Variational autoencoders with applications

Grabovac, Roberto

UNIVERSITY OF ZAGREB

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS

Roberto Grabovac

VARIATIONAL AUTOENCODERS WITH APPLICATIONS

Master's Thesis

Advisors:
doc. dr. sc. Hrvoje Planinić
assoc. prof. dr. sc. Zvonimir Bujanović

Zagreb, February, 2025.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Dragom Bogu koji mi je podario upornost i disciplinu te omogućio da se bavim onim što najviše volim. Zahvaljujem se i svojoj obitelji i prijateljima koji su bili konstantna podrška tijekom mog školovanja te su uvijek vjerovali u mene.

Contents

Introduction

Generative modeling is one of the most crucial aspects of modern science. Many real-world processes generate specific types of data, but these are often highly complex when all contributing factors are considered. This complexity creates the need for modeling, hypothesis formulation, and testing the observations. A key advantage of this approach lies in its ability to emphasize the most salient characteristics of a process while treating unknown or less relevant aspects as stochastic noise.

In this thesis, we explore the variational autoencoder (VAE) as a concrete example of generative modeling. It was introduced by Kingma and Welling in their 2013 paper, *Auto-Encoding Variational Bayes* [9], and emerged as a significant advancement in generative modeling. A VAE can be viewed as two coupled models, each parameterized independently. The recognition or inference model (encoder) approximates a latent, lower-dimensional representation of the data generated by the process we seek to model. This approximation is then passed to the generative model (decoder), which attempts to reconstruct the original input, followed by the optimization of both parameter sets to maximize the evidence lower bound (ELBO). After a sufficient number of iterations, during which both models are optimized, we obtain an approximation of the data-generating process. Then we can use generative part of VAE to create new, previously unseen data.

The first chapter introduces the fundamental principles of probabilistic models and presents the expectation-maximization algorithm, serving as the foundation for developing the VAE. The second chapter introduces neural networks, which form the backbone of the model architecture. In the third chapter, we combine these concepts by defining both models that make up the VAE, formulating the loss function used to update their parameters, and introducing the reparameterization trick—a powerful technique that reduces gradient noise arising from the sampling process in the recognition model. Finally, the last chapter describes the implementation of the VAE in two variations, differing in their loss functions, applied to the task of generating human portraits.

Chapter 1

Probabilistic models

Probabilistic models are mathematical descriptions of systems or processes using probability theory, enabling the modeling of uncertainty and complex dependencies while offering greater interpretability compared to deterministic ones. In this chapter, we present the foundations of these models and define the probabilistic concepts and terms used in their implementation. Through the analysis and definition, it will become apparent that these models grapple with challenges of intractability due to the Bayesian approach, which are addressed through optimization techniques based on the expectation-maximization algorithm, which form the foundation of highly efficient probabilistic models, including the variational autoencoder.

1.1 Probability theory fundamentals

We assume that the observed datapoint $\mathbf{x}$ is a multidimensional random sample from an unknown underlying process whose true distribution or probability density function $p^*(\mathbf{x})$, is unknown. In general, the observed data is assumed to be independently and identically distributed (i.i.d.). Since the distribution $p^*(\mathbf{x})$ is unknown, we parameterize the probabilistic model with $\boldsymbol{\theta}$ and optimize its parameters to produce a distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ such that:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = p(\mathbf{x} \mid \boldsymbol{\theta}) \approx p^*(\mathbf{x}).$$

Given that underlying processes can be complex, we require the model producing $p_{\boldsymbol{\theta}}(\mathbf{x})$ to be sufficiently flexible and capable of incorporating prior knowledge about the data distribution. This will be achieved using a Bayesian approach with a neural network model, which will be defined in the next chapter, as further refinement of the assumptions about data modeling and their dependencies is required. So, before formal definition of a model in Section 1.2, we can think of it as a mathematical concept which describes this unknown underlying process.

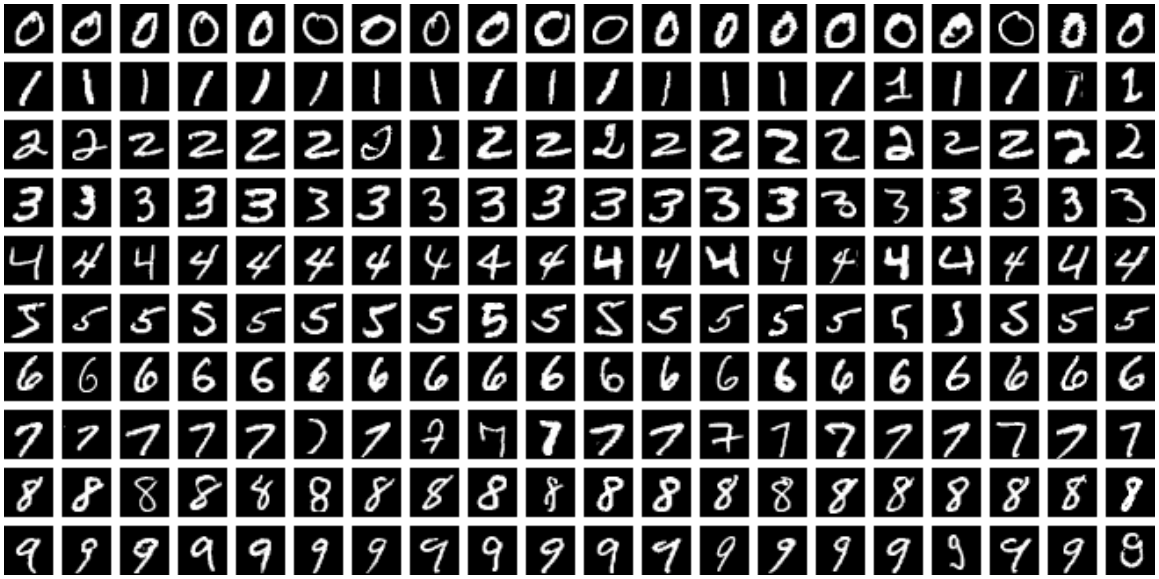


Figure 1.1: The MNIST dataset of digits. We assume there is an underlying process ($p^*(\mathbf{x})$) that describes how each digit ($\mathbf{x}$) is created, and our goal is to find a sufficiently good mathematical model ($p_{\theta}(\mathbf{x})$) that we can then use to generate these digits. For example, we might conclude that the generation of such images can be described by a probability density function, enabling us to sample from it. Based on these samples, we can then use a mapping (e.g., a *neural network*) to produce new images—but more on that later.

Information theory

Probability theory forms the basis of another important concept in understanding uncertainty, known as *information theory*. This field quantifies the information contained in data, providing critical insights into the structure of probabilistic models. Consider a discrete one-dimensional random variable X, and let us explore how much information is conveyed when its specific value is observed. If an event with very low probability occurs, it conveys more information than an event with high probability. Consequently, the amount of information can be interpreted as a *'degree of surprise'*, and its measure depends on the probability distribution $p(X)$. To express this formally, we define a function $h(x)$, which is a monotonic function of the probability $p(X)$, to quantify the information. Intuitively, h must satisfy the property that the information gained from observing two independent events x and y should equal the sum of the information gained from each event individually, i.e., $h(p(x)p(y)) = h(p(x)) + h(p(y))$.

Proposition 1.1.1. *Let h be a continuous real valued function $h : \mathbb{R}^+ \to \mathbb{R}$ that satisfies*

$h(p(x)p(y)) = h(p(x)) + h(p(y))$ for two independent events x and y. Then $h(p(x))$ must be proportional to $\ln p(x)$.

