1. Therefore, if the obtained result is normalized, a valid update is:

$$\varphi_i^{(k)} = \frac{\exp\{-\frac{1}{2}\|\mathbf{x}_i - \mathbf{m}_k\|^2 - \frac{1}{2}s_k^2\}}{\sum_{j=1}^{K}\exp\{-\frac{1}{2}\|\mathbf{x}_i - \mathbf{m}_j\|^2 - \frac{1}{2}s_j^2\}},$$

where $\varphi_i^{(k)}$ is the k-th component of $\varphi_i$ given $i \in \{1,\dots,N\}$ and $k \in \{1,\dots,K\}$.

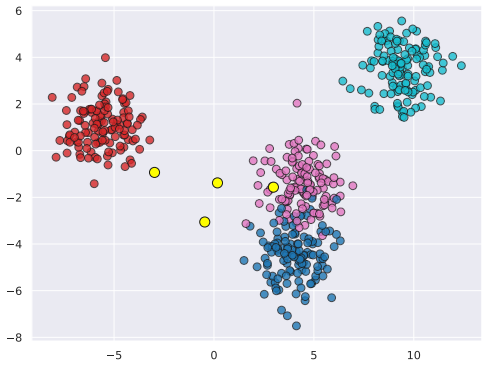
To update the variational parameters of the means $\mu_k$, the process is analogous. Since $\mu_k$ is a two-component vector, consider $\mu_k^{(j)}$ with $j = 1, 2$. Then:

$$q^*(\mu_k^{(j)}) \propto \exp\{\log p(\mu_k^{(j)}) + \sum_{i=1}^{N} \mathbb{E}_{-\mu_k^{(j)}}[\log p(x_i|\mathbf{c}_i,\boldsymbol{\mu})]\},$$

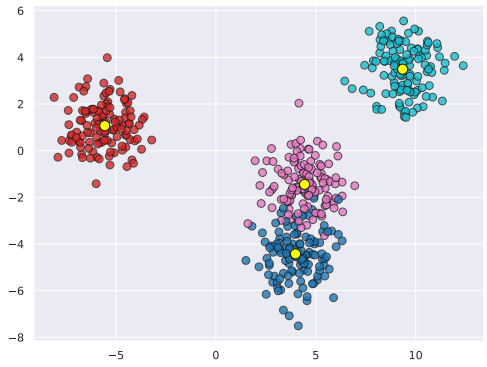
where the prior terms on the indicators $c_i$ have been included in the proportionality factor as they are fixed and do not depend on $\mu_k^{(j)}$. Taking logarithms and using the marginal prior of $\mu_k^{(j)}$:

$$\begin{aligned}
\log q^*(\mu_k^{(j)}) &= \log p(\mu_k^{(j)}) + \sum_{i=1}^{N} \mathbb{E}_{-\mu_k^{(j)}}\left[\log p\left(x_i^{(j)} \mid c_i, \boldsymbol{\mu}_j\right)\right] + \text{const} \\
&= \log p(\mu_k^{(j)}) + \sum_{i=1}^{N} \mathbb{E}_{-\mu_k^{(j)}}\left[c_i^{(k)} \log p\left(x_i^{(j)} \mid \mu_k^{(j)}\right)\right] + \text{const} \\
&= -\frac{\left(\mu_k^{(j)}\right)^2}{2\sigma^2} + \sum_{i=1}^{N} \mathbb{E}_{-\mu_k^{(j)}}[c_{ik}] \log\left(\frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{\left(x_i^{(j)} - \mu_k^{(j)}\right)^2}{2}\right\}\right) + \text{const} \\
&= -\frac{\left(\mu_k^{(j)}\right)^2}{2\sigma^2} + \sum_{i=1}^{N} \varphi_i^{(k)}\left(x_i^{(j)}\mu_k^{(j)} - \frac{\left(\mu_k^{(j)}\right)^2}{2}\right) + \text{const} \\
&= \left(\sum_{i=1}^{N} \varphi_i^{(k)} x_i^{(j)}\right)\mu_k^{(j)} - \left(\frac{1}{2\sigma^2} + \sum_{i=1}^{N}\frac{\varphi_i^{(k)}}{2}\right)\left(\mu_k^{(j)}\right)^2 + \text{const}.
\end{aligned}$$

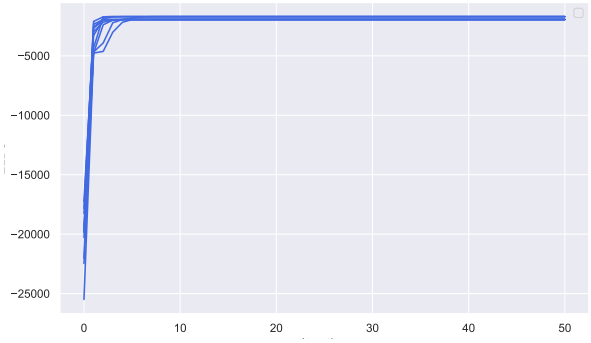
Taking exponentials, it is deduced that $q^*(\mu_k^{(j)})$ is a member of the two-parameter exponential family with sufficient statistics $\{\mu_k^{(j)}, (\mu_k^{(j)})^2\}$. In fact, this corresponds to a




(a)




(b)




(c)

Figure 2.2: Once the data has been generated, the model's variational parameters are randomly initialized, resulting in the data and centroids (in yellow) shown in (a). Once the coordinate ascent algorithm is applied, the final result is the centroids shown in (b). Finally, in (c) a monitoring of the ELBO for 10 applications of the algorithm with different initializations is shown. The graph reveals that convergence is very fast, but also that the ELBO has local minima as had been previously stated. Implementation in Appendix A.

normal distribution. If the expression of a generic normal distribution with unknown mean $m_k^{(j)}$ and variance $s_k^2$ is taken as a member of the exponential family and the expression of its sufficient statistics, by clearing, it is possible to obtain the following updates for the parameters of the bivariate distribution:

$$\mathbf{m}_k = \frac{\sum_i \varphi_i^{(k)} \mathbf{x}_i}{\frac{1}{\sigma^2} + \sum_i \varphi_i^{(k)}} \quad ; \quad s_k^2 = \frac{1}{\frac{1}{\sigma^2} + \sum_i \varphi_i^{(k)}}.$$

Once the updates are obtained, they can be used to apply the CAVI algorithm. To test the obtained variational method, 500 data points on the plane have been considered. Artificially generated, these data are known to be distributed in 4 Gaussian clusters with the identity covariance matrix, as indicated by the studied model. Figure 2.2 shows the results obtained by applying CAVI, as well as a study of the convergence of

the ELBO. The algorithm works really well, as the variational parameters have the final values that would be expected: as can be seen, the means $\mathbf{m}_k$ adjust to the theoretical centroids of the clusters; it can be seen that the variances $s_k$ are on the order of $10^{-3}$, this somehow means that the algorithm has converged and is «sure» of its approximation; for each point, $\boldsymbol{\varphi}_i$ is a vector with an entry very close to 1 and the rest practically zero. The Python implementation of this example can be consulted in Appendix A.

## 2.5 Stochastic Search

As an alternative to coordinate ascent, [17] introduces a new variational inference algorithm based on stochastic approximation of the ELBO gradient and control variables that try to regulate the variance of the approximation taken. The technique used is of vital importance since it is based on the reparameterization trick, which is the key to the *variational autoencoder*.

On the one hand, given a set of variational parameters $\phi$, it is possible to express the ELBO gradient as follows:

$$\nabla_\phi \mathcal{L} = \nabla_\phi \mathbb{E}_q[f(z)] + \nabla_\phi h(x, \phi).$$

Here, the function $\mathcal{L}$ has been divided into an intractable part (first term) and a tractable part (second term) on the right member. The objective is to perform an approximation by sampling $\nabla_\phi \mathbb{E}_q[f(z)]$ in order to search for the optimal variational parameters in the direction of the gradient. Thus, assuming the necessary regularity, it can be expressed:

$$\nabla_\phi \mathbb{E}_q[f(z)] = \nabla_\phi \int_z f(z) q(z|\phi) dz = \int_z f(z) \nabla_\phi q(z|\phi) dz = \int_z f(z) q(z|\phi) \nabla_\phi \log q(z|\phi) dz.$$

That is, $\nabla_\phi \mathbb{E}_q[f(z)] = \mathbb{E}_q[f(z) \nabla_\phi \log q(z|\phi)]$. Using Monte Carlo integration:

$$\nabla_\phi \mathbb{E}_q[f(z)] \simeq \frac{1}{L} \sum_{l=1}^{L} f(z^{(l)}) \log q(z^{(l)}|\phi),$$

where $z^{(l)} \sim q(z|\phi)$ has been taken for each $l = 1, \ldots, L$. Using this stochastic and unbiased approximation of the gradient and denoting it as $\xi$, in iteration $t$ we can update the variational parameters $\phi$ as:

$$\phi^{(t+1)} = \phi^{(t)} + \rho_t (\nabla_\phi h(x, \phi) + \xi_t). \tag{2.2}$$

Taking the step $\rho_t$ in such a way that $\sum_{t=1}^{\infty} \rho_t = \infty$ and $\sum_{t=1}^{\infty} \rho_t^2 < \infty$ guarantees convergence to a local optimum of $\mathcal{L}$ [19].

However, this estimator has a problem and that is that its variance can sometimes be very large. Given $L$ samples of a random vector $X$, the covariance of its unbiased

sample mean estimator $\bar{X}$ is such that $Cov(\bar{X}) = Cov(X)/L$. Therefore, if the values on the diagonal of $Cov(X)$ are large enough, the estimator will exhibit high variance and consequently, many samples will be needed to obtain a good estimate. A very large value of $L$ could make the algorithm computationally slow in practice. That is why we seek to reduce that variance and be able to compute the direction of the stochastic search much more efficiently.

To achieve this, a control variable is introduced. A control variable is a random variable highly correlated with a random variable of intractable expectation, but for which its expectation is tractable.

A variance reduction method is a method that modifies the function of a random variable $f(z)$ so that its expected value remains the same but its variance is reduced. This is where the control variable $g(z)$ comes into play, which approximates $f(z)$ well in regions of high probability and has an expectation over $q$ that is tractable in an analytical sense. Given $a \in \mathbb{R}$, it is defined:

$$\hat{f}(z) = f(z) - a(g(z) - \mathbb{E}_q[g(z)]).$$

Taking expectations, it is immediate that $\mathbb{E}_q[\hat{f}(z)] = \mathbb{E}_q[f(z)]$ and therefore, $\hat{f}$ can replace $f$ in the expression of $\mathcal{L}$.

Next, we will calculate $a \in \mathbb{R}$ in such a way that it minimizes the variance of $\hat{f}$. Thus, we have that:

$$\begin{aligned} Var(\hat{f}) &= Var(f - a(g - \mathbb{E}_q[g])) \\ &= Var(f) + Var(-a(g - \mathbb{E}_q[g])) + 2Cov(f, -a(g - \mathbb{E}_q[g])) \\ &= Var(f) + a^2 Var(g) - 2aCov(f, g), \end{aligned}$$

where we have used that $\mathbb{E}_q[g]$ is a constant quantity and basic properties of variance and covariance. Thus, deriving what was obtained with respect to $a$ and setting it to zero:

$$0 = -2Cov(f, g) + 2aVar(g),$$

$$a = \frac{Cov(f, g)}{Var(g)}.$$

Deriving again results in a constant positive quantity with respect to $a$, so the value obtained effectively minimizes the variance of $\hat{f}$. Normally the values of the quotient in the expression of $a$ are unknown but they can be approximated using the sample variance and covariance.

Substituting the value obtained for $a$ in the expression of the variance of $\hat{f}$:

$$Var(\hat{f}) = Var(f) + \left(\frac{Cov(f, g)}{Var(g)}\right)^2 Var(g) - 2\frac{Cov(f, g)}{Var(g)} Cov(f, g),$$

and dividing by the variance of $f$:

$$\frac{Var(\hat{f})}{Var(f)} = 1 - \frac{Cov(f,g)^2}{Var(f)Var(g)} = 1 - Corr(f,g)^2,$$

where $Corr$ is the Pearson correlation coefficient. Therefore, the greater the correlation between $f$ and $g$, the smaller the quotient $\frac{Var(\hat{f})}{Var(f)}$ will be, and therefore, the greater the reduction in the variance of $f$ with respect to $\hat{f}$.

Now, using the control variable $g$ and the reparameterization $\hat{f}$, the stochastic approximation to the gradient can be written as follows:

$$\nabla_\phi \mathbb{E}_q[f(z)] = \nabla_\phi \mathbb{E}_q[\hat{f}(z)] \simeq \hat{a}\nabla_\phi \mathbb{E}_q[g(z)] + \frac{1}{L}\sum_{l=1}^{L}(f(z^{(l)}) - \hat{a}g(z^{(l)}))\nabla_\phi \log q(z^{(l)}|\phi),$$

where $\hat{a}$ denotes the value of $a$ found to minimize $Var(f)$ calculated using sample values. Using this approximation and a highly correlated control variable along with Equation 2.2, it is possible to iterate and search for optimal variational parameters.

# 3

# The Autoencoder

The *Autoencoder* is a type of neural network trained to discover useful hidden representations in data. It consists of three main parts:

- An encoder function $f$ applies any data $\mathbf{x} \in \mathcal{D}$ to its code $\mathbf{z} = f(\mathbf{x})$.
- The *code* $\mathbf{z}$ is the hidden representation of the data generated by the encoder. It is the part of greatest interest as information about the data will be extracted from it.
- A decoder function $g$ that produces the network's output $\mathbf{r} = g(\mathbf{z})$.

It should be noted that both the *encoder* and the *decoder* can be seen as two neural networks themselves. The *encoder* is a neural network as deep as desired that takes data as input and produces its hidden representation $\mathbf{z}$ as output, usually with a different dimensionality than $\mathbf{x}$. On the other hand, the *decoder* is a neural network that has the code $\mathbf{z}$ as its input layer and that, through an arbitrary depth, produces as output a reconstruction $g(\mathbf{z}) = g(f(\mathbf{x}))$ which must have the same dimension as the data [6].

The training objective of an autoencoder is to make it capable of reconstructing the input data $\mathbf{x}$, for example images, as best as possible at the output. However, since there is special interest in the code $\mathbf{z}$ reflecting the data's own patterns, we seek to ensure that the *autoencoder* is not able to exactly represent $g(f(\mathbf{x})) = \mathbf{x}$ for all $\mathbf{x} \in \mathcal{D}$ through certain constraints. In this way, the model prioritizes aspects of the data that will be transmitted to the code $\mathbf{z}$, often resulting in the learning of useful properties of the data. The constraints that can achieve this effect are varied and range from injecting noise into the inputs to seeking the sparsity of the code $\mathbf{z}$, forcing the network to learn interesting data structures in order to achieve optimal performance in training [6].

This chapter mainly deals with deterministic autoencoders that have been part of the deep learning landscape for almost four decades. In this way, the aim is for the reader to be able to understand the architecture and basic notions of autoencoders before making the leap to their stochastic variant that uses latent variable models and that has been a fundamental part of the advance in generative artificial intelligence that has taken place in the last decade, the variational autoencoder.

The autoencoder is simply a special case of a neural network and can be trained as such using typical gradient-based optimization techniques. In this case, we want the reconstruction of the data to be as good as possible, that is, to minimize:

$$L(\mathbf{x}, g(f(\mathbf{x}))),$$

where $L$ is a certain loss function chosen depending on the problem. For example, the mean squared error.

Taking into account the approximation power of a neural network, this task can be trivial and, even worse, it can seem useless. However, remember that the goal is to make the *autoencoder* learn a useful representation $\mathbf{z}$ of the data as a side effect

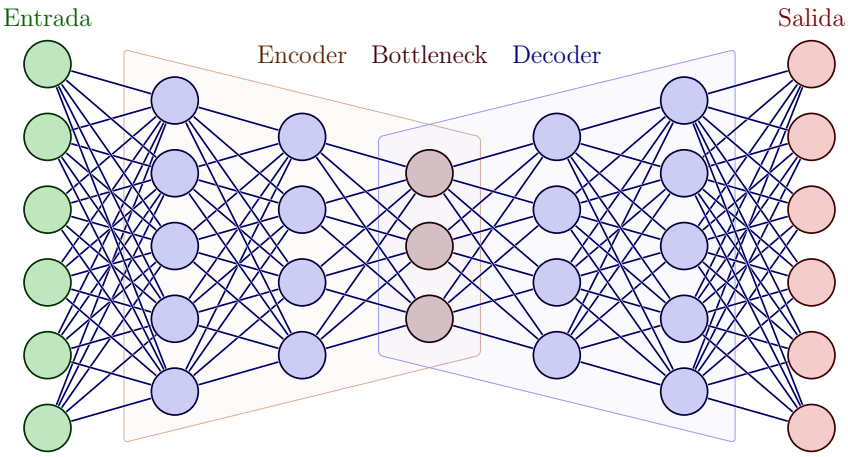



Figure 3.1: Autoencoder architecture. Made with the *TikZ* package of LaTeX. [22]

of the reconstruction task. It is therefore clear that the dimension of the code must be smaller than that of the data, because otherwise, even with linear activations, the neural network would have no problem copying the input data to the output without learning anything representative in the code $\mathbf{z}$. These autoencoders are called bottleneck autoencoders because they absorb the output of the encoder and the hidden representation layer is of a smaller dimensionality than the rest, see Figure 3.1.