Proof. We start with the property:

$$h(p^2) = h(p) + h(p) = 2h(p).$$

Assume that for all $k \leq K$, we have $h(p^k) = kh(p)$. For $k = K + 1$, we calculate:

$$h(p^{K+1}) = h(p^K p) = h(p^K) + h(p) = Kh(p) + h(p) = (K + 1)h(p).$$

Moreover,

$$h(p^{n/m}) = nh(p^{1/m}) = nm^{-1}h(p) = \frac{n}{m}h(p),$$

where we used:

$$h(p) = h((p^{\frac{1}{m}})^m) = mh(p^{\frac{1}{m}}) \implies h(p^{\frac{1}{m}}) = m^{-1}h(p).$$

This implies that $h(p^x) = xh(p)$, where x is a positive rational number, and by continuity, this also holds for any positive real number. If we define $p(x) = q^x$ for a positive real number q, it follows from the previous conclusions that $h(x)$ is indeed proportional to $\ln p(x)$:

$$\frac{h(p(x))}{\ln(p(x))} = \frac{h(q^x)}{\ln(q^x)} = \frac{xh(q)}{x\ln(q)} = \frac{h(q)}{\ln(q)} \implies h(p(x)) \propto \ln(p(x)).$$

□

We therefore define an information function $h(x) = -\ln p(x)$, where negative sign ensures that information is positive or zero. Choice of the logarithm base is arbitrary, due to the properties of logarithm function. Now we can give formal definition of an *entropy*.

Definition 1.1.2. *Let X be a discrete random variable. The **entropy** of X, denoted by $H[X]$, is defined as:*

$$H[X] = -\sum_i p_i \log_b p_i,$$

where the logarithm base b is commonly chosen as $b = 2$ (bits), $b = e$ (nats), or $b = 10$ (hartleys).

Next, we generalize the concept of entropy to a continuous random vector and choose $b = e$ as the base of the logarithm.

Definition 1.1.3. *Let $\mathbf{X} = (X_1, X_2, \ldots, X_n)^\top$ be a continuous random vector in $\mathbb{R}^n$ with joint probability density function $p(\mathbf{x})$, where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$. The joint **differential entropy** of $\mathbf{X}$, denoted by $H[\mathbf{X}]$, is defined as:*

$$H[\mathbf{X}] = -\int p(\mathbf{x}) \ln p(\mathbf{x})\, d\mathbf{x}.$$

It is also interesting to discuss the *principle of maximum entropy*. This principle states that, subject to known constraints (e.g., known mean and variance), the probability distribution which best represents the current state of knowledge is the one with the largest entropy, as entropy measures the average level of *'uncertainty'* inherent in the random variable's possible outcomes. By maximizing it, we avoid introducing any unwarranted assumptions or biases beyond the known constraints. We illustrate this principle using the example of a discrete random variable that describes the outcomes of a coin toss, i.e., whether it lands heads or tails. If the coin is unfair, meaning it has tails on both sides, the entropy of such a random variable is 0 because there is no uncertainty. On the other hand, if the coin is fair, and thus the probability of landing heads or tails is equal, the amount of uncertainty is maximal, and so is the entropy. This result is not coincidental; it can be proven that the uniform distribution maximizes entropy in the case of discrete random variables.

Let's determine the probability density function $p(x)$ that maximizes entropy for a continuous one-dimensional random variable. For such a distribution to exist, we maximize the differential entropy subject to the normalization constraint and constraints on the first and second moments of $p(x)$:

$$\int_{-\infty}^{\infty} p(x)\,dx = 1,$$

$$\int_{-\infty}^{\infty} x\,p(x)\,dx = \mu,$$

$$\int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,dx = \sigma^2.$$

By solving this optimization problem using methods such as the method of *Lagrange multipliers* (refer to [13] for the proof), we find that the probability density function $p(x)$ that maximizes the differential entropy under these constraints is:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Such a distribution is called the Gaussian (normal) distribution, and it plays a significant role in machine learning. Therefore, we will further study it and generalize it to higher dimensions, where we obtain an analogous result.

Gaussian distribution

Definition 1.1.4. *A random variable X is said to follow a* ***univariate Gaussian distribution*** *with mean $\mu \in \mathbb{R}$ and variance $\sigma^2 > 0$, denoted by $X \sim \mathcal{N}(\mu, \sigma^2)$, if its probability density*

function is given by:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \quad \forall x \in \mathbb{R}.$$

The coefficient at the beginning of the formula serves as a normalization factor, ensuring that the integral of the probability density function equals 1. The exponent of the exponential function represents the squared Z-score, whose formula is given by $Z = \frac{x-\mu}{\sigma}$. It measures the deviation of a value from its corresponding distribution, and the greater its absolute value, the higher the likelihood that the given data point is atypical for the distribution, i.e., an outlier. Let us emphasize why it is important to use the (squared) Z-score for such a metric instead of, for example, the Euclidean distance. The key issue with Euclidean distance is that it does not account for the variance of the data. Specifically, as the variance increases, the importance of such a distance diminishes because it simply reflects greater uncertainty or noise in the dataset. Thus, it is crucial to account for variance when assessing deviations by incorporating it into the denominator. This ensures that as the variance increases, the distance becomes less significant, reflecting the greater uncertainty or noise in the data. Finally, squaring the Z-score makes the PDF symmetrical around its maximum value, which corresponds to the mean. The previously discussed concepts need to be generalized to random vectors in order to define the multivariate Gaussian distribution. When multiple random variables are present within a random vector, there is the possibility of covariance between them, which must also be taken into account. In what follows, we define the covariance matrix, which represents the multidimensional generalization of variance.

Definition 1.1.5. *Let* $\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top$ *be a d-dimensional random vector with mean vector* $\boldsymbol{\mu} = \mathbb{E}[\mathbf{X}] = (\mu_1, \mu_2, \ldots, \mu_d)^\top$*, where* $\mu_i = \mathbb{E}[X_i]$ *for* $i = 1, 2, \ldots, d$*. The* ***covariance matrix*** Σ *of* $\mathbf{X}$ *is a* $d \times d$ *matrix defined as:*

$$\Sigma = \begin{bmatrix} Cov(X_1, X_1) & Cov(X_1, X_2) & \cdots & Cov(X_1, X_d) \\ Cov(X_2, X_1) & Cov(X_2, X_2) & \cdots & Cov(X_2, X_d) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(X_d, X_1) & Cov(X_d, X_2) & \cdots & Cov(X_d, X_d) \end{bmatrix}.$$

The covariance matrix Σ is symmetric, i.e., $\Sigma = \Sigma^\top$, and it can be shown that it is positive semidefinite. This means that $\mathbf{a}^\top \Sigma \mathbf{a} \geq 0$, for any non-zero vector $\mathbf{a} \in \mathbb{R}^d$. Furthermore, positive semidefinitness guarantees all of the eigenvalues to be non-negative. In order to establish the necessary foundation, we present two propositions, proving only the first one, which will later be essential for the geometric intuition.

Proposition 1.1.6. *Let* $\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top$ *be a d-dimensional random vector with mean* $\boldsymbol{\mu} = \mathbb{E}[\mathbf{X}]$ *and covariance matrix* Σ*, and let* $\mathbf{a} \in \mathbb{R}^d$ *be a constant unit vector. Then the*

variance of the projection of $\mathbf{X}$ *onto* $\mathbf{a}$ *is given by:*

$$\mathrm{Var}(\mathbf{a}^\top \mathbf{X}) = \mathbf{a}^\top \Sigma \mathbf{a}.$$

Proof. We compute the expected value and variance of $\mathbf{a}^\top X$:

$$\mathbb{E}[\mathbf{a}^\top \mathbf{X}] = \mathbf{a}^\top \mathbb{E}[\mathbf{X}] = \mathbf{a}^\top \boldsymbol{\mu},$$
$$\mathrm{Var}(\mathbf{a}^\top \mathbf{X}) = \mathbb{E}\left[(\mathbf{a}^\top \mathbf{X} - \mathbb{E}[\mathbf{a}^\top \mathbf{X}])^2\right].$$

Substituting the mean into the variance we get:

$$\mathrm{Var}(\mathbf{a}^\top \mathbf{X}) = \mathbb{E}\left[(\mathbf{a}^\top \mathbf{X} - \mathbf{a}^\top \boldsymbol{\mu})^2\right] = \mathbb{E}\left[(\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu}))^2\right].$$

The square can be written as:

$$(\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu}))^2 = (\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu}))\,(\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu})),$$

since $\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu})$ is a scalar, this can be expressed as:

$$(\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu}))^2 = \mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^\top \mathbf{a}.$$

Thus, the variance finally becomes:

$$\mathrm{Var}(\mathbf{a}^\top \mathbf{X}) = \mathbb{E}\left[\mathbf{a}^\top (\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^\top \mathbf{a}\right] = \mathbf{a}^\top \mathbb{E}\left[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^\top\right] \mathbf{a} = \mathbf{a}^\top \Sigma \mathbf{a}.$$

□

Proposition 1.1.7. *Let* $\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top$ *be a d-dimensional random vector with mean vector* $\boldsymbol{\mu} = \mathbb{E}[\mathbf{X}]$ *and covariance matrix* Σ*. The eigenvectors* $\mathbf{u}_i$ *of* Σ *represent the directions of maximum variance in the data, ordered by their corresponding eigenvalues. Formally, the eigenvectors and eigenvalues satisfy the following:*

1. *Each eigenvector* $\mathbf{u}_i$ *is determined by:*

$$\mathbf{u}_i = \arg \max_{\mathbf{a} \in \mathbb{R}^d, \|\mathbf{a}\| = 1, \mathbf{a} \perp [\{\mathbf{u}_1, \ldots, \mathbf{u}_{i-1}\}]} \mathbf{a}^\top \Sigma \mathbf{a}, \quad i = 1, \ldots, d$$

   where $\mathbf{a} \perp [\{\mathbf{u}_1, \ldots, \mathbf{u}_{i-1}\}]$ *ensures orthogonality to all previously selected eigenvectors.*

2. *For each* $\mathbf{u}_i$*, the associated eigenvalue* λ_i *satisfies:*

$$\mathbf{u}_i^\top \Sigma \mathbf{u}_i = \lambda_i,$$

   where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$.

Now we have everything necessary to define and give geometric interpretation of the generalized Z-score — the *Mahalanobis distance*.

Definition 1.1.8. *Given a random vector* $\mathbf{Q}$ *on* $\mathbb{R}^d$*, with mean vector* $\boldsymbol{\mu} = \mathbb{E}[\mathbf{Q}]$ *and covariance matrix* Σ*, the* ***Mahalanobis distance*** *of a point* $\mathbf{x} \in \mathbb{R}^d$ *from* $\mathbf{Q}$ *is given by*

$$D_M(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}. \tag{1.1}$$

This measure of distance has an interesting geometric interpretation. However, before proceeding, it is necessary to state the *spectral theorem for real symmetric matrices*.

Theorem 1.1.9. *Let* $A \in \mathbb{R}^{n \times n}$ *be a real symmetric matrix. Then there exists an orthogonal matrix* $Q \in \mathbb{R}^{n \times n}$ *and a diagonal matrix* $\Lambda \in \mathbb{R}^{n \times n}$ *such that*

$$A = Q \Lambda Q^\top.$$

Moreover, the diagonal entries of Λ *are the eigenvalues of A, and the columns of Q are the corresponding orthonormal eigenvectors of A.*

From the spectral theorem, we can express covariance matrix Σ using its eigenvectors and eigenvalues in the following form:

$$\Sigma = \sum_{i=1}^{d} \lambda_i \mathbf{u}_i \mathbf{u}_i^\top,$$

where λ_i represents the eigenvalues, and $\mathbf{u}_i$ are the corresponding eigenvectors. Similarly, the inverse of the covariance matrix Σ^{-1} can be represented as:

$$\Sigma^{-1} = \sum_{i=1}^{d} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^\top.$$

By substituting this representation of Σ^{-1} into (1.1), we get simplified squared Mahalanobis distance:

$$D_M^2(\mathbf{x}) = \sum_{i=1}^{d} \frac{y_i^2}{\lambda_i}, \tag{1.2}$$

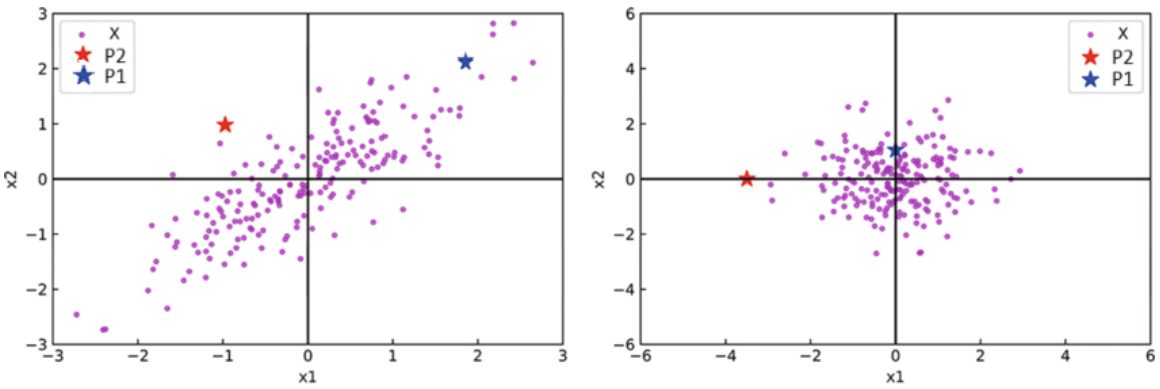
where the new variables y_i are defined as:

$$y_i = \mathbf{u}_i^\top (\mathbf{x} - \boldsymbol{\mu}).$$

Defining the transformed vector $\mathbf{y} = (y_1, y_2, \ldots, y_d)^\top$, we can write:

$$\mathbf{y} = \mathbf{U}(\mathbf{x} - \boldsymbol{\mu}),$$

where $\mathbf{U}$ is the orthogonal matrix whose rows consist of the eigenvectors $\mathbf{u}_i^\top$. If we closely examine the formula for y_i, we conclude that it represents the projection of the vector $\mathbf{x} - \boldsymbol{\mu}$ onto the new axis vector $\mathbf{u}_i$. It is then multiplied by the reciprocal of the eigenvalue in (1.2), ensuring that variance is accounted for when calculating the distance. To summarize, by calculating the eigenvectors of the covariance matrix, we identify the directions of maximum variance in the data, where the variance along an eigenvector is proportional to its corresponding eigenvalue. Such vectors are called the *principal components* of the data. This effectively transitions from the canonical basis to a basis defined by the eigenvectors, allowing us to account for the variance of each component when calculating the distance between two vectors, giving greater weight to the more stable components of the data. Note that if the covariance matrix is an identity matrix, the Mahalanobis distance simplifies to the Euclidean distance, treating all components equally. Relation between those two distances can be seen in Figure 1.2.



(a) Euclidean distance (b) Mahalanobis distance

Figure 1.2: Comparison of Euclidean and Mahalanobis distances: (a) The Euclidean distance treats all directions equally, ignoring the data's variance and correlations. (b) The Mahalanobis distance accounts for the data's covariance structure, effectively scaling distances by variance and accounting for correlations. The value of this metric is computed by applying the Euclidean distance following the coordinate transformation illustrated in the image. P1 and P2 represent observed principal components of the data, i.e., the vectors $\lambda_1\mathbf{u_1}$ and $\lambda_2\mathbf{u_2}$.

After an extensive analysis of the measure of the distance between a vector and a distribution, specifically the Mahalanobis distance, we are now ready to define the *multivariate Gaussian distribution*.