The low dimensionality of the code usually results in reconstructions with some noise but with a model that reveals certain characteristics of the data. Even so, caution should be exercised with the depth of the encoder and the decoder, since if they are given many layers they may still be able to copy the data without learning anything of value about its distribution. One way to solve this problem is through regularization techniques.

## 3.1 Regularized Autoencoders

*Autoencoders* to which regularization is applied solve the problems presented above. Thus, it is possible to train very deep encoder and decoder architectures and even with a code with a dimension greater than that of the data. In short, regularized autoencoders are able to learn prominent features of the data even when they learn an identity function that performs the copying task perfectly. To read more about regularized *autoencoders*, chapter 14 of [6] can be consulted, on which most of the present section is based.

### Sparse Autoencoders

A sparse Autoencoder, instead of limiting the number of nodes or the depth of the network, implements a regularizing term that incentivizes a less dense code in training,

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{z}),$$

where $\Omega$ is normally the $L_1$ regularizing function that rewards the model for null or almost null values. Note that it is not the typical $L_1$ regularization used in a neural network that affects the network weights, but in this case, it affects the hidden representation $\mathbf{z}$,

$$\Omega(\mathbf{z}) = \lambda \sum_{i=1}^{M} |z_i|,$$

where $M$ is the dimension of the code and $\lambda > 0$ is the regularizing parameter that indicates the penalty suffered due to $z_i$ that are very different from zero.

### *Denoising Autoencoders*

This type of autoencoder, also known as DAE, is an autoencoder whose training process consists of eliminating noise from the original data. A training example $\mathbf{x}$ is obtained and, from a certain process $C(\tilde{\mathbf{x}}|\mathbf{x})$, a corrupted version $\tilde{\mathbf{x}}$. The objective is to minimize a loss function $\mathcal{L}(\mathbf{x}, f(g(\tilde{\mathbf{x}})))$.

Looking at the autoencoder from a probabilistic point of view:

$$p_{\text{reconstruccion}}(\mathbf{x}|\tilde{\mathbf{x}}) = p_{\text{decoder}}(\mathbf{x}|\mathbf{z}),$$

where $\mathbf{z}$ is the code or the output of the encoder $f(\tilde{\mathbf{x}})$. The distribution associated with the decoder in the case that was being dealt with is given by $g(\mathbf{z})$. Thus, a loss function in this case can be:

$$\mathcal{L}(\mathbf{x}, f(g(\tilde{\mathbf{x}}))) = -\log p_{\text{reconstruccion}}(\mathbf{x}|\tilde{\mathbf{x}}) = -\log p_{\text{decoder}}(\mathbf{x}|\mathbf{z} = f(\tilde{\mathbf{x}})),$$

which makes sense, since by minimizing this loss, the density of the reconstruction of the data is being maximized. As in the models seen previously, an approximate minimization is performed using gradient-based techniques on the average of the data and the average of the noise that can be injected into them. This leads to the total cost:

$$-\mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[\mathbb{E}_{\tilde{\mathbf{x}}\sim C(\tilde{\mathbf{x}}|\mathbf{x})}[\log p_{\text{decoder}}(\mathbf{x}|\mathbf{z} = f(\tilde{\mathbf{x}}))]].$$

An alternative to likelihood can be the use of *Score Matching*, where $\nabla_{\mathbf{x}} \log p(\mathbf{x})$ is estimated. The vector field associated with the score points towards the regions of highest probability density in terms of the data distribution, that is, from the corrupted versions it points towards the nearest true data which are usually found in a subvariety of the space. The process is represented in Figure 3.2.

By learning to remove noise from data, the DAE is forced to learn aspects of its structure. In the case of images, learning that nearby pixels are strongly correlated with each other makes it possible to correct corrupted pixels.

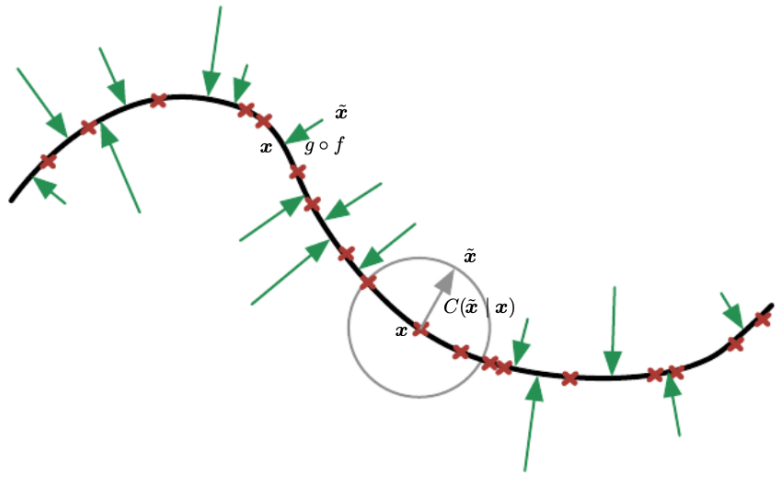



Figure 3.2: DAE reconstruction process where the data, which is concentrated in a variety of space, is represented in red. The corruption process is in gray and the learned vector field that estimates the *score* is determined by the green arrows. Figure extracted from [6].

## 3.2 Applications

Autoencoders are especially useful in manifold learning tasks, a branch of machine learning that starts from the idea that data usually resides in a low-dimensional manifold within the ambient space. These models allow learning the structure of said manifold[1] [6].

Thanks to their ability to learn compact representations, autoencoders are effective in dimensionality reduction and information retrieval. They represent a non-linear alternative to principal component analysis, which improves tasks such as the efficient search for similar examples, for example in natural language processing.

In addition, they stand out in applications such as anomaly detection in cybersecurity or financial fraud, where the probabilistic approach of the variational autoencoder will offer additional advantages.

[1] One way to characterize manifolds is by their tangent planes, which describe how data varies infinitesimally.

# 4 The Variational Autoencoder

The *Variational Autoencoder* (VAE) is one of the so-called deep latent variable models, it combines probabilistic modeling and deep learning techniques to perform generative, representation and semi-supervised learning tasks in an interpretive way. Later it will be seen that the ability to interpret the latent space that VAEs build is one of the greatest attractions of these models, since it allows performing the aforementioned tasks without being a statistical black box like conventional neural networks can be.

VAEs are proposed for the first time in [11]. In this article, the authors express the desire to perform efficient approximate inference and learning in continuous probabilistic models with latent variables that have intractable posterior distributions.

By a simple reparameterization of the ELBO, they find an unbiased and differentiable estimator based on simple standard sampling techniques. The differentiability in turn provides an unbiased estimator of the ELBO gradient, called in the text *Stochastic Gradient Variational Bayes*, which will allow the use of the usual gradient ascent techniques to take the ELBO to a local maximum through the *Autoencoding Variational Bayes* algorithm. At the same time, assumptions as strong about the form of the posterior are not made as the use of the mean-field family does, it will not be necessary to calculate expectations in an analytical form, which in many cases are intractable, and computationally expensive techniques such as MCMC will not be used. The resulting probabilistic model combined with the use of neural networks gives rise to the Variational Autoencoder.

## 4.1 Problem Statement

In this section, the inference problem will be stated step by step as it is done in [11]. Consider a set of i.i.d. data with continuous latent variables for each element where the aim is to perform efficient approximate Bayesian inference.

Let $\mathbf{x} = \{x^{(i)}\}_{i=1}^{N}$ be an i.i.d. sample of a random variable $\mathbf{x}$ which is assumed to depend on a hidden or latent process that is determined by the random vector $\mathbf{z}$. Let $\theta$ be a parameter vector that determines what will now be called the generative model, formed by the prior distribution $p_\theta(\mathbf{z})$ and the model's likelihood $p_\theta(\mathbf{x}|\mathbf{z})$.

On the other hand, consider the inference model $q_\phi(\mathbf{z}|\mathbf{x})$. As is usual, $\phi$ denotes the set of variational parameters that must be optimized in such a way that the inference model approximates the true posterior:

$$q_\phi(\mathbf{z}|\mathbf{x}) \approx p_\theta(\mathbf{z}|\mathbf{x}).$$

The first and most obvious similarity with *autoencoders* arises if we think of $q_\phi(\mathbf{z}|\mathbf{x})$ as a probabilistic *encoder* that, given $\mathbf{x}$, provides the code $\mathbf{z}$. Analogously, it is possible to perceive the model $p_\theta(\mathbf{x}|\mathbf{z})$ as a probabilistic *decoder*. When the variational inference algorithm is formulated, this similarity will be explored in depth.

The inference model can be any probabilistic graphical model verifying:

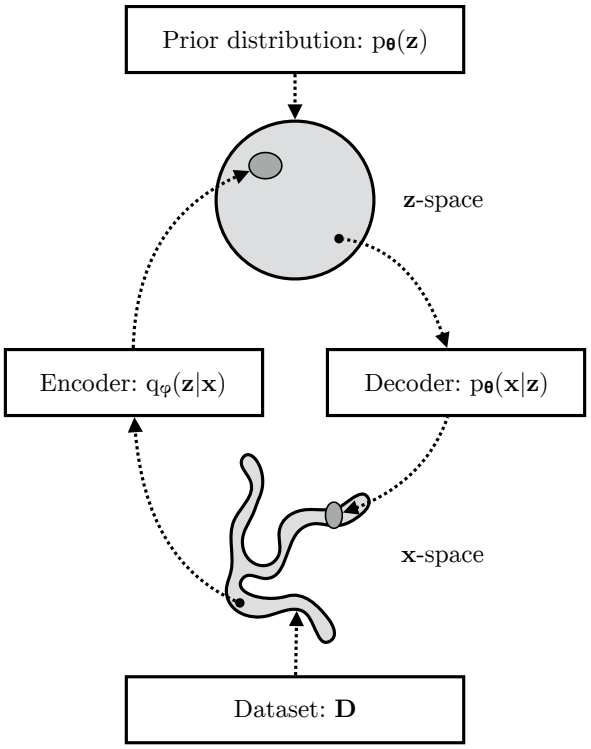



Figure 4.1: Scheme of the variational model. The assumption of a simple prior $p_\theta(\mathbf{z})$ for the latent variables can be appreciated, while the sample space of the data is highly complicated to describe in probabilistic terms. To interconnect these two spaces, the Encoder, $q_\phi(\mathbf{z}|\mathbf{x})$, and the Decoder, $p_\theta(\mathbf{x}|\mathbf{z})$, are used along with neural networks. Figure extracted from [12].

$$q_\phi(\mathbf{z}|\mathbf{x}) = q_\phi(z_1, \ldots, z_M|\mathbf{x}) = \prod_{j=1}^{M} q_\phi(z_j|Pa(z_j), \mathbf{x}).$$

where $Pa(z_j)$ denotes the set of parent nodes of $z_j$ in the directed graph. The idea in this case is for the VAE to be a deep latent variable model where the variational distribution $q_\phi(\mathbf{z}|\mathbf{x})$ is parameterized by a neural network. In this way, the variational parameters $\phi$ will be the weights and biases of the neural network. In short, if for example we have a multivariate normal as an approximate posterior, the model is:

$$(\mu, \log\sigma) = RedNeuronalCodificadora_\phi(\mathbf{x}),$$

$$q_\phi(z|x) = \mathcal{N}(\mu, diag(\sigma)).$$

To simplify the initial theoretical development, it is assumed that there is no explicit hierarchy between the model's latent variables. That is, the graphical model given by Figure 4.2.

The goal is to build an efficient approximate inference and learning algorithm in a scenario of intractability and a large amount of data that makes it preferable to update

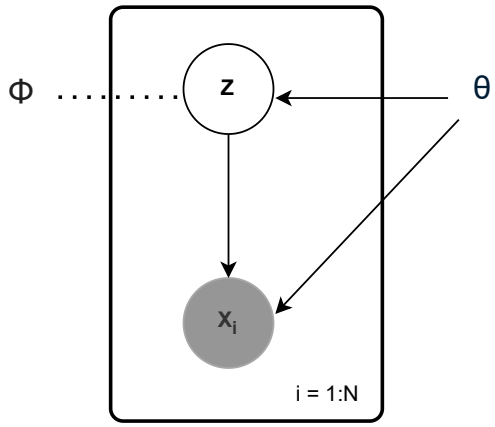



Figure 4.2: Probabilistic model used. Figure created with *draw.io*.

the variational parameters using small batches[1] of data in each iteration, or even a single point [11]. Under these conditions, the aims are:

- Efficiently approximate the MAP and/or Maximum Likelihood estimators for the model parameters $\theta$. They can be of interest in themselves and can also allow the generation of new artificial data similar to the real ones.
- Perform efficient posterior inference on the latent variables $\mathbf{z}$. Useful for representation or encoding tasks.
- Perform approximate marginal inference on the variable $\mathbf{x}$. Useful in tasks such as noise reduction.

## 4.2 Methodology

The ELBO will be, as in other variational inference methods, the objective of the optimization. $\mathcal{L}$ will be computed for each data point $x^{(i)}$ since in the proposed case we have by independence that:

$$\log p_\theta(x^{(1)},\dots,x^{(n)}) = \sum_{i=1}^{N} \log p_\theta(x^{(i)}),$$

and remember that it is possible to obtain that:

$$\log p_\theta(\mathbf{x}^{(i)}) = KL(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z}|\mathbf{x}^{(i)})) + \mathcal{L}(\theta,\phi,\mathbf{x}^{(i)}),$$

which was the expression that gave meaning to the maximization of the ELBO.

It is also worth remembering the following two expressions that will be helpful when it comes to obtaining the desired estimator. The first is the usual definition of the ELBO:

[1] Commonly known as *mini-batches*

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}[\log p(\mathbf{z}, \mathbf{x}^{(i)})] - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}[\log q(\mathbf{z})], \tag{4.1}$$

and the alternative ELBO expression obtained in Section 2.1:

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = -KL(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(\mathbf{i})})}[\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})]. \tag{4.2}$$

The goal is to maximize the ELBO gradient with respect to the variational parameters $\phi$ and the generative model parameters $\theta$ jointly. As just verified, having a set of i.i.d. data, it is possible to calculate:

$$\mathcal{L}(\theta, \phi, \mathcal{D}) = \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\theta, \phi, \mathbf{x}).$$

Calculating the gradient of $\mathcal{L}$ with respect to the model parameters $\theta$ is simple since the gradient can be introduced into the expectations of Equation 4.1 and an unbiased estimator can be calculated using Monte Carlo sampling.

The same does not happen with respect to the variational parameters of the model, since generally:

$$\nabla_\phi \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] \neq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\nabla_\phi f(\mathbf{z})].$$

That is, the derivatives with respect to $\phi$ can be somewhat problematic because the expectations are taken with respect to $q_\phi$. In this type of problem, the Monte Carlo gradient estimator together with the *Score* is usually used, which is:

$$\nabla_\phi \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z}) \nabla_\phi \log q_\phi(\mathbf{z})] \simeq \frac{1}{L} \sum_{l=1}^{L} f(\mathbf{z}) \nabla_\phi \log q_\phi(\mathbf{z}^{(l)}), \tag{4.3}$$

where for a fixed $i \in \{1, \ldots, N\}$, we have $z^{(l)} \sim q_\phi(\mathbf{z}|x^{(i)})$ for each $l \in \{1, \ldots, L\}$. The problem with this estimator is that it exhibits high variance, as seen in Section 2.5, which is not desirable in terms of time efficiency. The strategy found in [11] is not exactly the one seen in that section, but it is based on a control variable that is taken in such a way that it leads to a reduction in the variance of the estimator.

### *The reparameterization trick*

Let $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ be a conditional distribution. There are times when it is possible to express $\mathbf{z} = g_\phi(\epsilon, \mathbf{x})$, where $\epsilon$ is an auxiliary variable with an independent marginal distribution $p(\epsilon)$ and $g_\phi(.)$ is an invertible and differentiable function. This will be very useful since it will allow writing the random vector $\mathbf{z}$ as a deterministic variable[2]. Several advantages arise from taking this expression, as it will allow computing gradients with respect to the latent variables and performing backpropagation of the parameter gradients when implementing neural networks, as well as reducing the variance of the

[2]It is obvious that it will not be truly deterministic, the variable $\epsilon$ provides the randomness.

approximations and making learning and inference much more efficient. This process is called the *reparameterization trick* [11].