Definition 1.1.10. *A random vector* $\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top$ *in* $\mathbb{R}^d$ *is said to follow a* ***multi-***

variate Gaussian distribution *with mean vector* $\boldsymbol{\mu} \in \mathbb{R}^d$ *and covariance matrix* $\Sigma \in \mathbb{R}^{d\times d}$*, denoted by* $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$*, if its probability density function is given by:*

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \quad \forall \mathbf{x} \in \mathbb{R}^d. \tag{1.3}$$

The multivariate Gaussian distribution appears in numerous contexts and can be derived from various perspectives. For instance, we have observed that, for a single real variable, the Gaussian is the distribution that maximizes entropy. This property also extends to the multivariate Gaussian and can be demonstrated using a similar approach. We can also see that the multivariate Gaussian distribution is indeed an extension of its univariate case. Specifically, the exponent of the exponential function contains the squared Mahalanobis distance, which generalizes the squared Z-score to multiple dimensions, while the normalization factor ensures proper scaling during integration.

1.2 Machine learning models: Concepts and types

Today, we often encounter vast and complex datasets containing immense amounts of information that are beyond a human's capacity to process efficiently and draw precise, meaningful conclusions. This creates a need to automate tasks that are too intricate or time-consuming for manual analysis, enabling resource efficiency and improved accuracy. The solution to this challenge lies in machine learning models. But before demystifying their meaning and how they work, we need to start from the very beginning with the basics.

In machine learning, datasets and their composition vary depending on whether the approach is *supervised* or *unsupervised learning*. In supervised learning, the dataset D consists of m samples that include both input *features* and corresponding *labels*: $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^m\}$. Each sample comprises an input feature vector $\mathbf{x}_i \in \mathbb{R}^n$ and an associated label y_i. The features $\mathbf{x}$ represent the independent variables or inputs to the model, while the labels y are the dependent variables or outputs that the model aims to predict. The set of all possible feature vectors forms the *feature space*, which is the n-dimensional space encompassing all combinations of input variables. In unsupervised learning, the dataset D consists solely of input samples without associated labels: $D = \{\mathbf{x}_i\}_{i=1}^m$. In this scenario, there are no labels y_i. The model's objective is to discover patterns, structures, or relationships within the data based only on the features $\mathbf{x_i}$. Mixture of those two approaches is *semi-supervised learning* where the dataset D consists of a small set of labeled samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^l$ and a larger set of unlabeled samples $\{\mathbf{x}_j\}_{j=l+1}^m$. This approach combines aspects of both supervised and unsupervised learning, leveraging the labeled data to guide the learning process while using the unlabeled data to capture the underlying structure of the feature space. A compact representation of feature vectors is the *design matrix* $\mathbf{X}$. Each row corresponds to a single sample (feature vector), and each column corresponds to a

specific feature (or variable). If we have m samples and n features, the design matrix $\mathbf{X}$ is an $m \times n$ matrix defined as:

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix}.$$

Now, we establish a connection between the dataset and the model through the concept of a *hypothesis*. In machine learning, a hypothesis is a function that represents a potential explanation for the relationships within data. In supervised learning, the *hypothesis space* $\mathcal{H}$ comprises all functions $h : X \to Y$ that map input features $\mathbf{x} \in X$ to output labels $y \in Y$. The goal is to find the optimal hypothesis $h^* \in \mathcal{H}$ that best approximates the true mapping by minimizing a loss function over the training data. This can be mathematically formulated as:

$$h^* = \arg\min_{h \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(h(\mathbf{x}_i), y_i),$$

where $\mathcal{L}$ is a *loss function* that measures the discrepancy between the predicted output $h(\mathbf{x}_i)$ and the true output y_i. In unsupervised learning, hypotheses are functions that describe the underlying structure or distribution of the data without labeled outputs, focusing on patterns like clustering or density estimation within the input space $\mathbf{X}$. For clustering, the hypothesis might be that the data can be partitioned into k clusters $\{C_1, C_2, \ldots, C_k\}$, aiming to minimize an objective function such as:

$$\arg\min_{\{C_j\}} \sum_{j=1}^{k} \sum_{\mathbf{x}_i \in C_j} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2,$$

where $\boldsymbol{\mu}_j$ is the centroid of cluster C_j. Another example is density estimation, which we mentioned at the beginning of the Section 1.1 where the goal is to find the parameter $\boldsymbol{\theta}$ that best fits the i.i.d. data by maximizing the likelihood:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\boldsymbol{\theta}}(\mathbf{x}_i). \tag{1.4}$$

Density estimation, as shown in (1.4), is often referred to as *maximum likelihood estimation (MLE)*. However, for computational convenience, it is commonly expressed in the following form:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i).$$

Now, we define a *model* as a set of hypotheses. Since a hypothesis is a function, a model can be understood as a set of functions. Formally, we can define it as:

$$\mathcal{H} = \{h_\theta : \boldsymbol{\theta} \in \boldsymbol{\Theta}\},$$

where $\mathcal{H}$ represents the model, h_θ denotes a hypothesis, and $\boldsymbol{\Theta}$ is the set of parameters that characterize the hypotheses. Therefore, machine learning can be explained as the process of searching through the set of hypotheses $\mathcal{H}$ to find the best hypothesis $h \in \mathcal{H}$. The notion of what makes a hypothesis *best* depends on the loss function, which in turn depends on the type of model and the problem being solved. From this, we can see that machine learning is directly linked to optimization problems where techniques are determined by the loss function.

To summarize, we can divide the entire process into three components: selecting a parametrized model that hypothetically describes the relationship between the data, defining the loss function, and conducting an optimization procedure to minimize it. Now, let's explore the different types of models, based on the problems they are designed to solve, and conclude with the one that will be discussed in detail in this thesis.

Deterministic vs. stochastic models

A *deterministic model* provides a specific output for a given input without any inherent randomness. Mathematically, it defines a function $f : \mathcal{X} \to \mathcal{Y}$, where each input $\mathbf{x} \in \mathcal{X}$ maps to an output $\mathbf{y} = f(\mathbf{x}) \in \mathcal{Y}$. For example, in linear regression, the relationship between inputs and outputs is modeled as $\hat{y} = \boldsymbol{\theta}^\top \mathbf{x}$, where $\boldsymbol{\theta}$ are the learned parameters. In contrast, a *stochastic model* incorporates randomness and provides a probability distribution over possible outputs. It models the conditional probability $p(\mathbf{y}|\mathbf{x})$, capturing uncertainty in the predictions. Probabilistic models are inherently stochastic, as they characterize the uncertainty and variability in data. For instance, in probabilistic linear regression, we might model the outputs as $\mathbf{y} = \boldsymbol{\theta}^\top \mathbf{x} + \varepsilon$, where ε is a random error term typically assumed to follow a normal distribution.

Generative vs. discriminative models

Generative models aim to model the joint probability distribution $p(\mathbf{x}, \mathbf{y})$ of the inputs and outputs. They can generate new realistic data instances by sampling from this distribution because they learn how the data is generated. Examples are *Gaussian mixture models* (convex combination of multivariate Gaussian distributions) and the main focus of this thesis - *variational autoencoder (VAE)*. On the other hand, *discriminative models* focus on modeling the conditional probability $p(\mathbf{y}|\mathbf{x})$ directly or learning a direct mapping from inputs to outputs without modeling the underlying data distribution. The most obvious example is *logistic regression*.

Latent variable models

The previously mentioned types of models can also be probabilistic in nature. *Probabilistic models* offer an advantage over non-probabilistic ones because they account for variability and uncertainty in the data, enabling better predictions and decision-making, especially in complex situations. For instance, in a *k-means* model, each sample is assigned a single label. However, a sample could be very close to two clusters, yet the output only shows the cluster to which the model assigned it. This results in the loss of crucial information as the uncertainty in choosing between the two clusters is ignored. Therefore, it is preferable to use a probabilistic model, which could, for example, return the probabilities of the sample belonging to each cluster.

In general, probabilistic models do not assume the existence of hidden features; instead, they learn directly from the provided variables. In contrast, *latent variable models* assume the presence of hidden features, known as *latent variables*, which are not directly observed but are inferred from the observed data. There are many intuitive reasons why it is natural to assume the existence of latent variables. For example, imagine a model whose feature space consists of grayscale images containing circles, and the goal is to learn to generate these images. Initially, the size of this space would equal the number of pixels in the image (e.g., 28 x 28 = 784). However, upon closer consideration, creating an image requires only the coordinates of the circle's center and its radius, making the *latent space* three-dimensional. Figure 1.3 illustrates this concept. More about *latent space* and *latent variables* will be discussed in Section 1.4, as the *variational autoencoder* is a generative model that fundamentally relies on them.

1.3 Frequentist and Bayesian approach

In statistics and probability, two main approaches have been widely studied: the *frequentist* and *Bayesian approaches*. Each of these frameworks has its own assumptions, unique principles, as well as advantages and disadvantages. The frequentist approach introduces the concept of probability as being synonymous with long-run frequency and emphasizes observable data rather than subjective beliefs. It assumes that parameters (such as the mean or variance) are fixed, unknown values, while randomness is explained through sampling from a random process, as different data samples may lead to different estimates. The main goal is to use data to make objective inferences about the fixed parameters without incorporating prior beliefs. We have already seen the maximum likelihood estimation of the dataset $\mathcal{D}$ as one of the methods to achieve this goal:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathcal{D}) = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\boldsymbol{\theta}}(\mathbf{x}_i),$$

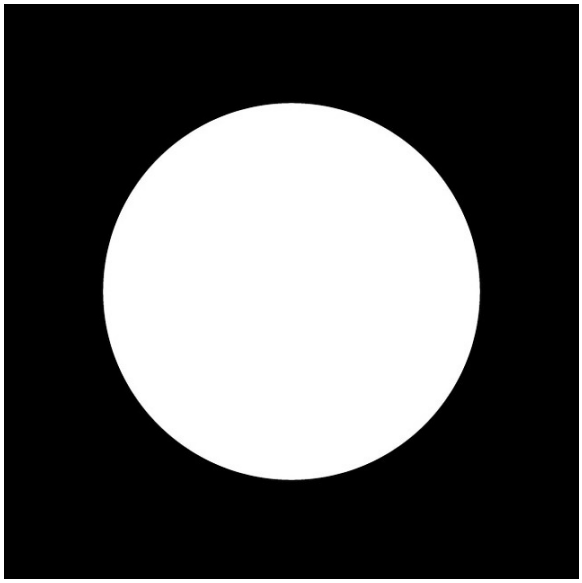


Figure 1.3: Let us assume that we want to create a generative model based on latent variables to produce circles similar to those in the given image. One of the model's hyperparameters is the dimension of the latent space. Naturally, this dimension is significantly smaller than the number of pixels composing the image. We can observe that a dimension of three is sufficient, as each circle is characterized by its center coordinates (x, y) and its radius r. Thus, for the given image, we could design a model that outputs (x, y, r). Of course, specifying a latent space of dimension three does not necessarily mean that the model will explicitly learn to represent the center coordinates and radius. However, this reasoning provides an intuitive guide for understanding latent space and selecting an appropriate dimensionality.

which chooses the parameter values under which the observed data is most probable. This aligns with the principle of selecting the most plausible explanation for the data. However, this approach carries the risk of overfitting in complex models and, more critically, provides limited insight into the uncertainty of parameter estimates. When the sample size is small, the observed data may not adequately represent the true underlying population distribution. This leads to variability in the MLE estimates because the likelihood function is constructed based on the limited data available. On the other hand, the Bayesian approach incorporates prior beliefs about the parameters and updates these beliefs using observed data. It contrasts with the frequentist approach by explicitly treating parameters as random variables with associated probability distributions, rather than fixed unknown quantities, thereby providing a natural way to quantify the uncertainty of parameter estimates. This

involves calculating posterior probabilities using Bayes' theorem:

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

where $p(\mathcal{D}|\boldsymbol{\theta})$ represents the conditional probability, which we have previously denoted as $p_{\boldsymbol{\theta}}(\mathcal{D})$. From the Bayesian viewpoint, there is only a single data set $\mathcal{D}$ (the one that is actually observed). The process of calculating posterior probabilities is iterative, meaning that the posterior probability becomes the prior for future data that will be observed. A key feature of the Bayesian perspective is that incorporating prior knowledge is a natural part of the process. Imagine you observe a student who has scored 100% on the first three exams. Using maximum likelihood estimation, you might predict that the student will continue to score 100% on all future exams. However, a Bayesian approach, incorporating prior knowledge (such as the fact that perfect scores are rare), would lead to a more cautious prediction, acknowledging the possibility of variability in future exam scores.

However, a major drawback of the Bayesian framework is that it often requires solving highly complex integrals during inference, which become intractable, as we will see in Section 1.4. This is because modern deep learning models can have millions or even billions of parameters and transformations, and since the expression to be integrated depends on them, even basic approximations of such integrals become computationally impractical. Given a limited compute budget and abundant training data, it is often more effective to use maximum likelihood methods, typically combined with regularization, on a large neural network rather than applying a Bayesian approach to a smaller model. Luckily, since intractable integrals are commonly encountered, various techniques have been developed to make them tractable or to approximate them efficiently. *Amortized (variational) inference* is one such technique, which we will explore later in this paper.

1.4 Latent space

First, we consider the theoretical construct known as the *data manifold*, which is derived from the properties of the data itself. The data manifold represents the lower-dimensional structure within the high-dimensional space where the observed data actually resides. In high-dimensional datasets, although the ambient space may have many dimensions, the data typically occupies a region of much lower intrinsic dimensionality. The *manifold hypothesis* posits that natural high-dimensional data lies on or near a manifold of significantly lower dimensionality. For instance, images are often represented as high-dimensional vectors (e.g., a 64×64 grayscale image corresponds to 4096 dimensions), while the intrinsic dimensionality is much lower, as the true underlying factors that vary in the data, such as pose, lighting, and object identity, are fewer. Another way to see that real data is confined to low-dimensional manifolds is to consider the task of generating random images.