In the following result, the estimator that will be used in the algorithm is obtained:

**Proposition 4.1.** *Given the directed graphical model defined by Figure 4.2, a random vector* $\boldsymbol{z} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x})$*, an independent random variable* $\epsilon \sim p(\epsilon)$ *and an invertible and differentiable transformation* $g_\phi(.)$ *such that* $\boldsymbol{z} = g(\epsilon, \boldsymbol{x})$*. Then,*

$$\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[f(z)] \simeq \frac{1}{L}\sum_{l=1}^{L} f(g_\phi(\boldsymbol{x}, \epsilon^{(l)})),$$

*where* $\epsilon^{(l)} \sim p(\epsilon)$*, for any* $l \in \{1, \dots, L\}$*.*

Proof:

Consider the change of variable $\mathbf{z} = g(\epsilon, \mathbf{x})$. Then, by the probabilistic change of variable theorem, if $J_{g_\phi}(\epsilon, \mathbf{x})$ represents the Jacobian matrix—taken with respect to $\epsilon$—associated with said change of variable, we have that:

$$q_\phi(\mathbf{z}|\mathbf{x}) = p(\epsilon)\left|\det J_{g_\phi}(\epsilon, \mathbf{x})\right|^{-1}.$$

Furthermore, in terms of volumes, we can go from the latent space of $\mathbf{z}$ to that of $\epsilon$ using the Jacobian matrix along with the integral change of variable theorem, that is:

$$\prod_i dz_i = \left|\det J_{g_\phi}(\epsilon, \mathbf{x})\right| \prod_i d\epsilon_i.$$

Returning to the first equality and using the latter in it:

$$q_\phi(\mathbf{z}|\mathbf{x}) \prod_i dz_i = p(\boldsymbol{\epsilon})\left|\det J_{g_\phi}(\epsilon, \mathbf{x})\right|^{-1}\left|\det J_{g_\phi}(\epsilon, \mathbf{x})\right| \prod_i d\epsilon_i = p(\epsilon)\prod_i d\epsilon_i.$$

Therefore, if we use the usual notation $d\mathbf{z} = \prod_i dz_i$ and $d\epsilon = \prod_i d\epsilon_i$, it is possible to see that:

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] = \int q_\phi(\mathbf{z}|\mathbf{x}) f(\mathbf{z}) d\mathbf{z} = \int p(\epsilon) f(g_\phi(\mathbf{x}, \epsilon)) d\epsilon = \mathbb{E}_{p(\epsilon)}[f(g_\phi(\mathbf{x}, \epsilon))].$$

Finally, applying Monte Carlo, the following unbiased estimator is obtained:

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] \simeq \frac{1}{L}\sum_{l=1}^{L} f(g_\phi(\mathbf{x}, \epsilon^{(l)})),$$

where $\epsilon^{(l)} \sim p(\epsilon)$ has been sampled for each $l \in \{1, \dots, L\}$.

■

Unlike the estimator (4.3) which can have an excessively large variance since it depends on $q_\phi(\mathbf{z}|\mathbf{x})$, the variance of the estimator that has just been obtained now depends on $p(\epsilon)$, which acts as a control variable. By choosing this auxiliary distribution appropriately, it will be possible to control the variance when estimating $\mathcal{L}$ and its gradient.

It is important to indicate in which cases it is feasible to perform the change of variable and how it would be carried out:

- In the case where the conditioned density $q_\phi(\mathbf{z}|\mathbf{x})$ is invertible and its inverse can be calculated, it is enough to take $g_\phi(\mathbf{x}, \epsilon)$ as the inverse and $\epsilon \sim \mathcal{U}(0, \mathbf{I})$. Examples of probability distributions that verify this are the exponential, Pareto, Cauchy or Weibull.
- Probability distributions that can be parameterized by a location parameter and a scale parameter whose standard distribution is (loc = 0, esc = 1). Taking $\epsilon$ as the standard distribution of the respective family of distributions, $g_\phi(\mathbf{x}, \epsilon) = loc + esc \cdot \epsilon$ is defined. Examples: normal distribution, Laplace, Student's t, uniform, etc.
- Probability distributions that can be expressed as transformations of other auxiliary random variables. The log-normal, gamma, beta, chi-square or F-Snedecor distributions are some of those that verify this property.

### *Estimators*

Applying the estimator obtained thanks to the reparameterization trick to Equation 4.1, the first unbiased approximation of the ELBO is obtained:

$$\tilde{\mathcal{L}}^A(\theta, \phi, \mathbf{x}^{(i)}) = \frac{1}{L} \sum_{l=1}^{L} \Big( \log p_\theta(\mathbf{x}^{(i)}, \mathbf{z}^{(i,l)}) - \log q_\phi(\mathbf{z}^{(i,l)}|\mathbf{x}^{(i)}) \Big), \tag{4.4}$$

where $\mathbf{z}^{(i,l)} = g_\phi(\epsilon^{(i,l)}, \mathbf{x}^{(i)})$ and $\epsilon^{(l)} \sim p(\epsilon)$. The notation presented in the original article [11] is followed.

Paying attention to the alternative ELBO expression 4.2, in some cases the KL-divergence can be calculated analytically so that only the term $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})]$ requires estimation by sampling. Thus, analogously to estimator A, another estimator is obtained that will normally have even less variance than the previous one:

$$\tilde{\mathcal{L}}^B(\theta, \phi, \mathbf{x}^{(i)}) = -KL(q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) \| p_\theta(\mathbf{z})) + \frac{1}{L} \sum_{i=1}^{L} \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)}), \tag{4.5}$$

where again $\mathbf{z}^{(i,l)} = g_\phi(\epsilon^{(i,l)}, \mathbf{x}^{(i)})$ and $\epsilon^{(l)} \sim p(\epsilon)$.

Using either of these two estimators, thanks to their low variance given $N$ points in a dataset $X$, it is possible to construct the estimator using batches of size $P < N$, especially with a view to iterative gradient ascent:

$$\mathcal{L}(\theta, \phi, X) \simeq \tilde{\mathcal{L}}^P(\theta, \phi, X^P) = \frac{N}{P} \sum_{i=1}^{P} \tilde{L}(\theta, \phi, \mathbf{x}^{(i)}),$$

where the batch $X^P = \{\mathbf{x}^{(i)}\}_{i=1}^P$ is a random sample of size $P$ from the total dataset $X$ of $N$ points whose use will make the optimization algorithm more efficient. It is possible to take derivatives of these estimators and therefore, use these derivatives with gradient-based stochastic optimization methods.

In addition, thanks to the change of variable that has been made to calculate these approximations, it is possible to immediately obtain an unbiased estimator of $\nabla_{\theta,\phi}\mathcal{L}$, simply by taking the gradient of the estimator. Indeed:

$$\mathbb{E}_{p(\epsilon)}[\nabla_{\theta,\phi}\tilde{\mathcal{L}}(\theta, \phi, x^{(i)})] = \nabla_{\theta,\phi}\mathbb{E}_{p(\epsilon)}[\tilde{\mathcal{L}}(\theta, \phi, x^{(i)})] = \nabla_{\theta,\phi}\mathcal{L}(\theta, \phi, x^{(i)}),$$

where the unbiasedness of $\tilde{\mathcal{L}}$ has been used.

It is interesting to analyze expression (4.5) provided by estimator B, as it is an objective function that makes the relationship with standard *autoencoders* even more evident. On the one hand, the KL-divergence term acts as a regularizing expression of the model in a similar way to how it occurred in *sparse autoencoders*. On the other hand, the sum $\sum_{i=1}^{L} \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})$ is sought to be maximized, where all the summands are negative or zero, therefore we want $\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})$ to approach zero, that is, that $p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})$ approaches 1 as much as possible, which means the model's likelihood is being maximized. This term is the negative reconstruction error that appears in the loss function of an *autoencoder*.

## *Algorithm and considerations*

**Algorithm 2** Autoencoding Variational-Bayes [11]

Initialize parameters $\theta, \phi$.
**Repeat**
  $X^M \leftarrow$ Take a random sample of size $p$.
  $\epsilon \leftarrow$ Random samples from $p(\epsilon)$.
  $g \leftarrow \nabla_{\theta,\phi}\tilde{\mathcal{L}}^P(\theta, \phi; \mathbf{X}^P, \epsilon)$.
  $\theta, \phi \leftarrow$ Update parameters using stochastic gradient descent Compute ELBO.
**until:** $\mathcal{L}$ *converges.*
**Output:** $\theta, \phi$.

When calculating the gradient estimator, it is necessary to compute $\log q_\phi(\mathbf{z}|\mathbf{x})$ taking into account the change of variable provided by the reparameterization trick $\mathbf{z} = g_\phi(\mathbf{x}, \epsilon)$. Since it was initially required that the transformation $g_\phi(.)$ had an inverse, then by the probabilistic change of variable theorem we have that:

$$q_\phi(\mathbf{z}|\mathbf{x}) = p(\epsilon) \left| \det J_{g_\phi}(\epsilon, \mathbf{x}) \right|^{-1},$$

and taking logarithms,

$$\log q_\phi(\mathbf{z}|\mathbf{x}) = \log p(\epsilon) - \log \left| \det J_{g_\phi}(\epsilon, \mathbf{x}) \right|.$$

Next, it will be checked that it is possible to construct transformations $g_\phi(.)$ that result in highly flexible inference models $q_\phi(\mathbf{z}|\mathbf{x})$ while the Jacobian matrix $J_{g_\phi}(\epsilon, \mathbf{x})$ is easy to obtain.

### *Choices for the variational density $q_\phi(\boldsymbol{z}|x)$*

A first option is to use a **factorized multivariate normal distribution** [12], $q_\phi(\mathbf{z}|\mathbf{x}) \sim \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, diag(\boldsymbol{\sigma}^2))$, as in the example presented at the beginning of the chapter, where:

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = RedNeuronalCodificadora_\phi(\mathbf{x}),$$

$$q_\phi(\mathbf{z}|\mathbf{x}) = \prod_{i=1}^{M} q_\phi(z_i|\mathbf{x}) = \prod_{i=1}^{M} \mathcal{N}(z_i; \mu_i, \sigma_i^2).$$

To apply the reparameterization trick, $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$ is taken and it is immediate that:

$$z = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon},$$

where $\odot$ denotes element-wise product between the vectors involved. Indeed, the Jacobian matrix $J_{g_\phi}(\epsilon, \mathbf{x})$ is simple, since we have that:

$$J_{g_\phi}(\epsilon, \mathbf{x}) = diag(\sigma),$$

that is, a diagonal matrix where each entry is the standard deviation of the corresponding latent component. In this way:

$$\log q_\phi(\mathbf{z}|\mathbf{x}) = \log p(\epsilon) - \log \left| \det J_{g_\phi}(\epsilon, \mathbf{x}) \right| = \sum_{i=1}^{M} (log \mathcal{N}(\epsilon_i; 0, 1) - \log \sigma_i).$$

As mentioned previously, it is possible to use the estimator $\tilde{\mathcal{L}}^B$ when $KL[q(\mathbf{z}|\mathbf{x})|p(\mathbf{z})]$ can be calculated analytically. An example where this occurs is when the prior $p_\theta(\mathbf{z}) \sim \mathcal{N}(0, \mathbf{I})$ and the posterior $q_\phi(\mathbf{z}|\mathbf{x})$ is the one given in this case. Indeed, if $M$ is the dimensionality of the latent space:

$$\begin{aligned} -KL(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z})) &= \int q_\phi(\mathbf{z}|\mathbf{x})(\log p_\theta(\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x}))d\mathbf{z} \\ &= \int \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2) \log \mathcal{N}(z; \mathbf{0}, \mathbf{I}) d\mathbf{z} + H(q_\phi) \\ &= \frac{1}{2} \sum_{i=1}^{M} (1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2). \end{aligned}$$

The drawback of using this distribution is that it does not capture possible correlations between the latent variables that could lead to a more flexible model that better explains the data, that is, a richer latent space.

To solve this, the case of a **normal distribution with a full covariance matrix** can be considered [12]:

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

Let the reparameterization be given by:

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

In this case, it is required that $L$ is a lower or upper triangular matrix with a non-zero diagonal. It is clear that the off-diagonal elements are responsible for the correlations between the elements of the vector $\mathbf{z}$.

To find $L$, consider $\Sigma$, which, being the covariance matrix of a well-defined normal distribution, is known to be Hermitian and positive definite, therefore its Cholesky factorization exists. The triangular matrix $L$ is taken as the one involved in said decomposition:

$$\Sigma = LL^T.$$

Indeed, in the presented reparameterization, the matrix L corresponds to the one used in the Cholesky decomposition of this matrix:

$$\begin{aligned}\Sigma &= \mathbb{E}[(\mathbf{z} - \mathbb{E}[\mathbf{z}])(\mathbf{z} - \mathbb{E}[\mathbf{z}])^T] \\ &= \mathbb{E}[(\mu + L\epsilon - \mu)(\mu + L\epsilon - \mu)^T] = \mathbb{E}[L\epsilon(L\epsilon)^T] \\ &= L\mathbb{E}[\epsilon\epsilon^T]L^T = LCov(\epsilon)L^T = LIL^T = LL^T.\end{aligned}$$

This reparameterization is useful since it is immediate that $J_{g_\phi}(\epsilon, \mathbf{x}) = L$. Being a triangular matrix, it is very simple to compute its determinant:

$$\log q_\phi(\mathbf{z}|\mathbf{x}) = \log p(\epsilon) - \log \left|\det J_{g_\phi}(\epsilon, \mathbf{x})\right| = \sum_{i=1}^{M} (log\mathcal{N}(\epsilon_i; 0, 1) - \log |L_{ii}|).$$

One way to construct a matrix $L$ that meets the desired properties proposed in [12] is to learn it jointly with the vector $\boldsymbol{\mu}$:

$$(\mu, \log\sigma, L') = RedNeuronalCodificadora_\phi(\mathbf{x}),$$
$$L = L_{mask} \odot L' + diag(\sigma),$$
$$z = \mu + L\epsilon.$$

Here, a mask matrix $L_{mask}$ is used that has zeros on and above the diagonal. The rest of the entries are ones. In this way, the learned entries of the matrix $L'$ below the diagonal are kept and the diagonal is completed with the learned variance vector $\sigma^2$.

## 4.3 Approximation Theorem

The following is a technical lemma that in statistical literature is usually known as the Rosenblatt transformation [20] and that will help to prove the next theorem.

In the lemma it will be assumed that, given a vector of N random variables, the distribution function of each one of them conditioned on the previous variables is continuous and strictly increasing, thus ensuring that we are in the case of interest where the inverse of the conditioned univariate distribution functions exists. Thus, cases of little practical interest such as continuous probability distributions that have constant densities in certain intervals are discarded.

**Lemma 4.1** (Rosenblatt Transformation, [20])**.** *Consider a continuous N-dimensional probability distribution given by a continuous density function $p^*(\boldsymbol{x})$ on $A = \{\boldsymbol{x} \in \mathbb{R}^N : p^*(\boldsymbol{x}) > 0\}$. Let $\boldsymbol{z} = (z_1, \ldots, z_N)$ be a vector of independent random variables that follow a standard normal $\mathcal{N}(0,1)$ with distribution function $G$. In turn, given $i \in \{1, \ldots, N\}$ we define:*

$$F_i(x_i) = F(x_i | x_1, \ldots, x_{i-1}),$$

*where $F(x_i|x_1, \ldots, x_{i-1})$ is the probability distribution function, which is required to be continuous and strictly increasing, associated with the conditional $p^*(x_i|x_1, \ldots, x_{i-1})$. Then, the transformation $f : \mathbb{R}^N \to \mathbb{R}^N$ given by:*

$$f(\boldsymbol{z}) = (F_1^{-1}(G(z_1)), F_2^{-1}(G(z_2)), \ldots, F_N^{-1}(G(z_N)))$$

*recovers the joint distribution $p^*$ in its entirety.*

Proof:

Note in the first instance that the function $f$ in the statement is well-defined, since the inverse of the conditioned distribution functions will be guaranteed as they are continuous, strictly increasing and associated with a single variable.