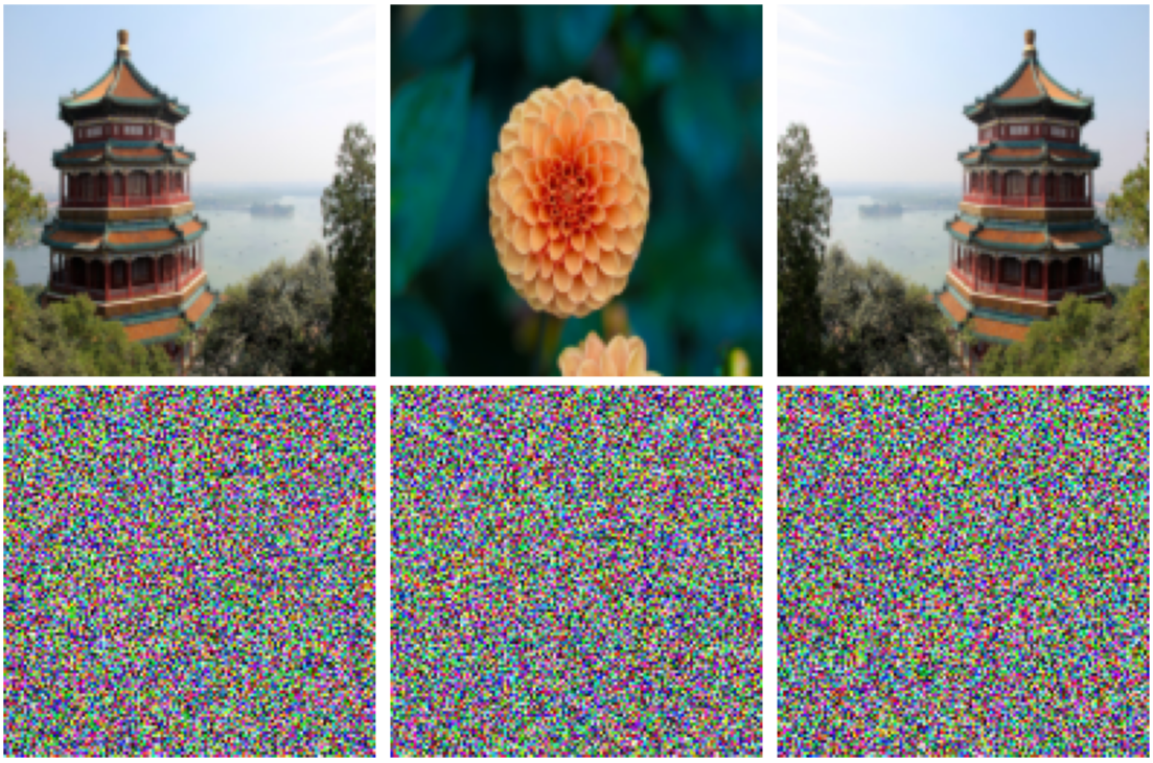


Figure 1.4: Examples of pictures in 128×128 feature space where each feature is one pixel. The top row shows examples of natural images, whereas the bottom row shows randomly generated images obtained by drawing pixel values from a uniform probability distribution over the possible pixel colours.

In Figure 1.4, we observe examples of natural images along with synthetic images of the same resolution, generated by sampling each of the red, green, and blue intensities for every pixel independently and randomly from a uniform distribution. It is clear that none of the synthetic images resemble natural images. This discrepancy arises because random images lack the strong pixel correlations that natural images exhibit. For instance, adjacent pixels in natural images have a much higher probability of being the same or having similar colours compared to the randomly generated counterparts. So, each image in Figure 1.4 represents a point in a high-dimensional space, but natural images only occupy a tiny fraction of this space. Therefore, it is reasonable to apply this approach in practice, which has led to the development of *latent variable models*. As the name suggests, these models are based on the existence of *latent variables* and simulate the data manifold using a *latent space*. Latent variables are variables that are not directly observed in the data but

are inferred from the model. They represent underlying, hidden factors that influence the observed data and together they define *latent space*. Formally:

(i) *observed variables*: $\mathbf{X} = (X_1, X_2, \ldots, X_n)$, where each X_i is a random variable.

(ii) *latent variables*: $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_m)$, where each Z_j is a random variable.

It is important to note that the data manifold is a theoretical construct, while the latent space is its practical implementation in the form of an approximation. Parameters such as dimensionality are typically specified when defining the model. Training follows a Bayesian approach, where we aim to align the latent variable space as closely as possible with its prior distribution, which is often assumed to be a multivariate Gaussian distribution.

Now, we define posterior probability using the latent variables:

$$p(\mathbf{Z} = \mathbf{z} \mid \mathbf{X} = \mathbf{x}, \boldsymbol{\theta}) = \frac{p(\mathbf{X} = \mathbf{x}, \mathbf{Z} = \mathbf{z} \mid \boldsymbol{\theta})}{p(\mathbf{X} = \mathbf{x} \mid \boldsymbol{\theta})}.$$

The marginal probability of the observed data $\mathbf{X}$ in the presence of latent variables $\mathbf{Z}$ can be expressed differently depending on whether the latent variables are continuous or discrete. For continuous latent variables, the marginal probability is computed as an integral over all possible values of $\mathbf{Z}$:

$$p(\mathbf{X} = \mathbf{x} \mid \boldsymbol{\theta}) = \int_{\mathbb{R}^m} p(\mathbf{X} = \mathbf{x}, \mathbf{Z} = \mathbf{z} \mid \boldsymbol{\theta})\, d\mathbf{z} = \int_{\mathbb{R}^m} p(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} = \mathbf{z}, \boldsymbol{\theta})\, p(\mathbf{Z} = \mathbf{z} \mid \boldsymbol{\theta})\, d\mathbf{z}.$$

For discrete latent variables, the marginal probability is instead computed as a summation over all possible discrete values of $\mathbf{Z}$:

$$p(\mathbf{X} = \mathbf{x} \mid \boldsymbol{\theta}) = \sum_{\mathbf{z}} p(\mathbf{X} = \mathbf{x}, \mathbf{Z} = \mathbf{z} \mid \boldsymbol{\theta}) = \sum_{\mathbf{z}} p(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} = \mathbf{z}, \boldsymbol{\theta})\, p(\mathbf{Z} = \mathbf{z} \mid \boldsymbol{\theta}).$$

However, the general problem with such models is intractability, which often arises due to the high dimensionality of the latent space, making it computationally infeasible to calculate the marginal likelihood $p(\mathbf{X} = \mathbf{x} \mid \boldsymbol{\theta})$. Bayes' rule thus implies the same holds for the posterior probability, since this normalization constant appears in the denominator. Nevertheless, there are techniques that address this issue, and the one we will use is *amortized variational inference*. More on this will follow.

In training latent variable models, the goal is to make the distribution of latent variables as close as possible, or as minimally distant, from their prior distribution. So, we need a metric to compare two distributions.

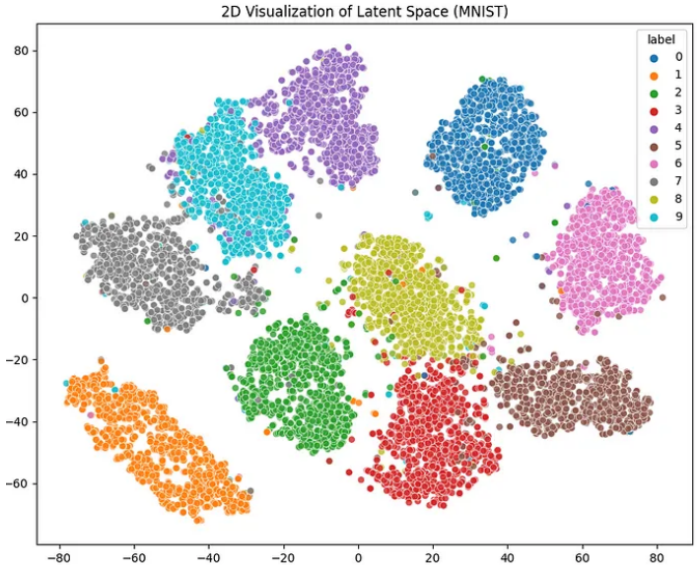


Figure 1.5: The latent representation of the MNIST dataset in a two-dimensional space organizes digits into distinct latent clusters. Each cluster is represented by a corresponding color. If $\mathbf{x}$ represents the digit 7, then the density $p(\mathbf{Z} = (-60, 20) \mid \mathbf{X} = \mathbf{x}, \boldsymbol{\theta})$ is higher than, for instance, $p(\mathbf{Z} = (80, 0) \mid \mathbf{X} = \mathbf{x}, \boldsymbol{\theta})$, for some fixed parameter $\boldsymbol{\theta}$. To achieve the most accurate latent representation of the input data $\mathbf{x}$, we aim to maximize $p(\mathbf{Z} = \mathbf{z} \mid \mathbf{X} = \mathbf{x}, \boldsymbol{\theta})$.

1.5 Kullback–Leibler divergence

In this section we introduce a metric to compare the similarity between two distributions—namely, *the Kullback-Leibler divergence*. Consider a situation where we approximate a true probability distribution $p(x)$ with another distribution $q(x)$. The additional information required to describe data using $q(x)$, rather than the true distribution $p(x)$, can be expressed as the KL divergence:

$$\begin{aligned} \mathrm{KL}(p\|q) &= -\int p(x) \ln q(x)\, dx - \left(-\int p(x) \ln p(x)\, dx\right) \\ &= -\int p(x) \ln \frac{q(x)}{p(x)}\, dx. \end{aligned}$$

This measure, also referred to as *relative entropy*, is asymmetric ($\mathrm{KL}(p\|q) \neq \mathrm{KL}(q\|p)$) and quantifies the dissimilarity between the two distributions. Importantly, KL divergence satisfies $\mathrm{KL}(p\|q) \geq 0$, with equality if and only if $p(x) = q(x)$. To prove this statement, we introduce convex functions and Jensen's inequality.

Definition 1.5.1. *For a real-valued function f, we say that f is convex on an interval $I \subseteq \mathbb{R}$ if:*

$$f\left(\frac{x_1 + x_2}{2}\right) \leq \frac{f(x_1) + f(x_2)}{2},$$

for all $x_1, x_2 \in I$.

Theorem 1.5.2 (Jensen's Inequality)**.** *Let $f : I \to \mathbb{R}$ be a function defined on an open interval $I \subseteq \mathbb{R}$. The function f is continous and convex on I if the following inequality holds:*

$$f((1-\lambda)x_1 + \lambda x_2) \leq (1-\lambda)f(x_1) + \lambda f(x_2), \tag{1.5}$$

for all $x_1, x_2 \in I$ and $\lambda \in [0, 1]$.

Using the technique of proof by induction, we can generalize (1.5) for a convex function f that acts on a convex combination of points as follows:

$$f\left(\sum_{i=1}^{M} \lambda_i x_i\right) \leq \sum_{i=1}^{M} \lambda_i f(x_i). \tag{1.6}$$

If we associate the coefficients with the probabilities of the events of the discrete random variable X as $\mathbb{P}(X = x_i) = \lambda_i$, then (1.6) can be written:

$$f\left(\mathbb{E}[X]\right) \leq \mathbb{E}[f(X)]. \tag{1.7}$$

Observe that each derived result above for the convex function (inequations (1.5), (1.6) and (1.7)) has an analogous counterpart for the concave case, with all inequalities reversed, since if f is convex, then $-f$ is concave.

For continuous variables, Jensen's inequality takes the analogous form:

$$f\left(\int x p(x)\, dx\right) \leq \int f(x) p(x)\, dx. \tag{1.8}$$

Applying the inequality in the form of (1.8) to the KL-divergence, we obtain:

$$\mathrm{KL}(p\|q) = -\int p(x) \ln \frac{q(x)}{p(x)}\, dx \geq -\ln \int q(x)\, dx = 0.$$

It is important to note that we relied on the convexity of $-\ln x$ for the inequality, and the normalization condition $\int q(x)\, dx = 1$ for the final equality. Since $-\ln x$ is strictly convex, equality holds if and only if $q(x) = p(x)$ for all x. Thus, the Kullback–Leibler divergence can be interpreted as a measure of the dissimilarity of the two distributions $p(x)$ and $q(x)$.

This metric also has a profound relationship with the maximum likelihood estimation. When the data is generated from an unknown true distribution $p(x)$, we often approximate

$p(x)$ using a parametric distribution $q(x \mid \boldsymbol{\theta})$ defined by parameters $\boldsymbol{\theta}$. Recall that when provided with N points sampled from the distribution $p(x)$, the expectation of an arbitrary function f can be approximated as a finite sum:

$$\mathbb{E}[f] \approx \frac{1}{N} \sum_{n=1}^{N} f(x_n). \tag{1.9}$$

Thus, from equation (1.9), we establish the goal to minimize:

$$\mathrm{KL}(p\|q) = -\int p(x) \ln \frac{q(x)}{p(x)}\, dx \approx \frac{1}{N} \sum_{n=1}^{N} \left(-\ln q(x_n \mid \boldsymbol{\theta}) + \ln p(x_n)\right).$$

Since the term involving $\ln p(x_n)$ is independent of $\boldsymbol{\theta}$, minimizing the KL divergence is equivalent to maximizing the log-likelihood of the data under $q(x \mid \boldsymbol{\theta})$.

1.6 Expectation maximization algorithm

We now introduce one of the fundamental algorithms for latent variable models, which will serve as the starting point for the variational autoencoder. Suppose we have a dataset $\mathcal{D} = \{\mathbf{x_1}, \ldots, \mathbf{x_N}\}$ consisting of i.i.d. samples for training a latent variable model $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$, where $\mathbf{z}$ is the latent variable (for example, an image can be represented as $\mathbf{x}$, while its lower-dimensional representation corresponds to the latent variable $\mathbf{z}$). The marginal density of $\mathbf{x}$ can then be obtained by integrating over all possible values of $\mathbf{z}$:

$$\ln p_{\boldsymbol{\theta}}(\mathbf{x}) = \ln p(\mathbf{x} \mid \boldsymbol{\theta}) = \ln \int p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta}) d\mathbf{z} \implies \ln p_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{i=1}^{N} \ln \int p(\mathbf{x_i}, \mathbf{z} \mid \boldsymbol{\theta}) d\mathbf{z}. \tag{1.10}$$

Looking closely at equation (1.10), we observe that the integral prevents the logarithm from directly acting on the joint distribution. This results in a non-convex optimization problem for the model parameters, which has been empirically shown to be difficult to solve. To introduce approximation techniques, we begin with the new terminology. Consider the case where, for each observation $\mathbf{x}_i$, the corresponding value of the latent variable $\mathbf{z}_i$ is known. In this scenario, we refer to the pair $\{(\mathbf{x}_i, \mathbf{z}_i)\}_{i=1}^{N}$ as the *complete* data set, while the actual observed data $\mathbf{x}_i$ is considered *incomplete*. In practice, the complete data set is not available, as our goal is to use the model to estimate the corresponding latent variables for the observed variables. However, it turns out that by proceeding this way, we can derive a lower bound, and by maximizing this bound, we effectively maximize the log-likelihood.

In the following analysis, we will focus on a single data point $\mathbf{x}$, and the obtained results can be easily extended to the entire dataset $\mathcal{D}$ using the implication in (1.10). Thus, we

aim to optimize $\ln p_{\boldsymbol{\theta}}(\mathbf{x})$ with respect to the parameters $\boldsymbol{\theta}$, i.e., the expression:

$$\ln p_{\boldsymbol{\theta}}(\mathbf{x}) = \ln \int p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta}) d\mathbf{z}.$$

Using Jensen's inequality for the concave function $f(x) = \ln x$, for an arbitrary distribution Q over $\mathbf{z}$ (where $\int Q(\mathbf{z}) d\mathbf{z} = 1$ and $Q(\mathbf{z}) \geq 0$), we obtain:

$$\ln p_{\boldsymbol{\theta}}(\mathbf{x}) = \ln \int p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta}) d\mathbf{z} = \ln \int Q(\mathbf{z}) \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{Q(\mathbf{z})} d\mathbf{z} \geq \int Q(\mathbf{z}) \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{Q(\mathbf{z})} d\mathbf{z}. \quad (1.11)$$

More precisely, Jensen's inequality is applied to:

$$f\left(\mathbb{E}_{\mathbf{z} \sim Q}\left[\frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{Q(\mathbf{z})}\right]\right) \geq \mathbb{E}_{\mathbf{z} \sim Q}\left[f\left(\frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{Q(\mathbf{z})}\right)\right].$$