On the other hand, to demonstrate that the distribution given by the transformation $f$ applied to $\mathbf{z} \sim p(\mathbf{z})$ generates the distribution associated with $p^*(\mathbf{x})$, it must be demonstrated by the change of variable theorem that:

$$p^*(\mathbf{x}) = p(\mathbf{z}) |\det J_f(\mathbf{z})|^{-1}.$$

In fact, by the definition of the components of $f$, it is immediate that its Jacobian matrix is lower triangular, so:

$$\det J_f(\mathbf{z}) = \prod_{i=1}^{N} \frac{\partial f_i}{\partial z_i}(\mathbf{z}) = \prod_{i=1}^{N} \frac{d}{dz_i}[F_i^{-1}(G(z_i))] = \prod_{i=1}^{N} \frac{1}{p^*(x_i|x_1,\dots,x_{i-1})} p(z_i),$$

where the expression of the derivative for the inverse of a real function of a real variable has been used and that the derivative of the distribution functions in continuous variables recovers the associated density function. In the denominator, it has been applied that $F_i^{-1}(G(z_i))$ produces a sample from $p^*(x_i|x_1,\dots,x_{i-1})$ by the inverse transform method [21]. Finally, using the independence between the variables contained in $\mathbf{z}$:

$$\begin{aligned} p(\mathbf{z})|\det J_f(\mathbf{z})|^{-1} &= p(\mathbf{z}) \prod_{i=1}^{N} \frac{p^*(x_i|x_1,\dots,x_{i-1})}{p(z_i)} = \prod_{i=1}^{N} p^*(x_i|x_1,\dots,x_{i-1}) \\ &= p^*(x_1)p^*(x_2|x_1)\cdot\dots\cdot p^*(x_N|x_1,\dots,x_{N-1}) = p^*(\mathbf{x}). \end{aligned}$$

■

The following theorem shows the great probabilistic approximation power that a standard VAE can have. It is a generalization of the result for the one-dimensional case proposed in [5].

**Theorem 4.1.** *Consider an N-dimensional probability distribution $p^*(\boldsymbol{x})$ that verifies that $p^*(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \mathbb{R}^N$ and whose density function is $C^\infty(\mathbb{R}^N)$ with bounded partial derivatives. Furthermore, the conditioned distribution functions, as defined in Lemma 4.1, are strictly increasing and continuous. Then there exists a VAE that, provided with a sufficiently expressive encoder and decoder, is capable of approximating $p^*(\boldsymbol{x})$ with arbitrary error when learning the approximate variational posterior $q(\boldsymbol{z}|\boldsymbol{x})$.*

Proof:

It is known that the VAE optimizes

$$\mathcal{L} = \log p(\mathbf{x}) - KL(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})),$$

where given $\sigma \in \mathbb{R}$ we are going to take $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|f(\mathbf{z}), \sigma^2\mathbf{I})$, given $\mathbf{z} \in \mathcal{N}(\mathbf{0},\mathbf{I})$ as a priori, and $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(z|\mu(\mathbf{x}), \Sigma(\mathbf{x}))$. The dimension of the latent space is taken to be the same as that of the sample space.

The best possible solution occurs when it is verified that:

$$p(\mathbf{x}) = p^*(\mathbf{x}), \quad \forall \mathbf{x} \in \mathbb{R}^n \qquad \wedge \qquad KL(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) = 0.$$

In that case, the VAE will have fulfilled its purpose and will also have learned the distribution $p^*$. Therefore, it must be proven that there exist $f$, $\mu$ and $\Sigma$ verifying the above. Indeed, by finding the functions that verify the above, we will have finished, since the existence of neural networks that approximate them with arbitrary error $\epsilon > 0$ is guaranteed by the universal approximation theorem, and with it the existence of an *encoder* that is capable of learning $\mu$ and $\Sigma$ and a *decoder* that does the same with the function $f$ applied to the latent space.

Thus, the probability distribution functions $F_i$ defined as in Lemma 4.1 are considered, then by said result:

$$f(\mathbf{z}) = (F_1^{-1}(G(z_1)), F_2^{-1}(G(z_2)), \dots, F_N^{-1}(G(z_N)))$$

follows the distribution $p^*(\mathbf{x})$, then:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} = \int \mathcal{N}(\mathbf{x}|f(\mathbf{z}), \sigma^2\mathbf{I})p(\mathbf{z})d\mathbf{z} = \mathbb{E}_{p(z)}[\mathcal{N}(\mathbf{x}|f(\mathbf{z}), \sigma^2\mathbf{I})].$$

and if we take $\sigma \to 0$, then the normal becomes a degenerate distribution at the point $f(\mathbf{z})$ and thus, $p(\mathbf{x})$ converges to $p^*(\mathbf{x})$ as desired.

Now, for this distribution to be learnable in the usual ELBO-based VAE training, an encoder must be constructible in such a way that $q(\mathbf{z}|\mathbf{x})$ verifies in the same situation that $\sigma \to 0$ that:

$$KL(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) \to 0.$$

In that case, let $g(\mathbf{x}) = (G^{-1}(F_1(\mathbf{x}_1)), \dots, G^{-1}(F_N(\mathbf{x}_N)))$, note that $g = f^{-1}$. Also, let's take

$$q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|g(\mathbf{x}), J_g(\mathbf{x})(J_g(\mathbf{x}))^T\sigma^2).$$

Let's also consider the change of variable $\mathbf{z} = g(\mathbf{x}) + (\mathbf{z}^0 - g(\mathbf{x}))\sigma$. Then, we have that $\mathbf{z}^0 = (\mathbf{z} - g(\mathbf{x}))\frac{1}{\sigma} + g(\mathbf{x})$, and with it, applying the change of variable theorem:

$$\begin{aligned} p^0(\mathbf{z}^0|\mathbf{x}) &= p(\mathbf{z} = g(\mathbf{x}) + (\mathbf{z}^0 - g(\mathbf{x}))\sigma|\mathbf{x})\sigma^N, \\ q^0(\mathbf{z}^0|\mathbf{x}) &= \mathcal{N}(\mathbf{z}^0|g(\mathbf{x}), J_g(\mathbf{x})(J_g(\mathbf{x}))^T). \end{aligned}$$

Since it is immediate from its definition that the KL-divergence is an operator invariant to affine transformations in the sample space, it follows that:

$$KL(q(\mathbf{z}|\mathbf{x})||p(\mathbf{z}|\mathbf{x})) = KL(p^0(\mathbf{z}^0|\mathbf{x})||q^0(\mathbf{z}^0|\mathbf{x})).$$

Now, $q^0(\mathbf{z}^0|\mathbf{x})$ does not depend on $\sigma$ and its covariance matrix is well-defined. It is therefore sufficient to show that $p^0(\mathbf{z}^0|\mathbf{x}) \to q^0(\mathbf{z}^0|\mathbf{x})$ for all $\mathbf{z}^0$, in which case the desired result will be obtained.

If we take $r = g(\mathbf{x}) + (\mathbf{z}^0 - g(\mathbf{x}))\sigma$, then:

$$\begin{aligned} p^0(\mathbf{z}^0|\mathbf{x}) = p(\mathbf{z} = r|\mathbf{x})\sigma^N &= \frac{p(\mathbf{x} = x|\mathbf{z} = r)p(\mathbf{z} = r)\sigma^N}{p(\mathbf{x} = x)} \\ &= \frac{\mathcal{N}(\mathbf{x}|f(r), \sigma^2\mathbf{I})p(\mathbf{z} = r)\sigma^N}{p(\mathbf{x} = x)} \\ &= C\mathcal{N}(x; f(g(\mathbf{x}) + (\mathbf{z}^0 - g(\mathbf{x}))\sigma), \sigma^2\mathbf{I})). \end{aligned}$$

As $\sigma \to 0$, we have that $p(\mathbf{x}) \to p^*(\mathbf{x})$ and that $p(\mathbf{z} = r) \to p(g(\mathbf{x}))$, so both are constants with respect to the latent variables for sufficiently small $\sigma$. All of this, together with

$\sigma^N$, form part of a normalizing constant $C$ that ensures the expression is a probability density over $\mathbf{z}^0$ when integrating over the sample space.

Thanks to the hypothesis, it is possible to apply Taylor's Theorem to $f$ around the point $g(\mathbf{x})$, so that there will exist a $\xi$ in the segment joining $g(\mathbf{x})$ and $g(\mathbf{x}) + (\mathbf{z}^0 - g(\mathbf{x}))\sigma$ such that:

$$f(g(\mathbf{x}) + (\mathbf{z}^0 - g(\mathbf{x}))\sigma) = f(g(\mathbf{x})) + J_f(g(\mathbf{x}))(\mathbf{z}^0 - g(\mathbf{z}))\sigma + \frac{1}{2}((\mathbf{z}^0 - g(\mathbf{x}))\sigma)^T H_f(\xi)((\mathbf{z}^0 - g(\mathbf{x}))\sigma).$$

Remember that $f^{-1} = g$, so $f(g(\mathbf{x})) = \mathbf{x}$ and therefore:

$$p^0(\mathbf{z}^0|\mathbf{x}) = C\mathcal{N}(\mathbf{x}; \mathbf{x} + J_f(g(\mathbf{x}))(\mathbf{z}^0 - g(\mathbf{x}))\sigma + \frac{1}{2}((\mathbf{z}^0 - g(\mathbf{x}))\sigma)^T H_f(\xi)((\mathbf{z}^0 - g(\mathbf{x}))\sigma), \sigma^2\mathbf{I}).$$

Furthermore:

$$\mathcal{N}(\mathbf{x}; \mu, \Sigma) = (2\pi)^{-\frac{N}{2}} \det(\Sigma)^{-\frac{1}{2}} \exp\left\{\frac{-1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}.$$

Then, if we take:

$$\mu(\mathbf{x}, \mathbf{z^0}, \sigma) = g(\mathbf{x}) - \frac{\sigma^2}{\sigma} J_f(g(\mathbf{x}))^{-1}(\mathbf{z}^0 - g(\mathbf{x}))^T H_f(\xi)(\mathbf{z}^0 - g(\mathbf{x})),$$

it is possible to expand the expression contained in the exponential of the normal as:

$$-\frac{1}{2}\left(\mathbf{z}^0 - \mu(\mathbf{x}, \mathbf{z^0}, \sigma)\right)^T (J_f(g(\mathbf{x})))^T J_f(g(\mathbf{x}))\left(\mathbf{z}^0 - \mu(\mathbf{x}, \mathbf{z^0}, \sigma)\right),$$

so that:

$$p^0(\mathbf{z}^0 \mid \mathbf{x}) = \frac{C\sqrt{\det\left((J_f(g(\mathbf{x}))^{-1}(J_f(g(\mathbf{x}))^{-1})^T\right)}}{\sigma^N}\mathcal{N}\left(\mathbf{z}^0;\ \mu(\mathbf{x}, \mathbf{z^0}, \sigma), J_f(g(\mathbf{x}))^{-1}(J_f(g(\mathbf{x}))^{-1})^T\right).$$

Here, being a probability distribution over $\mathbf{z}^0$, the only possibility is that:

$$C = \frac{\sigma^N}{\sqrt{\det\left((J_f(g(\mathbf{x}))^{-1}(J_f(g(\mathbf{x}))^{-1})^T\right)}}.$$

On the other hand, note that the elements of $H_f(\xi)$ and $J_f(g(\mathbf{x}))^{-1}$ will be bounded by hypothesis, since by the construction of $f$ they will be functions that factorize into partial derivatives of the conditioned densities of $p^*$ and of the standard normal density that had $G$ as its distribution function. Therefore, if we make $\sigma \to 0$ and apply the Inverse Function Theorem to $f$ and $g$, then:

$$p^0(\mathbf{z}^0|\mathbf{x}) \to \mathcal{N}(\mathbf{z}^0; g(\mathbf{x}), J_g(\mathbf{x})(J_g(\mathbf{x}))^T) = q^0(\mathbf{z}^0|\mathbf{x}),$$

which concludes the proof.

■

# 5 Practical Applications of the VAE

This chapter aims to show that the variational autoencoder is not only a model of great theoretical and statistical interest, but also a versatile tool with multiple practical applications. Unlike other approaches in artificial intelligence, the VAE combines the approximation capacity of neural networks with a probabilistic structure that makes it interpretable.

Its latent space, if well trained, is smooth, continuous and semantically meaningful. This means that it captures high-level features—such as a smile, the slant of a handwritten digit or the presence of a tumor in a medical image—in a structured and manipulable way. These properties make it an ideal model for generative tasks, noise reduction or anomaly detection.

The implementation of the model has been done from scratch in Python using the *PyTorch* library. This platform, developed by Meta AI, has established itself as one of the most widely used in research due to its flexibility, performance on GPU and its ecosystem of tools for neural networks, optimization and visualization.

A standard version of the VAE has been adopted with a slight modification. The variational density used is a factorized multivariate normal (see 4.2), whose KL-divergence can be calculated analytically (see 4.2). Then, to avoid the so-called *posterior collapse*, where the function $q(\mathbf{z}|\mathbf{x})$ prematurely converges to the prior $p(\mathbf{z})$, a parameter $\beta > 0$ has been introduced that controls the weight of the KL-divergence in the ELBO. This parameter is progressively increased during training, following the scheme known as $\beta$-VAE [7].

The source code of the model, along with its training and auxiliary functions, as well as the different applications, are available in Appendices B and C. For optimization, the Adam (*Adaptive Moment Estimation*) algorithm has been used, widely adopted in deep learning for its efficiency and stability. Like gradient descent, it uses gradient-based optimization techniques, but with the use of weighted moving averages. Interestingly, this optimizer was proposed by one of the original authors of the VAE in [10].

The training has been carried out entirely on an Nvidia GPU, which has allowed the process to be considerably accelerated compared to the exclusive use of a CPU.

## 5.1 Generative Modeling and Latent Space Exploration

The *MNIST* dataset [13] is one of the most used in the field of machine learning. Over the years, it has become the reference test to validate the performance of new statistical models and neural network architectures.

This set consists of grayscale images of handwritten digits, with a resolution of $28 \times 28$ pixels. In this experiment, a VAE is trained whose latent space is restricted to $\mathbb{R}^2$, with the aim of directly visualizing the organization of the latent space learned among the different classes of digits.

Despite having some 60,000 training images, the simplicity of the samples requires

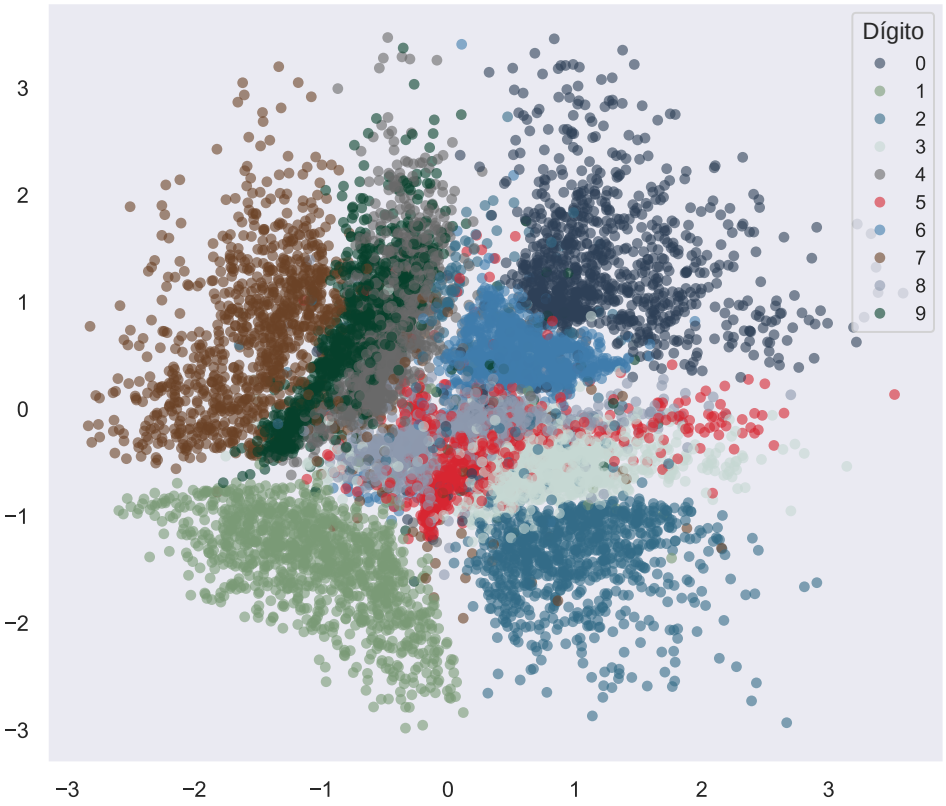



Figure 5.1: Two-dimensional latent space built by the VAE trained with MNIST.

caution when defining the model's capacity. A VAE that is too expressive could ignore the latent structure and simply copy the inputs to the output, without learning useful or semantically meaningful representations.

Figure 5.1 shows the distribution of the digits in the learned latent space. It can be observed that some classes partially overlap, especially those that correspond to digits with similar shapes. For example, the regions corresponding to digits 4 and 9 appear contiguous, reflecting their similarity in terms of strokes and visual structure.

Once the model has been trained and the latent structure in $\mathbb{R}^2$ has been visualized, new samples have been generated directly from said space. To do this, a regular grid of points has been built on the latent plane and each of its vectors $\mathbf{z}$ has been propagated exclusively through the already trained *decoder*, that is, through the generative model $p(\mathbf{x}|\mathbf{z})$.

The result can be seen in Figure 5.2, which shows the decoding of the complete grid and the new samples generated. This visualization allows us to clearly observe how the model interpolates between different classes of digits and how the generated shape varies depending on the latent position. The smooth transitions between regions illustrate that the learned latent space is not only continuous, but also semantically coherent, capturing progressive variations in the style and shape of the digits.

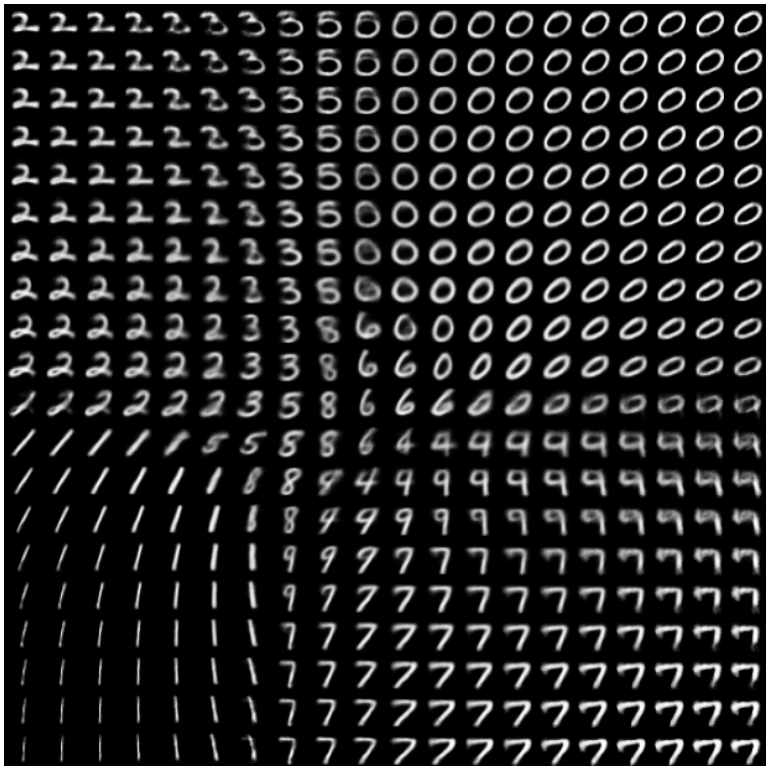
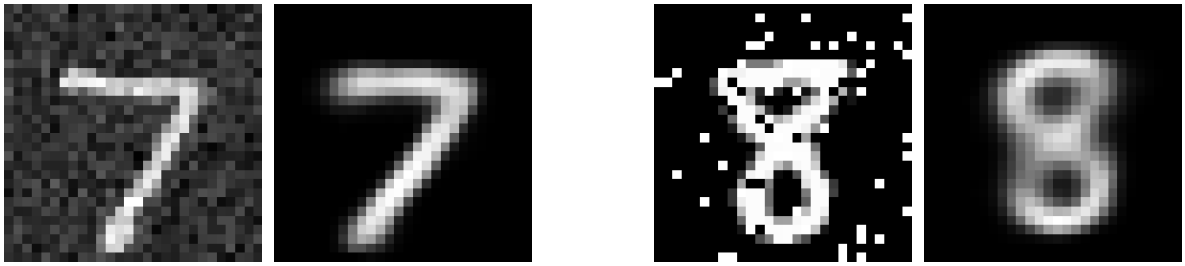


Figure 5.2: Decoding of a regular grid in the latent plane



(a) (b)

Figure 5.3: Gaussian noise reduction (a) and image reconstruction (b)

## 5.2 Noise Reduction and Image Reconstruction

Traditional autoencoders are useful for tasks such as noise removal or reconstruction of incomplete images, but only when they have been explicitly trained with corrupted data. In contrast, variational autoencoders stand out for their ability to perform these tasks even without specific training, thanks to the probabilistic and regularized structure of their latent space.

This is because the VAE learns a coherent and robust latent representation, capable of generating plausible reconstructions despite perturbations or loss of information in the input data. As can be seen in Figure 5.3, the model manages to reconstruct digits from noisy and incomplete versions, demonstrating its ability to naturally infer missing information.

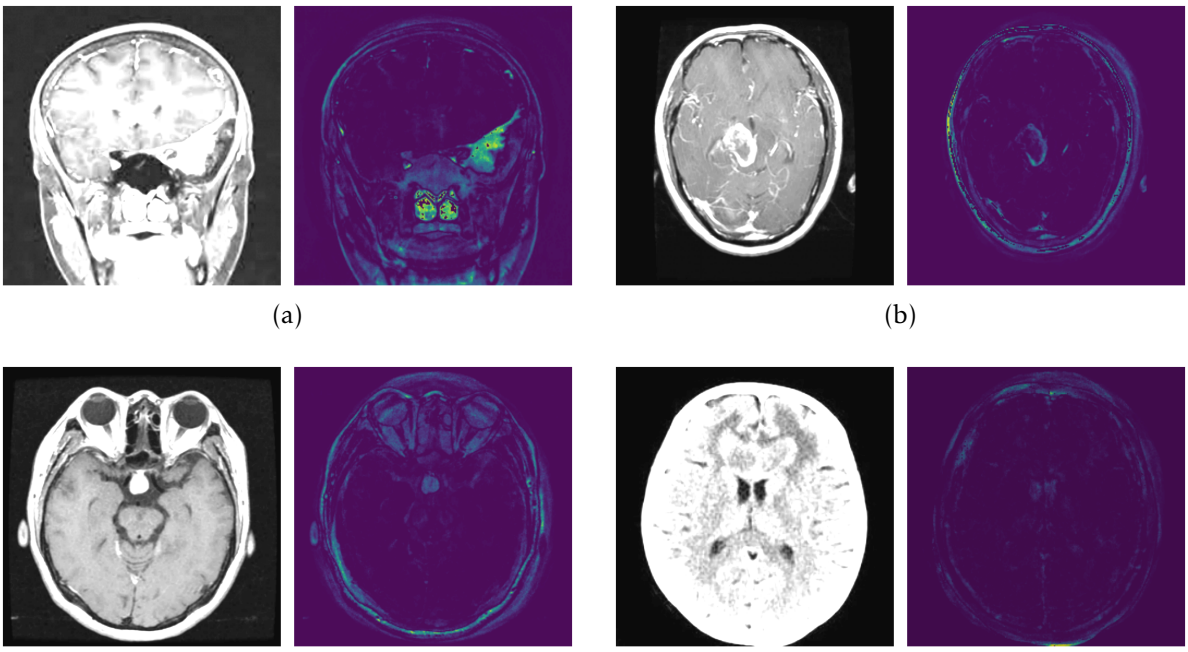



Figure 5.4: Heat maps of the VAE reconstruction error applied to brain magnetic resonances. In (a), a meningioma is clearly reflected by a high reconstruction error in the right white matter. In (b) and (c), corresponding to glioma and pituitary gland cancer respectively, the model also fails in subtle areas affected by the disease, thus leaving them exposed. In (d), a healthy brain is shown, where the reconstruction error is practically imperceptible, with an almost off heat map. It should be noted that the model also makes errors in the cerebral cortex due to the high contrast with the black background of the resonances and the small amount of data with which it has been trained.

## 5.3 Anomaly Detection

Next, we will proceed to the detection of anomalies in the form of tumors using brain scans. To perform this task, a dataset of magnetic resonance images of healthy brains and brains affected by different types of brain cancer has been used, both taken from different angles [16].

In this case, having an unsupervised feature model like the VAE, it will be trained with 1,900 images of healthy brains, with the aim that the model learns to reconstruct them and that its latent space represents what a statistically healthy brain is like.

Later, the model is tested with images of affected and healthy patients. As expected, the VAE fails to reconstruct the tumor regions, since its latent space does not include the necessary information for this, having not seen them during training. From the reconstruction error, a heat map is generated that allows the affected areas to be visually located. The results are presented in Figure 5.4.

Figure 5.5: CelebA training examples.

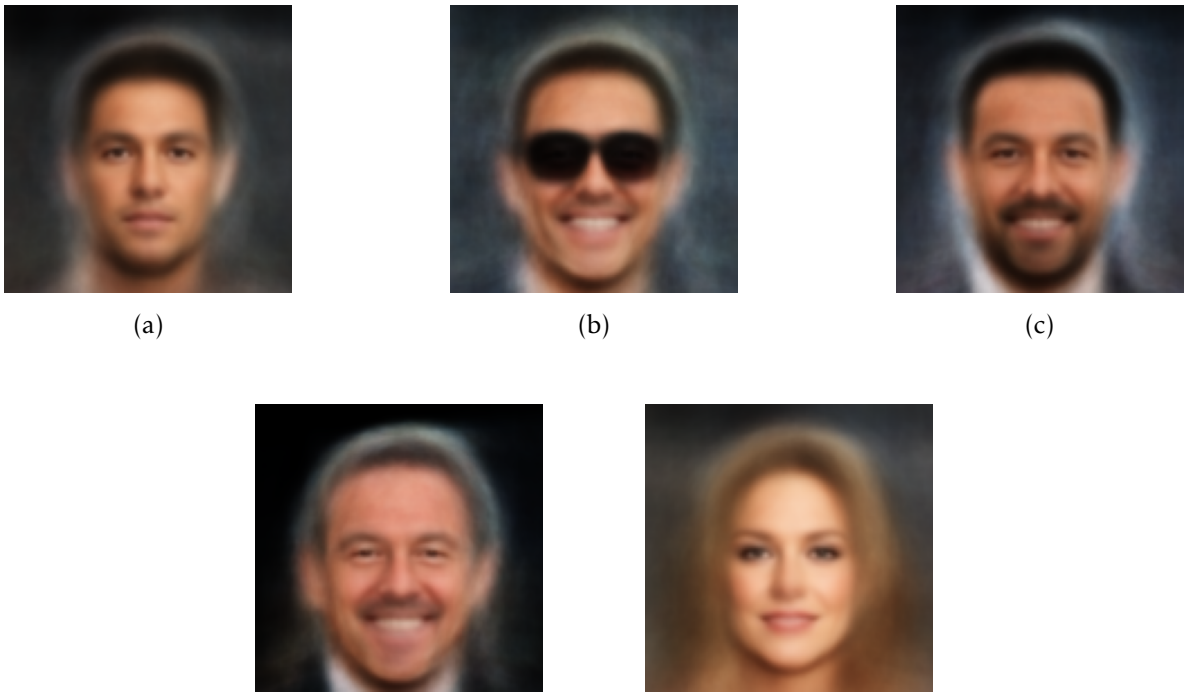
(a) (b)

Figure 5.6: Image from the test set (a) and its reconstructed version by the trained VAE (b).

## 5.4 Latent Manipulation

The *Variational Autoencoder* (VAE) allows learning structured latent representations of data and manipulating them to induce semantic modifications in the samples. However, the simplicity of its variational density—a factorized multivariate normal—limits the fidelity of reconstructions, which are often blurry. For this reason, diffusion models currently lead in generative image modeling. Even so, VAEs continue to be useful tools, especially when combined with more complex models, such as diffusion models, to control high-level attributes such as facial expression, hair color or age.

In this application, a VAE has been trained on the *CelebA* dataset [14], with some 200,000 images of celebrity faces, of which 1,000 were reserved for evaluation. The training was carried out for 2 hours and 50 minutes on a GPU, after resizing the images to $128 \times 128$ pixels. Figure 5.6 shows a typical reconstruction example, with the expected loss of sharpness. The main interest lies in the learned latent space $\mathbb{R}^{256}$: using the available labels (such as «smiling» or «with beard»), semantic direction vectors



(a) (b) (c)

(d) (e)

Figure 5.7: Manipulation of facial attributes by interpolation in the latent space. In (a), the smile of the original sample from Figure 5.6b is removed. Next, in (b), a more pronounced smile has been added along with sunglasses. Image (c) shows the incorporation of a beard. In cases (d) and (e), more general characteristics are modified: in (d), the face is aged, while in (e) it is transformed into a female blond version with a slight smile. These results illustrate how the VAE allows modifying high-level semantic attributes from linear operations in the latent space. See https://huggingface.co/spaces/angelramos/vae-celeba-manipulation to perform modifications like the previous ones interactively.

can be estimated and by a simple controlled translation in a single direction of the 256-dimensional space, semantic aspects can be changed, which in this case correspond to facial details.

For example, by calculating the difference between the latent means of smiling and non-smiling images, a vector $\mathbf{z}_{\text{sonrisa}}$ is obtained. Given the latent representation $\mathbf{z}$ of an image that has passed through the *encoder*, if a controlled translation $\mathbf{z}^* = \mathbf{z} + \alpha\mathbf{z}_{\text{sonrisa}}$ is applied, it allows adding ($\alpha > 0$) or removing ($\alpha < 0$) its smile. By modifying the latent encoding and computing the result through the *decoder*, the desired result will be obtained applied to the image. Figure 5.7 shows some modifications applied to the evaluation example shown previously in Figure 5.6.

This application shows that, despite its generative limitations in high-complexity tasks, a VAE can modify semantic attributes of images through simple linear operations in its latent space, which, together with its efficiency and probabilistic basis, makes it a useful tool for controlled facial editing.

# *Conclusion*

Throughout this document, a long journey has been taken, starting from the foundations of Bayesian inference and variational inference, culminating in the construction, analysis, and implementation of the Variational Autoencoder, one of the most representative models of modern generative learning. Far from being limited to a superficial description, the work has sought to delve with mathematical rigor into both the theoretical aspects that support these models and their practical applications.

The first part focused on building a solid foundation, addressing key concepts such as neural networks, variational inference, and the role of probabilistic graphical models. This knowledge served as a foundation for understanding how a posterior distribution can be efficiently approximated using the ELBO and techniques such as the CAVI algorithm, even in scenarios where the classic Bayesian approach provided by Markov Chain Monte Carlo (MCMC) is intractable.

On this basis, the Variational Autoencoder was introduced as a model that combines deep neural networks with approximate Bayesian inference. Special attention was paid to its formulation, the implications of the reparameterization trick, and the role of the latent space. As a complement, an approximation theorem was presented that highlights its expressive capacity in complex contexts.

The last part of the work was practical in nature. The model was implemented and its capabilities were tested in tasks such as data generation, noise reduction, and anomaly detection. These applications served to empirically validate the potential of the VAE and to reinforce the intuitive understanding of the concepts developed in the theoretical part.

In short, this work has not only served to study a specific model in depth, but has also allowed to demonstrate how different branches of mathematics—probability, optimization, statistics, and deep learning—can be coherently united to solve highly complex real problems that today enjoy growing interest in the academic and professional fields.

# *Bibliography*


[1] C. M. Bishop, H. Bishop. *Deep Learning: Foundations and Concepts*. Springer, 2023.

[2] C.M. Bishop. *Pattern recognition and Machine Learning*. Springer, 2006.

[3] D. M. Blei, A. Kucukelbir, J. D. McAuliffe. *Variational Inference: A review for statisticians*. Journal of the American Statistical Association, 112(518), 859–877, 2017.

[4] G. Cybenko. *Approximation by superpositions of a sigmoidal function*. Mathematics of Control, Signals and Systems. Springer, 1989.

[5] C. Doersch. *Tutorial on Variational Autoencoders*. arXiv preprint arXiv:1606.05908, 2016.

[6] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. MIT Press, 2016.

[7] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, A. Lerchner. *β-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*. International Conference on Learning Representations (ICLR), 2017.

[8] K. Hornik, M. Stinchcombe, H White. *Multilayer feedforward networks are universal approximators*.Neural Networks, Vol. 2, pp. 359-366. Pergamon Press plc, 1989.

[9] M. I. Jordan, Z. Ghahramani, T.S. Jaakkola, L. K. Saul. *An Introduction to Variational Methods for Graphical Models.* Kluwer Academic Publishers, 1999.

[10] D. P. Kingma, J. Ba. *Adam: A Method for Stochastic Optimization*. 3rd International Conference on Learning Representations (ICLR), 2015.

[11] D. P. Kingma. M. Welling. *Auto-Encoding Variational Bayes*. 2nd International Conference on Learning Representations (ICLR), 2014.

[12] D. P. Kingma. M. Welling. *An Introduction to Variational Autoencoders*. Foundations and Trends in Machine Learning 12(4), 307–392, 2019.

[13] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11), 2278–2324, 1998.

[14] Z. Liu, P. Luo, X. Wang, X. Tang. *Deep Learning Face Attributes in the Wild*. En Proceedings of the IEEE International Conference on Computer Vision (ICCV) (pp. 3730–3738), 2015.

[15] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[16] M. Nickparvar. *Brain Tumor MRI Dataset (version 2)*. `https://doi.org/10.34740/KAGGLE/DSV/2645886` DOI:10.34740/KAGGLE/DSV/2645886, 2024.

[17] J. Paisley, D. M. Blei, M. I. Jordan. *Variational Bayesian Inference with Stochastic Search*. Proceedings of the 29th International Conference on Machine Learning (ICML) (pp. 1363–1370), 2012.

[18] J. Pearl. *Probabilistic reasoning in intelligent systems*. Morgan-Kaufman, 1988.

[19] H. Robbins, S. Monro. *A Stochastic Approximation Method*. The Annal of Mathematical Statistics 22(3), 400–407, 1951.

[20] M. Rosenblatt. *Remarks on a Multivariate Transformation*. The Annals of Mathematical Statistics, 23(3), 470-472, 1952.

[21] R.Y. Rubinstein, D.P. Kroese. *Simulation and the Monte Carlo method*. 3ª edición. Wiley, 2017.

[22] TikZ contributors. TikZ.net - TikZ diagrams for LaTeX. `https://tikz.net/` .

# Appendices

# A

# CAVI Algorithm Implementation

This Appendix shows the Python code used for the practical implementation of CAVI in Section 2.4.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
import seaborn as sns

def initialize_parameters(n_samples,n_centers):
  """
  Initializes the model parameters given the sample size and the number of clusters.

  Parameters
  ----------
  n_samples: int
    Sample size.

  n_centers: int
    Number of clusters.

  Returns
  --------
  m: numpy.array
    Matrix formed by the variational mean vectors.

  s2: numpy.array
    Vector of variational variances for each cluster.

  phi: numpy.array
    Variational phi vector of probabilities.
  """
  n = n_samples
  k = n_centers

  m = np.random.randn(k,2)*2
  s2 = np.ones(k)*10 # these are variances
  phi = np.ones((n,k))*1/k

  return m, s2, phi


def compute_elbo(X, m, s2, phi, sigma2):
  """
  Computes the ELBO of the model.

  Parameters
  ----------
  X: numpy.array
    Input data matrix (n_samples, n_features).

  m: numpy.array
    Matrix of variational means.

  s2: numpy.array
    Vector of variational variances.

  phi: numpy.array
    Variational phi matrix with responsibilities.

  sigma2: float
    Variance of the generative model.

  Returns
  -------
```

```
  elbo: float
    The value of the ELBO bound.
  """
  n,k = phi.shape

  term1 = - k * np.log(2*np.pi*sigma2) - 1/(2*sigma2)*(np.sum(m**2) + 2*np.sum(s2))
  term2 = -n*np.log(k)


  term3 = 0
  for i in range(n):
      for j in range(k):
          diff = X[i] - m[j]
          term3 += phi[i, j] * (-0.5 * np.log(2 * np.pi) - 0.5 * np.sum(diff**2) -
              s2[j])

  term4 = -np.sum(phi*np.log(phi))
  term5 = -np.sum(np.log(2*np.pi*np.e*s2))

  elbo = term1 + term2 + term3 + term4 + term5

  return elbo

def update_phi(X,m,s2,phi):
    """
    Updates the phi matrix of variational responsibilities.

    Parameters
    ----------
    X: numpy.array
      Input data (n_samples x n_features).

    m: numpy.array
      Current variational means.

    s2: numpy.array
      Current variational variances.

    phi: numpy.array
      Current responsibility matrix.

    Returns
    -------
    phi: numpy.array
      Updated responsibility matrix.
    """

    n, _ = X.shape
    k = m.shape[0]
    phi = np.zeros((n, k))

    for i in range(n):
        for j in range(k):
            diff = X[i,:] - m[j,:]                      # (d,)
            sq_norm = np.dot(diff, diff)           # ||x_i - m_k||^2
            var_term = s2[j]               # s_{k}^2
            phi[i, j] = np.exp(-0.5 * sq_norm - 0.5 * var_term)

        # Normalize
        phi[i, :] /= np.sum(phi[i, :])

    return phi


def update_centers(X,phi,sigma2,m,s2):
  """
  Updates the variational means m and variances s2 of the model.

```

```
  Parameters
  ----------
  X: numpy.array
    Input data (n_samples x n_features).

  phi: numpy.array
    Matrix of variational responsibilities.

  sigma2: float
    Variance of the generative model.

  m: numpy.array
    Current means.

  s2: numpy.array
    Current variances.

  Returns
  -------
  m: numpy.array
    Updated means.

  s2: numpy.array
    Updated variances.
  """
  n = X.shape[1]
  k = m.shape[0]

  for j in range(k):
    m[j,:] = np.dot(phi[:, j], X)/(1/sigma2 + np.sum(phi[:,j]))
    s2[j] = 1/(1/sigma2 + np.sum(phi[:,j]))

  return m,s2


def cavi(X,n_centers,sigma2, max_iter = 50, plot_start = True):
  """
  Runs the CAVI (Coordinate Ascent Variational Inference) algorithm.

  Parameters
  ----------
  X: numpy.array
    Input data (n_samples x n_features).

  n_centers: int
    Number of clusters.

  sigma2: float
    Variance of the generative model.

  max_iter: int
    Maximum number of iterations.

  plot_start: bool
    If True, it shows a plot with the initial centers. Default: True.

  Returns
  -------
  phi: numpy.array
    Matrix of variational responsibilities.

  m: numpy.array
    Learned variational means.

  s2: numpy.array
    Learned variational variances.

  elbo_values: list
```

```
    ELBO values during the iterations.
  """

  n = X.shape[0]
  k = n_centers

  m, s2 , phi = initialize_parameters(n,k)

  if plot_start:
    fig, ax = plt.subplots(figsize = (8,6))
    scatter = ax.scatter(X[:,0], X[:, 1], c = y, cmap='tab10', edgecolor = 'k', s = 60, alpha = 0.8)
    scatter = ax.scatter(m[:,0], m[:,1], c = 'yellow', edgecolor = 'k', s = 100, alpha = 1)
    plt.savefig("inicio.pdf", format="pdf", dpi=300)
    plt.show()
    print("ELBO MONITORING")

  elbo_values = []
  elbo_values.append(compute_elbo(X, m, s2, phi, sigma2))

  for i in range(max_iter):
    phi = update_phi(X,m,s2,phi)
    m, s2 = update_centers(X,phi,sigma2,m,s2)
    elbo_values.append(compute_elbo(X , m , s2, phi, sigma2))

    if i % 5 == 0 and plot_start:
      print(f"ELBO value at iteration {i}: {elbo_values[-1]}")  # ELBO monitoring

  return phi, m, s2, elbo_values


"""Application"""
# Initial data definition
np.random.seed(123)
X, y = make_blobs(n_samples=500, n_features=2, centers=4, cluster_std= 1, shuffle= True)


sns.set(style = "darkgrid", context = "notebook")
fig, ax = plt.subplots(figsize = (8,6))
scatter = ax.scatter(X[:,0], X[:, 1], c = y, cmap='tab10', edgecolor = 'k', s = 60, alpha = 0.8)
ax.set_title("Four Normally Distributed Clusters")

plt.show()

#CAVI
phi, m, s2, elbo_values = cavi(X, n_centers=4, sigma2=1, max_iter=50)


# ELBO monitoring
plt.figure(figsize=(8, 5))
plt.xlabel("Iteration", fontsize = 19)
plt.ylabel("ELBO")
plt.grid(True)
plt.legend()
plt.tight_layout()

elbos = np.zeros((10, 51))


for i in range(elbos.shape[0]):
  _,_,_, elbo_values = cavi(X, n_centers=4, sigma2=1, max_iter=50, plot_start=False)
  plt.plot(elbo_values, label="ELBO", color='royalblue')

plt.savefig("elbo.pdf", format = 'pdf', dpi = 300)
plt.show()
```

```
261
262 # Final result
263 fig, ax = plt.subplots(figsize = (8,6))
264 scatter = ax.scatter(X[:,0], X[:, 1], c = y, cmap='tab10', edgecolor = 'k', s = 60,
        alpha = 0.8)
265 scatter = ax.scatter(m[:,0], m[:,1], c = 'yellow', edgecolor = 'k', s = 100, alpha
        = 1)
266 #plt.savefig("final.pdf", format = 'pdf', dpi = 300)
267
268 plt.show()
```

Listing A.1: CAVI implementation for a Gaussian mixture

# B

# *Model Implementation and Utilities*

In this appendix, all the source code used for the construction of the model and its training scheme is presented. Likewise, the training procedure corresponding to Section 5.4 is included, since, unlike other applications, it has required a more complex and optimized process, capable of taking advantage of both the CPU and the GPU to speed up execution. All functionalities have been implemented within the *PyTorch* ecosystem.

The complete code along with that presented in Appendix C can be consulted in a more structured way in the following repository:

https://github.com/angelr-code/vae-project

The program `model.py` defines the VAE class, allowing the model to be adapted to different scenarios by configuring parameters such as the dimensions of the input layer, hidden layers, latent space, and the activation function of the output layer, all of which are user-specifiable.

```python
import torch
from torch import nn

class Encoder(nn.Module):
    """
    Encoder network for the Variational Autoencoder (VAE).

    Maps the input data into the parameters of a Gaussian distribution
    in the latent space.

    It inherits from the torch nn Module.

    Parameters
    ----------
    input_dim: int
        input dimension of the VAE.

    hidden_dims: int list
        list containing the hidden layers dimensions.

    latent_dim: int
        latent space dimension of the VAE.
    """

    def __init__(self, input_dim, hidden_dims, latent_dim):
        super().__init__()

        layers = []
        in_dim = input_dim
        for h_dim in hidden_dims:
            layers.append(nn.Linear(in_dim, h_dim))
            layers.append(nn.ReLU())
            in_dim = h_dim

        self.encoder = nn.Sequential(*layers)
        self.hidden2mu = nn.Linear(hidden_dims[-1], latent_dim)
        self.hidden2logvar = nn.Linear(hidden_dims[-1], latent_dim)

        # We model the log-variance to ensure a postive variance
        # after applying the exponential function.

    def forward(self, x):
        """
        Forward pass through the encoder.
```

```
45
46          Parameters
47          ----------
48          x : torch.Tensor
49              Input tensor of shape (batch_size, input_dim).
50
51          Returns
52          -------
53          mu : torch.Tensor
54              Mean of the approximate posterior.
55
56          logvar : torch.Tensor
57              Log-variance of the approximate posterior.
58          """
59
60          h = self.encoder(x)
61          mu = self.hidden2mu(h)
62          logvar = self.hidden2logvar(h)
63          return mu, logvar
64
65  class Decoder(nn.Module):
66      """
67      Decoder network for the Variational Autoencoder (VAE).
68
69      Maps the latent space representation into the data reconstruction (output).
70
71      It inherits from the torch nn Module.
72
73      Parameters
74      ----------
75      latent_dim: int
76          latent space dimension.
77
78      hidden_dims: int list
79          list containing the hidden layers dimensions.
80
81      output_dim: int
82          output dimension of the VAE.
83
84      f_out: str
85          Indicates the output layer activation.
86          Must be 'sigmoid' or 'tanh'.
87      """
88
89      def __init__(self, latent_dim, hidden_dims, output_dim, f_out):
90          super().__init__()
91          layers = []
92
93          in_dim = latent_dim
94          for h_dim in reversed(hidden_dims):
95              layers.append(nn.Linear(in_dim, h_dim))
96              layers.append(nn.ReLU())
97              in_dim = h_dim
98
99          self.decoder = nn.Sequential(*layers)
100         self.hidden2out = nn.Linear(hidden_dims[0], output_dim)
101
102         #If activation value is unexpected we raise an error.
103         if f_out != 'sigmoid' and f_out != 'tanh':
104             raise ValueError("The output activation must be either 'sigmoid' or
                    'tanh'")
105
106         self.f_out = f_out
107
108
109     def forward(self, z):
110         """
111         Forward pass through the decoder.
```

```
112
113         Parameters
114         ----------
115         z : torch.Tensor
116             Input tensor of shape (batch_size, latent_dim).
117
118         Returns
119         -------
120         x_hat : torch.Tensor
121             Output tensor.
122         """
123
124         h = self.decoder(z)
125
126         if self.f_out == 'sigmoid':
127             x_hat = torch.sigmoid(self.hidden2out(h))
128         else:
129             x_hat = torch.tanh(self.hidden2out(h))
130
131         return x_hat
132
133
134 class VAE(nn.Module):
135     """
136     Variational Autoencoder (VAE) Architecture.
137     The approximate posterior is a factorized multidimensional Gaussian
            distribution.
138
139     Maps the input data into its reconstruction. It combines the Encoder
140     and Decoder Modules.
141
142     It inherits from the torch nn Module.
143
144     Parameters
145     ----------
146     input_dim: int
147         input dimension of the VAE
148
149     hidden_dims: int list
150         list containing the hidden layers dimensions
151
152     latent_dim: int
153         latent space dimension
154
155     f_out: str
156         Indicates the output layer activation.
157         Must be 'sigmoid' or 'tanh'.
158     """
159
160     def __init__(self, input_dim, hidden_dims, latent_dim, f_out):
161         super().__init__()
162         self.encoder = Encoder(input_dim, hidden_dims, latent_dim)
163         self.decoder = Decoder(latent_dim, hidden_dims, input_dim, f_out) # In the
                VAE the input dimension matches the output dimension
164
165     def reparameterize(self, mu, logvar):
166         """
167         Samples from the latent space using the reparametrization trick.
168
169         Parameters
170         ----------
171         mu : torch.Tensor
172             Mean of the approximate posterior.
173
174         logvar : torch.Tensor
175             Log-variance of the approximate posterior.
176
177         Returns
```

```
        -------
        torch.Tensor
            Latent space samples.

        """

        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        """
        Forward pass through the Variational Autoencoder (VAE).

        Parameters
        ----------
        x : torch.Tensor
            Input tensor of shape (batch_size, input_dim).

        Returns
        -------
        x_hat : torch.Tensor
            Data reconstruction (output).

        mu : torch.Tensor
            Mean of the approximate posterior.

        logvar : torch.Tensor
            Log-variance of the approximate posterior.
        """

        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        x_hat = self.decoder(z)
        return x_hat, mu, logvar
```

Listing B.1: VAE model class definition

Next, the code for `train.py` is shown, where the negative ELBO is defined as the loss function to be minimized. In turn, the main function responsible for training the model is included.

```
import torch
from torch.nn import functional as F



def loss(x, x_hat, mu, logvar, beta, input_dim, f_out):
    """
    Computes the Variational Autoencoder (VAE) negative ELBO given datapoints and
        their reconstruction.
    Thus, minimizing this expresion will be equivalent to maximizing the ELBO.

    Computes both the KL divergence KL[q(z|x) || p(z)] and the reconstruction loss
        and then adds
    them up to get the ELBO.

    Parameters
    ----------
        x : torch.Tensor
            Input tensor of shape (batch_size, input_dim).

        x_hat: torch.Tensor
            VAE x reconstruction (output) (bath_size, input_dim).

        mu: torch.Tensor
            Mean of the approximate posterior (batch_size, latent dim).
```

```
24
25          logvar: torch.Tensor
26              Log-variance of the approximate posterior (batch_size, latent dim).
27
28          beta: float
29              Used to control the influence of KL divergence in the loss function
                    expression.
30
31          input_dim: int
32              Dimension of the flattened image tensors.
33
34          f_out: str
35              Indicates the output layer activation.
36              Must be 'sigmoid' or 'tanh'.
37
38      Returns
39      -------
40          -ELBO: torch.Tensor
41      """
42
43
44      KL = - 0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())     # KL[q(z|x)
            || p(z)]
45
46
47      # Depending on the output layer activation we choose the suitable
            reconstruction error
48      if f_out == 'sigmoid':
49          reconstruction_error = F.binary_cross_entropy(x_hat, x.view(-1, input_dim),
                reduction='sum') # Reconstruction Loss. Binary Cross Entropy
50      else:
51          reconstruction_error = F.mse_loss(x_hat, x.view(-1, input_dim),
                reduction='sum') # Reconstruction Loss. Mean Squared Error
52
53
54      return beta*KL + reconstruction_error, KL, reconstruction_error
55
56
57
58  def train(model, dataloader, optimizer, device, f_out, max_beta = 4, epochs = 20,
        print_loss = True, labels = False):
59      """
60      Trains the Variational Autoencoder (VAE) maximizing the ELBO.
61
62      Parameters
63      ----------
64
65          model: nn.Module
66              VAE model to be trained.
67
68          dataloader: torch.utils.data.DataLoader
69              DataLoader which loads the training data.
70
71          optimizer: torch.optim
72              Optimization method.
73
74          device: torch.device
75              Training device (GPU or CPU).
76
77          f_out: str
78              Indicates the output layer activation.
79              Must be 'sigmoid' or 'tanh'.
80
81          max_beta: float
82              Maximum beta used in the ELBO. Default is 4.
83
84          epochs: int
85              Number of training epochs. Default is 20.
```

```
86
87          print_loss: bool
88              If True, shows the training loss per epoch. Default: True
89
90          labels: bool
91              If true, the data contains labels that must be ignored. Default: False
92      """
93
94      #If activation value is unexpected we raise an error.
95      if f_out != 'sigmoid' and f_out != 'tanh':
96          raise ValueError("The output activation must be either 'sigmoid' or 'tanh'")
97
98      model.to(device)
99      model.train() # Training mode
100
101     beta = 1/epochs
102
103     for epoch in range(epochs):
104         train_loss = 0.0
105
106         # We iterate through the mini-batches
107         for _, data in enumerate(dataloader):
108
109             if labels:
110                 x, _ = data # In some datasets like MNIST there are labels, we
                        ignore them in the VAE.
111             else:
112                 x = data
113
114             x = x.to(device)
115             optimizer.zero_grad()   # Sets all loss function gradients to zero
116             x_hat, mu, logvar = model(x)
117             loss_value, kl, reconstruction_error = loss(x, x_hat, mu, logvar, beta,
                    input_dim = x.shape[1], f_out = f_out)
118             loss_value.backward() # Loss Backprop. New gradients
119             optimizer.step()    # Updates the model parameters following the chosen
                    optimizer scheme
120             train_loss += loss_value.item() # The loss is in torch.Tensor format,
                    we transform it to float
121
122         beta = min((epoch+1)/epochs, 1.0) * max_beta # Beta parameter annealing
123
124         if print_loss:
125             print(f'Epoch {epoch + 1} -----> -ELBO: {train_loss /
                    len(dataloader.dataset)} | KL: {kl} | Reconstruction Error:
                    {reconstruction_error}')
```

Listing B.2: Training and loss function definition

The following snippet stores the training configurations used for each of the applications.

```
1  # This file contains the configuration for the VAE model based on the task it is
       being used for.
2
3  mnist_configs = {
4          'input_dim': 784,
5          'hidden_dims': [512, 256],
6          'latent_dim': 2
7  }
8
9  brain_configs = {
10         'input_dim': 65536,
11         'hidden_dims': [1024,512,256],
12         'latent_dim': 64
13 }
14
15 celeba_configs = {
```

```
    'input_dim': 49152,
    'hidden_dims': [2048, 1024, 512],
    'latent_dim': 256
}
```

Listing B.3: Training configurations

Next, the training process carried out in Section 5.4 is shown. It differs from the rest in the use of 8 parallel CPU processes to feed the GPU more quickly with data batches for training, since this is the one that performs most of the computation. Finally, the model parameters adjusted at the end of the training are saved in the file `vae_celeba_trained.pth`.

For the rest of the training sessions, data loading by CPU parallelization has not been necessary and the training sessions have been carried out in the same *Jupyter Notebook* in which the application was implemented.

```
import torch
from torch import optim
from pathlib import Path
from model import VAE
from utils import load_celeba
from configs import celeba_configs
from train import train


def main():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    ROOT_DIR = Path(__file__).resolve().parents[1]
    DATA_DIR = ROOT_DIR / "data" / "celeba" / "img_align_celeba"
    celeba_loader = load_celeba(DATA_DIR, num_workers = 8)

    model = VAE(**celeba_configs, f_out='tanh').to(device)
    optimizer = optim.Adam(model.parameters(), lr=3e-4)

    train(model, celeba_loader, optimizer, device, f_out='tanh', epochs=100,
        max_beta = 2)

    torch.save(model.state_dict(), 'vae_celeba_trained.pth')

if __name__ == "__main__":
    main()
```

Listing B.4: Training process for CelebA dataset

Next, `utils.py` is shown, which includes various functionalities and utilities used to perform visualizations and definitions of datasets using *PyTorch* in the applications of Appendix C.

```python
import os
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from torch.utils.data import Dataset, DataLoader
import torchvision
from torchvision import transforms
import torch



##### CelebA #####

class CeleADataset(Dataset):
    """
    Custom celebA dataset for unsupervised learning tasks. It inherits from the
        torch.utils.data Dataset Module.

    Parameters
    ----------

    root_dir: str
        Path to the directory containing the dataset images.

    image_files: str list
        List of image files to be selected. Useful when filtering by attributes.
        If None, gets the entire dataset.
        Default is None.

    transforms: callable, optional
        Transformation methods applied to raw image data (usually a composition
        of torchvision.transforms). Default is None.
    """

    # A torch custom Dataset class must always implement the three functions:
        __init__, __len__, and __getitem__.

    def __init__(self, root_dir, image_files = None, transform = None):
        self.root_dir = root_dir
        self.transform = transform

        if image_files:
            self.image_files = [os.path.join(self.root_dir, f) for f in
                image_files] # Gets the selected files
        else:
            self.image_files = [f for f in os.listdir(root_dir) if
                f.endswith(".jpg")]  # Avoids getting corrupted files.


    def __len__(self):
        return len(self.image_files)


    def __getitem__(self, idx):
        """Gets an Image from the dataset at a given index"""

        img_name = os.path.join(self.root_dir, self.image_files[idx])


        # We open and convert the image to RGB using PIL
        try:
            image = Image.open(img_name).convert("RGB")
```

```
        except Exception as e:
            print(f"Error loading image {img_name}: {e}")
            return None

        # Applies the image the transformation methods if given.
        if self.transform:
            image = self.transform(image)

        image = image.view(-1)  # Flattens the Image

        return image



def load_celeba(root_dir, image_files = None, batch_size = 128, image_size = 128, num_workers = 0):
    """
    Creates the CelebA torch DataLoader.

    Parameters
    ----------
    root_dir: str
        Path to the directory containing the dataset images.

    image_files: str list
        List of image files to be selected. Useful when filtering by attributes.
        If None, gets the entire dataset.
        Default is None.

    batch_size: int
        Size of the training mini-batches. Default is 128.

    image_size: int
        Image size resolution in pixels (img_size x img_size). Default is 128.

    num_workers: int
        Number of paralel subprocesses used for data loading.
        Higher values may speed up data loading but use more CPU resources. Default is 0.

    Returns
    -------
    dataloader: torch.utils.data.DataLoader
        CelebA torch dataloader.
    """

    # Dataset transformations
    transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.CenterCrop(image_size),
        transforms.ToTensor(),
        transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5)) # We normalize the inputs to [-1, 1]. The 3 dimensions are because of the 3 RGB channels.
    ])

    dataset = CeleADataset(root_dir = root_dir, image_files = image_files, transform = transform)

    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers= num_workers,
        pin_memory=True, #True because I will use CUDA.
        drop_last=True # If len(last mini-batch) != batch_size it will be dropped.
    )

    return dataloader
```

```
125
126
127 def denormalize(tensor):
128     """
129     Denormalizes the loaded images for representation purposes.
130
131     Parameters
132     ----------
133     tensor: torch.Tensor
134         Tensor to be denormalized.
135
136     Returns
137     -------
138     torch.Tensor
139         Denormalized tensor.
140     """
141
142     if len(tensor.shape) == 4:
143         result = tensor.clone().detach()
144         result = (result * 0.5 ) + 0.5 # Inverse transformation
145         return result
146     else:
147         # Single image
148         return (tensor * 0.5) + 0.5
149
150
151 def visualize_celeba_examples(dataloader, num_examples, img_size = 64,  fig_size =
        (15,15), download = False):
152     """
153     Function to visualize a small subset of CelebA dataset samples.
154
155     Parameters
156     ----------
157     dataloader: torch.utils.data.DataLoader
158         CelebA dataloader process.
159
160     num_examples: int
161         Number of samples to be shown. Must be smaller than the batch size.
162
163     image_size: int
164         Image size resolution in pixels (img_size x img_size). Default is 64.
165
166     fig_size: tuple
167         Plot size. Default is (15,15).
168
169     download: bool
170         If True downloads the examples to pdf. Default is False.
171     """
172
173     # Gets a minibatch and num_examples elements from it
174     dataiter = iter(dataloader)
175     images = next(dataiter)
176
177     try:
178         images = images[:num_examples]
179     except Exception as e:
180         print(f"The argument num_examples cannot be grater than the batch size.
                Error: {e}")
181
182     images = images.view(num_examples, 3, img_size, img_size) # Unflatens the image
        and gets a matrix for each RGB channel.
183
184     images = denormalize(images)
185
186     nrow = int(np.sqrt(num_examples)) # number of rows in the plot grid
187
188     grid = torchvision.utils.make_grid(images, nrow=nrow, padding=2)
189
```

```
    # We move the grid tensor to the CPU and transform it into a NumPy array
    grid_np = grid.cpu().numpy().transpose((1,2,0))

    plt.figure(figsize=fig_size)
    plt.imshow(grid_np)
    plt.axis('off')

    if download:
        plt.savefig('celeba_examples.pdf', format = 'pdf', bbox_inches = 'tight')

    plt.show()

def image_reconstruction(root, model, device, download = None):
    """
    Given a face image it shows its VAE reconstruction.

    Parameters
    ----------
    root: str
        Image root

    model: torch.nn.Module
        Trained VAE model

    device: torch.device
        Device on which computation is being performed.

    downmload: str
        Given a string downloads the plot in pdf format with that file name.
            Default is None.
    """
    image = Image.open(root).convert("RGB")

    # We apply the same loading transformations.
    transform = transforms.Compose([
        transforms.Resize(128),
        transforms.CenterCrop(128),
        transforms.ToTensor(),
        transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
    ])

    x = transform(image).to(device)

    x = x.view(-1)

    x_hat, _, _ = model(x)

    x_hat = denormalize(x_hat)

    with torch.no_grad():
        x_hat = x_hat.view(3, 128, 128).cpu().numpy()

    x_hat = np.transpose(x_hat, (1,2,0))

    plt.imshow(x_hat)
    plt.axis('off')

    if download:
        plt.savefig(download, format = 'pdf', bbox_inches="tight")

    plt.show()


def latent_interpolation(model, device, directions, img_root, smile, male, blond,
    beard, young, glasses):
    """
    Applies latent space interpolation on a given face image using a trained VAE
        model and learned attribute directions.
```

```
255
256     This function encodes an input image into the latent space using the VAE
            encoder, modifies the latent vector
257     by interpolating along multiple attribute directions (blond hair, smile, male,
            etc), and decodes both the
258     original and modified latent vectors to later produce the corresponding face
            images.
259
260     Parameters
261     ----------
262     model: torch.nn.Module
263         Trained VAE model.
264
265     device: torch.device
266         Device on which computation is being performed.
267
268     directions: dict
269         Dictionary of learned latent directions for each facial attribute.
270
271     img_root: str
272         Path to the input image to be interpolated.
273
274     smile: float
275         Interpolation factor for the smile attribute.
276
277     male: float
278         Interpolation factor for the male attribute.
279
280     blond: float
281         Interpolation factor for the blond attribute.
282
283     beard: float
284         Interpolation factor for the beard attribute.
285
286     young: float
287         Interpolation factor for the young attribute.
288
289     glasses: float
290         Interpolation factor for the glasses attribute.
291
292
293     Returns
294     -------
295     Tuple[np.ndarray, np.ndarray]:
296         - The original reconstructed image as a NumPy array.
297         - The modified image with interpolated attributes as a NumPy array.
298     """
299
300     # Image, transformations and pass through encoder
301     image = Image.open(img_root).convert("RGB")
302
303     transform = transforms.Compose([
304         transforms.Resize(128),
305         transforms.CenterCrop(128),
306         transforms.ToTensor(),
307         transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
308     ])
309     x = transform(image).to(device)
310     x = x.view(-1)
311
312     mu, logvar = model.encoder(x)
313     std = torch.exp(0.5 * logvar).to(device)
314     eps = torch.randn_like(std).to(device)
315     z = mu + eps * std
316
317     # Original reconstruction to numpy
318     z_original = z.clone()
319     x_hat_original = model.decoder(z_original)
```

```
    x_hat_original = x_hat_original.view(3, 128, 128)
    x_hat_original = x_hat_original * 0.5 + 0.5  # denormalize
    original_np = x_hat_original.detach().cpu().numpy().transpose(1, 2, 0)

    #modfied reconstruction
    alpha_values = [smile, male, blond, beard, young, glasses]
    z_mod = z.clone()

    for i, dir_name in enumerate(directions.keys()):
        z_mod = z_mod + alpha_values[i] * directions[dir_name].to(device)

    x_hat = model.decoder(z_mod)
    x_hat = x_hat.view(3, 128, 128)
    x_hat = x_hat * 0.5 + 0.5  # denormalize
    mod_np = x_hat.detach().cpu().numpy().transpose(1, 2, 0)

    return original_np, mod_np

##### Anomaly Detection #####

class BrainTumorDataset(Dataset):
    """
    Brain Tumor MRI Dataset class.

    Parameters
    ----------
    root_dir: str
        Path to the directory containing the dataset images.

    transforms: callable, optional
        Transformation methods applied to raw image data (usually a composition
        of torchvision.transforms). Default is None.
    """

    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_files = [f for f in os.listdir(root_dir) if f.endswith(".jpg")]

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        """Gets an Image from the dataset at a given index"""

        img_name = os.path.join(self.root_dir, self.image_files[idx])


        try:
            image = Image.open(img_name).convert("L") # grayscale
        except Exception as e:
            print(f"Error loading image {img_name}: {e}")
            return None

        if self.transform:
            image = self.transform(image)

        image = image.view(-1)

        return image


def load_brain(root_dir, batch_size = 128, image_size = 256, num_workers = 0):
    """
    Creates the Brain Tumor MRI Dataset torch DataLoader.

    Parameters
    ----------
```

```
    root_dir: str
        Path to the directory containing the dataset images.

    batch_size: int
        Size of the training mini-batches. Default is 128.

    image_size: int
        Image size resolution in pixels (img_size x img_size). Default is 128.

    num_workers: int
        Number of paralel subprocesses used for data loading.
        Higher values may speed up data loading but use more CPU resources. Default
            is 0.

    Returns
    -------
    dataloader: torch.utils.data.DataLoader
        Brain Tumor MRI torch dataloader.
    """

    # Dataset transformations
    transform = transforms.Compose([
        transforms.Resize((image_size, image_size)),
        transforms.Grayscale(num_output_channels=1),
        transforms.GaussianBlur(kernel_size=5, sigma=(0.1,0.2)),    # We apply
            GaussianBlur for better reconstruction at skull borders
        transforms.ToTensor(),
        transforms.Normalize(mean=0.5, std=0.5),    # Applies normalization to set
            all values between [-1, 1]
    ])

    dataset = BrainTumorDataset(root_dir = root_dir, transform = transform)

    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers= num_workers,
        pin_memory=True
    )

    return dataloader

def visualize_brain_examples(dataloader, num_examples,  img_size = 256,  fig_size =
    (15,15)):
    """
    Function to visualize a small subset of Brain Tumor MRI dataset samples.

    Parameters
    ----------
    dataloader: torch.utils.data.DataLoader
        Brain Tumor MRI dataloader process.

    num_examples: int
        Number of samples to be shown. Must be smaller than the batch size.

    image_size: int
        Image size resolution in pixels (img_size x img_size). Default is 128.

    fig_size: tuple
        Plot size. Default is (15,15).
    """

    # Gets a minibatch and num_examples elements from it
    dataiter = iter(dataloader)
    images = next(dataiter)

    try:
```

```
        images = images[:num_examples]
    except Exception as e:
        print(f"The argument num_examples cannot be grater than the batch size. Error: {e}")

    images = images.view(num_examples, 1, img_size, img_size) # Unflatens the image

    images = denormalize(images)

    nrow = int(np.sqrt(num_examples)) # number of rows in the plot grid

    grid = torchvision.utils.make_grid(images, nrow=nrow, padding=2)

    # We move the grid tensor to the CPU and transform it into a NumPy array
    grid_np = grid.cpu().numpy().transpose((1,2,0))

    plt.figure(figsize=fig_size)
    plt.imshow(grid_np, cmap='gray')
    plt.axis('off')
    plt.show()



def visualize_heatmap(image, model, device, threshold, img_size = (256,256), cmap = 'hot', download = None):
    """
    Given an Image and trained model outputs a heatmap of reconstruction error.

    image: PIL.Image.Image
        Brain Image.

    model: torch.nn.Module
        VAE trained model.

    device: torch.device
        Device to be used.

    threshold: float
        Threshold to adjust the heatmap contrast.

    img_size: int tuple
        Indicates the image size by pixels. Default is (256,256)

    cmap: str
        Matplotlib heatmap type. Default is 'hot'

    downmload: str
        Given a string downloads the plot in pdf format with that file name. Default is None.
    """

    transform = transforms.Compose([
        transforms.Resize(img_size),
        transforms.Grayscale(num_output_channels=1),
        transforms.GaussianBlur(kernel_size=5, sigma=(0.1,0.2)),
        transforms.ToTensor(),
        transforms.Normalize(mean=0.5, std=0.5),
    ])

    transform_noblur = transforms.Compose([
        transforms.Resize(img_size),
        transforms.Grayscale(num_output_channels=1),
        transforms.ToTensor(),
        transforms.Normalize(mean=0.5, std=0.5),
    ])

    x = transform(image)
    x = x.to(device)
```

```
    x = x.unsqueeze(0)

    original_tensor = transform_noblur(image).to(device).unsqueeze(0)


    with torch.no_grad():
        output, _, _ = model(x.view(x.size(0), -1))

    x_hat = output.view(x.shape)

    error_map = (original_tensor - x_hat).squeeze().cpu().numpy() ** 2

    image_np = denormalize(original_tensor)
    image_np = image_np.squeeze().cpu().numpy()

    mask = (image_np < threshold).astype(float)
    masked_error = error_map * mask

    background = torch.zeros_like(original_tensor)

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(image_np, cmap='gray')
    plt.axis('off')

    # heatmap
    plt.subplot(1, 2, 2)
    plt.imshow(background.squeeze().cpu().numpy(), cmap='gray')
    plt.imshow(masked_error, cmap=cmap, alpha=0.6)
    plt.axis('off')

    plt.tight_layout()
    if download:
        plt.savefig(download, format = 'pdf', bbox_inches="tight")
    plt.show()


##### MNIST #####

def latent_visualization(z, labels, download = False):
    """
    Prints a VAE 2-D latent space.

    Parameters
    ----------
    z: torch.Tensor
        A torch.Tensor matrix containing the latent encodings
        of datapoints.

    labels: torch.Tensor
        Contains the labels assigned to each datapoint.

    dowload: bool
        If true downloads the plot in a pdf format. Default: False.
    """

    df = pd.DataFrame({
        'z1': z[:,0],
        'z2': z[:,1],
        'label': labels
    })

    #This is a personalized color palette for the different digits latent points
    palette = [
        "#2E4057",
        "#7A9A76",
        "#336B87",
```

```
        "#C6D8D3",
        "#696969",
        "#D72631",
        "#3F7CAC",
        "#6B4226",
        "#8D99AE",
        "#06402B"
    ]

    sns.set_palette(palette)
    sns.set(style="dark")
    plt.figure(figsize=(7, 6))

    sns.scatterplot(
        x='z1', y='z2', hue='label', palette=palette, data=df,
        s=30, alpha=0.6, edgecolor='none', linewidth=0
    )

    plt.xlabel('')
    plt.ylabel('')
    plt.legend(title='Digit', loc='upper right', frameon=True, fontsize='small',
        title_fontsize='large')
    plt.tight_layout()

    if download:
        plt.savefig("latent_space_mnist.pdf", format = "pdf", bbox_inches="tight")

    plt.show()



def print_manifold(model, device, start = -3, end = 4, grid_size = 20, download =
    False):
    """
    Visualizes the latent manifold of a VAE by decoding a regular 2D grid of latent
        space coordinates.

    Parameters
    ----------
    model: torch.nn.Module
        VAE trained model.

    device: torch.device
        device to be used

    start: float
        grid start position. Default is -3

    end: float
        grid end position. Default is 4.

    grid_size: int
        number of images per row and column. Default is 20

    download: bool
        If true downloads the plot in a pdf format. Default: False.
    """

    # Generate the grid
    n = grid_size
    z_values = torch.linspace(start, end, n)

    image_size = 28
    canvas = torch.zeros(image_size * n, image_size * n)

    model.eval()
    with torch.no_grad():
```

```
650         for i, z1 in enumerate(z_values):
651             for j, z2 in enumerate(z_values):
652                 z = torch.tensor([[z1, z2]], dtype=torch.float32).to(device)
653                 x_hat = model.decoder(z).cpu().view(image_size, image_size)
654                 canvas[(n - i - 1)*image_size:(n - i)*image_size,
                        j*image_size:(j+1)*image_size] = x_hat
655 
656     plt.figure(figsize=(8, 8))
657     plt.imshow(canvas, cmap="gray")
658     plt.axis('off')
659 
660     if download:
661         plt.savefig("manifold_mnist.pdf", format = "pdf", bbox_inches="tight")
662 
663     plt.show()
```

Listing B.5: Utility functions for data handling and visualization

# C

# *Practical Applications Implementation*

This appendix includes the code used to apply the VAE to the different applications presented in Chapter 5.

The code is presented in Python *scripts* although it was actually implemented in *Jupyter Notebooks*, which is why it may seem a bit messy and unstructured, although as *scripts* they are completely functional given the dependencies defined in Appendix B. To maintain the aesthetics and clarity of the appendices, it was decided to include the applications as pure Python code. However, in the notebooks folder of the project https://github.com/angelr-code/vae-project the implementations can be seen in a more appropriate way.

Latent exploration, generative modeling, noise reduction and reconstruction of missing parts applied to MNIST

```
import torch
from torch import optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
from pathlib import Path
from src.model import VAE
from src.configs import mnist_configs
from src.train import train
from src.utils import latent_visualization, print_manifold
from skimage.util import random_noise



'''Defining the dataset and training the model'''

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1)) # Flat input
])

ROOT_DIR = Path(__file__).resolve().parents[1]
DATA_DIR = ROOT_DIR / "data"

mnist = datasets.MNIST(root= DATA_DIR,
                       download = False,
                       train = True,
                       transform = transform)

train_loader = DataLoader(mnist,
                          batch_size=128,
                          shuffle=True,
                          pin_memory=True, # This will improve loading and training
                          drop_last=True # Last batch will be dropped if its size
                              is not 64
                          )

model = VAE(**mnist_configs, f_out = 'sigmoid')

optimizer = optim.Adam(model.parameters(), lr= 3e-4)

train(model,train_loader,optimizer,device, max_beta = 4, epochs=50,
    f_out='sigmoid', labels=True)

'''Latent Space visualization'''
```

```
46
47  mnist_test = datasets.MNIST(root = DATA_DIR,
48                              download=False,
49                              train=False,
50                              transform=transform)
51
52  test_loader = DataLoader(mnist_test,
53                           batch_size=64,
54                           shuffle=False,
55                           pin_memory=True,
56                           drop_last=False)
57
58  model.eval()
59  z = []
60  labels = []
61
62  with torch.no_grad():
63      for x, y in test_loader:
64          x = x.to(device)
65          mu, _ = model.encoder(x)
66          z.append(mu.cpu())
67          labels.append(y)
68
69  z = torch.cat(z)
70  labels = torch.cat(labels)
71
72  latent_visualization(z, labels)
73
74  '''Generative Modeling'''
75
76  plt.rcdefaults() # Resets the Matplotlib settings to avoid unexpected results in
        the next plots
77
78  print_manifold(model, device)
79
80
81  '''Denoising and Imputation'''
82
83  def add_gaussian_noise(tensor, eps = 0.1):
84      return tensor + torch.randn_like(tensor)*eps
85
86
87  def salt_and_pepper(x, amount):
88      x = x.cpu().numpy()
89      x = random_noise(x, mode='s&p', amount = amount)
90      x = torch.tensor(x, dtype=torch.float32)
91      return x
92
93
94  #Denoising
95  transform = transforms.Compose([
96      transforms.ToTensor(),
97      transforms.Lambda(lambda x: x.view(-1)),
98      transforms.Lambda(lambda x:add_gaussian_noise(x))
99  ])
100
101 mnist_noise = datasets.MNIST(root = DATA_DIR,
102                              download=False,
103                              train=False,
104                              transform=transform)
105
106 noise_loader = DataLoader(mnist_noise,
107                           batch_size=64,
108                           shuffle=False,
109                           pin_memory=True,
110                           drop_last=False)
111
112
```

```
iterator = iter(noise_loader)

data, _ = next(iterator)

x = data[0].to(device)

corrupted_img = x.cpu().numpy()
corrupted_img = corrupted_img.reshape(28,28)


x_hat, _, _ = model(x)


with torch.no_grad():
    reconstruction_img = x_hat.cpu().numpy().reshape(28, 28)

plt.subplots(nrows = 1, ncols = 2)

plt.subplot(1, 2, 1)
plt.imshow(corrupted_img, cmap='gray')
plt.axis('off')


plt.subplot(1, 2, 2)
plt.imshow(reconstruction_img, cmap='gray')
plt.axis('off')
plt.tight_layout()
plt.show()


# Imputation

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1)),
    transforms.Lambda(lambda x:salt_and_pepper(x, amount = 0.15))
])

mnist_noise = datasets.MNIST(root = DATA_DIR,
                             download=False,
                             train=False,
                             transform=transform)

noise_loader = DataLoader(mnist_noise,
                          batch_size=64,
                          shuffle=False,
                          pin_memory=True,
                          drop_last=False)

iterator = iter(noise_loader)

data, _ = next(iterator)

x = data[0].to(device)

corrupted_img = x.cpu().numpy()
corrupted_img = corrupted_img.reshape(28,28)


x_hat, _, _ = model(x)

with torch.no_grad():
    reconstruction_img = x_hat.cpu().numpy().reshape(28, 28)


plt.subplots(nrows = 1, ncols = 2)

plt.subplot(1, 2, 1)
```

```
plt.imshow(corrupted_img, cmap='gray')
plt.axis('off')


plt.subplot(1, 2, 2)
plt.imshow(reconstruction_img, cmap='gray')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Listing C.1: MNIST application script

Anomaly detection in brain magnetic resonances

```
import torch
from torch import optim
from PIL import Image
from pathlib import Path
from src.model import VAE
from src.configs import brain_configs
from src.train import train
from src.utils import load_brain, visualize_brain_examples, visualize_heatmap

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

ROOT_DIR = Path(__file__).resolve().parents[1]

"""Loading and visualizing training data"""
DATA_DIR = ROOT_DIR / "data" / "mri_brain_tumor" / "Training" / "notumor"
brain_loader = load_brain(DATA_DIR)
visualize_brain_examples(dataloader=brain_loader, num_examples=4, fig_size=(8,8))

"""Defining and training the model"""
model = VAE(**brain_configs, f_out = 'tanh')
optimizer = optim.Adam(model.parameters(), lr= 3e-4)
train(model,brain_loader,optimizer,device,epochs=50, f_out='tanh', max_beta = 1.5)

"""Testing"""
model.eval()

# meningioma
MENINGIOMA_IMG = ROOT_DIR / "data" / "mri_brain_tumor" / "Testing" / "meningioma" /
    "Te-meTr_0003.jpg"
image = Image.open(MENINGIOMA_IMG).convert('L')
visualize_heatmap(image, model, device, threshold=1, cmap='viridis')

# glioma
GLIOMA_IMG = ROOT_DIR / "data" / "mri_brain_tumor" / "Testing" / "glioma" /
    "Te-gl_0026.jpg"
image = Image.open(GLIOMA_IMG).convert('L')
visualize_heatmap(image, model, device, threshold=0.58, cmap='viridis')

# pituitary
PITUITARY_IMG = ROOT_DIR / "data" / "mri_brain_tumor" / "Testing" / "pituitary" /
    "Te-pi_0036.jpg"
image = Image.open(PITUITARY_IMG).convert('L')
visualize_heatmap(image, model, device, threshold=1, cmap='viridis')

# no tumor
NOTUMOR_IMG = ROOT_DIR / "data" / "mri_brain_tumor" / "Testing" / "notumor" /
    "Te-no_0043.jpg"
image = Image.open(NOTUMOR_IMG).convert('L')
visualize_heatmap(image, model, device, threshold=1, cmap='viridis')
```

Listing C.2: Anomaly detection script

Manipulation of the latent space to modify facial features.

```python
import torch
import pandas as pd
from PIL import Image
import matplotlib.pyplot as plt
from pathlib import Path

from src.model import VAE
from src.utils import load_celeba, visualize_celeba_examples, image_reconstruction, latent_interpolation
from src.configs import celeba_configs

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

"""Training examples visualization"""

ROOT_DIR = Path(__file__).resolve().parents[1]
DATA_DIR = ROOT_DIR / "data" / "celeba" / "train"

celeba_loader = load_celeba(DATA_DIR)
visualize_celeba_examples(dataloader=celeba_loader, num_examples=9, img_size=128, fig_size=(12,12))
model = VAE(**celeba_configs, f_out='tanh').to(device)

WEIGHTS_PATH = ROOT_DIR / "weights" / "vae_celeba_trained.pth"
model.load_state_dict(torch.load(WEIGHTS_PATH, map_location= device))


"""Testing"""


model.eval()
TEST_IMG = ROOT_DIR / "data" / "celeba" / "test" / "000012.jpg"
image = Image.open(TEST_IMG).convert('RGB')

plt.imshow(image)
plt.axis('off')
plt.title('Imagen Original')
plt.show()

image_reconstruction(TEST_IMG, model, device)


"""Latent directions calculation"""

# This may take a while

ATTRIBUTES_PATH = ROOT_DIR / "data" / "celeba" / "list_attr_celeba.csv"
df = pd.read_csv(ATTRIBUTES_PATH, sep = ',')
columns = ['Smiling', 'Male', 'Blond_Hair', 'No_Beard', 'Young', 'Eyeglasses']

df = df.set_index('image_id')
df = df[columns]
df.rename(columns={'No_Beard': 'Beard'}, inplace = True)
df['Beard'] = df['Beard']*(-1)

files_per_attribute = {}

for column in df.columns:
    files_per_attribute[column] = df[df[column] == 1].index.tolist()
    files_per_attribute[f'no_{column}'] = df[df[column] == -1].index.tolist()

loaders = {}

for attribute, files in files_per_attribute.items():
    loaders[attribute] = load_celeba(DATA_DIR, image_files = files)


```

```
def encode_latents(dataloader, model, device):
    latents = []

    with torch.no_grad():
        for imgs in dataloader:
            imgs = imgs.to(device)
            imgs = imgs.view(imgs.size(0), -1)
            mu, _ = model.encoder(imgs)
            latents.append(mu)

    return torch.cat(latents, dim = 0)

latents = {}

for attribute, loader in loaders.items():
    z = encode_latents(loader, model, device)
    mean = z.mean(dim = 0)
    latents[attribute] = mean


smile_dir = latents['Smiling'] - latents['no_Smiling']
smile_dir.to(device)

male_dir = latents['Male'] - latents['no_Male']
male_dir.to(device)

blond_dir = latents['Blond_Hair'] - latents['no_Blond_Hair']
blond_dir.to(device)

beard_dir = latents['Beard'] - latents['no_Beard']
beard_dir.to(device)

young_dir = latents['Young'] - latents['no_Young']
young_dir.to(device)

eyeglasses_dir = latents['Eyeglasses'] - latents['no_Eyeglasses']
eyeglasses_dir.to(device)

directions = {
    'smile': smile_dir,
    'male': male_dir,
    'blond': blond_dir,
    'beard': beard_dir,
    'young': young_dir,
    'eyeglasses': eyeglasses_dir
}



"""Visualizations"""



#Original
_, modified_img = latent_interpolation(model, device, directions, TEST_IMG,
    0,0,0,0,0,0)
plt.imshow(modified_img)
plt.axis('off')
plt.show()


# No Smile
_, modified_img = latent_interpolation(model, device, directions, TEST_IMG, smile =
    -1.5, male = 0, blond = 0, beard = 0, young = 0, glasses = 0)
plt.imshow(modified_img)
plt.axis('off')
plt.show()

```

```
# old
_, modified_img = latent_interpolation(model, device, directions, TEST_IMG, smile =
    0, male = 0, blond = 0, beard = 0, young = -3.5, glasses = 0)
plt.imshow(modified_img)
plt.axis('off')
plt.show()

# beard
_, modified_img = latent_interpolation(model, device, directions, TEST_IMG, smile =
    0, male = 0, blond = .5, beard = 3, young = 0, glasses = 0)
plt.imshow(modified_img)
plt.axis('off')
plt.show()

# Smile and glasses
_, modified_img = latent_interpolation(model, device, directions, TEST_IMG, smile =
    1, male = 0, blond = 0, beard = 0, young = 2, glasses = 3.5)
plt.imshow(modified_img)
plt.axis('off')
plt.show()

# blond woman with no smile
_, modified_img = latent_interpolation(model, device, directions, TEST_IMG, smile =
    -1, male = -1.5, blond = 0.75, beard = 0, young = 0, glasses = 0)
plt.imshow(modified_img)
plt.axis('off')
plt.show()
```

Listing C.3: Latent manipulation script