The inequality (1.11) shows that for an arbitrary Q, we have a lower bound on $\ln p_{\boldsymbol{\theta}}(\mathbf{x})$. Now, we aim to find a distribution that achieves equality for a fixed $\boldsymbol{\theta}$. By choosing $Q(\mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$, it is easy to see that we obtain equality:

$$\begin{aligned}
\int Q(\mathbf{z}) \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{Q(\mathbf{z})} d\mathbf{z} &= \int p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}) \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})} d\mathbf{z} \\
&= \int p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}) \ln \frac{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}) p(\mathbf{x} \mid \boldsymbol{\theta})}{p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})} d\mathbf{z} \\
&= \int p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}) \ln p(\mathbf{x} \mid \boldsymbol{\theta}) d\mathbf{z} \\
&= \ln p(\mathbf{x} \mid \boldsymbol{\theta}) \int p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}) d\mathbf{z} \\
&= \ln p(\mathbf{x} \mid \boldsymbol{\theta}). \quad \left(\text{since } \int p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}) d\mathbf{z} = 1\right)
\end{aligned}$$

For the simplicity of the analysis in the following, let us denote:

$$\text{ELBO}(\mathbf{x}, Q, \boldsymbol{\theta}) = \int Q(\mathbf{z}) \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{Q(\mathbf{z})} d\mathbf{z}.$$

The notation comes from the fact that this expression is referred to in the literature as the *evidence lower bound (ELBO)*. We will also use it in defining the loss function of the variational autoencoder, where we will see its additional interpretations. Thus, up to this point, we have proven:

$$\forall Q, \boldsymbol{\theta}, \mathbf{x}, \quad \ln p(\mathbf{x} \mid \boldsymbol{\theta}) \geq \text{ELBO}(\mathbf{x}, Q, \boldsymbol{\theta}). \quad (1.12)$$

Intuitively, the main idea of the EM algorithm will be the alternating updates of Q and $\boldsymbol{\theta}$:

a) Set $Q(\mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$, which gives $\text{ELBO}(\mathbf{x}, Q, \boldsymbol{\theta}) = p_{\boldsymbol{\theta}}(\mathbf{x})$ for the data $\mathbf{x}$ and the current value of the parameter $\boldsymbol{\theta}$ (E-step);

b) Maximize $\text{ELBO}(\mathbf{x}, Q, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ for a fixed choice of Q (M-step).

We have reached the conclusions for a single $\mathbf{x}$. Now, we extend this to $\mathcal{D} = \{\mathbf{x_1}, \ldots, \mathbf{x_N}\}$ by indexing based on the sample:

$$\text{ELBO}(\mathbf{x_i}, Q_i, \boldsymbol{\theta}) = \int Q(\mathbf{z_i}) \ln \frac{p(\mathbf{x_i}, \mathbf{z_i} \mid \boldsymbol{\theta})}{Q(\mathbf{z_i})} d\mathbf{z_i}.$$

By summing over all the samples, we obtain:

$$\begin{aligned} \ln p_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{i=1}^{N} \ln \int p(\mathbf{x_i}, \mathbf{z_i} \mid \boldsymbol{\theta}) d\mathbf{z_i} &\geq \sum_{i=1}^{N} \text{ELBO}(\mathbf{x}_i, Q_i, \boldsymbol{\theta}) \\ &= \sum_{i=1}^{N} \int Q_i(\mathbf{z_i}) \ln \frac{p(\mathbf{x_i}, \mathbf{z_i} \mid \boldsymbol{\theta})}{Q_i(\mathbf{z_i})} d\mathbf{z_i}. \end{aligned} \tag{1.13}$$

Now, the analogous result follows, namely, for an arbitrary set of distributions $Q_1, \ldots, Q_N$, the expression:

$$\sum_{i=1}^{N} \int Q_i(\mathbf{z_i}) \ln \frac{p(\mathbf{x_i}, \mathbf{z_i} \mid \boldsymbol{\theta})}{Q_i(\mathbf{z_i})} d\mathbf{z_i}$$

represents a lower bound on the log-likelihood $\ln p_{\boldsymbol{\theta}}(\mathcal{D})$. Thus, in the E-step, we set $Q_i(\mathbf{z}_i) = p(\mathbf{z}_i \mid x_i, \boldsymbol{\theta}), i = 1, \ldots, N$ and compute the log-likelihood. After that, we maximize the expression in (1.13) with respect to the parameter $\boldsymbol{\theta}$. The pseudocode is shown in Algorithm 1.

Now, it is necessary to prove the convergence of the algorithm. Let us assume that $\boldsymbol{\theta}^{(t)}$ and $\boldsymbol{\theta}^{(t+1)}$ are the parameters from two consecutive iterations of the algorithm. We will prove:

$$\ln p_{\boldsymbol{\theta}^{(t)}}(\mathcal{D}) = \ln p(\mathcal{D} \mid \boldsymbol{\theta}^{(t)}) \leq \ln p(\mathcal{D} \mid \boldsymbol{\theta}^{(t+1)}) = p_{\boldsymbol{\theta}^{(t+1)}}(\mathcal{D}).$$

By the construction of the algorithm, in iteration t we have chosen $Q_i^{(t)}(\mathbf{z}_i) = p(\mathbf{z}_i \mid x_i, \boldsymbol{\theta}^{(t)}), i = 1, \ldots, N$, and with this choice of distributions Q_i, the following holds:

$$\ln p(\mathcal{D} \mid \boldsymbol{\theta}^{(t)}) = \sum_{i=1}^{N} \text{ELBO}(\mathbf{x}_i, Q_i^{(t)}, \boldsymbol{\theta}^{(t)}).$$

Then, we know that the parameters $\boldsymbol{\theta}^{(t+1)}$ are obtained by maximizing the right-hand side of the above expression. Therefore, we have:

$$\begin{aligned}\ln p(\mathcal{D} \mid \boldsymbol{\theta}^{(t+1)}) &\geq \sum_{i=1}^{N} \text{ELBO}(\mathbf{x}_i, Q_i^{(t)}, \boldsymbol{\theta}^{(t+1)}) \\ &\geq \sum_{i=1}^{N} \text{ELBO}(\mathbf{x}_i, Q_i^{(t)}, \boldsymbol{\theta}^{(t)}) \\ &= \ln p(\mathcal{D} \mid \boldsymbol{\theta}^{(t)}).\end{aligned}$$

The first inequality follows from the fact that the evidence lower bound holds for an arbitrary choice of Q and $\boldsymbol{\theta}$, while the second follows from the M-step, where we choose $\boldsymbol{\theta}^{(t+1)}$ such that:

$$\arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \text{ELBO}(\mathbf{x_i}, Q_i^{(t)}, \boldsymbol{\theta}).$$

Thus, EM ensures that the log-likelihood converges monotonically. One reasonable convergence test would be to check if the increase in log-likelihood between successive iterations is smaller than some tolerance parameter, and to declare convergence if EM is improving it too slowly. With this, we have finally described the EM algorithm and proven its correctness.

Algorithm 1 Expectation-maximization algorithm

procedure EMAlgorithm
  Input: i.i.d data $\{(\mathbf{x}_i, \mathbf{z}_i)\}_{i=1}^N$, joint distribution $p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta})$, initial parameters $\boldsymbol{\theta}^{\text{old}}$
  Output: Final parameters $\boldsymbol{\theta}$
  while not converged **do**
    $Q_i(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x_i}, \boldsymbol{\theta}^{\text{old}}), \quad i = 1, \ldots, N$ ▷ E-step
    $\boldsymbol{\theta}^{\text{new}} \leftarrow \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \text{ELBO}(\mathbf{x_i}, Q_i, \boldsymbol{\theta})$ ▷ M step
    $\mathcal{L} \leftarrow \sum_{i=1}^{N} \ln p(\mathbf{x}_i \mid \boldsymbol{\theta}^{\text{new}})$ ▷ Evaluate log-likelihood
    $\boldsymbol{\theta}^{\text{old}} \leftarrow \boldsymbol{\theta}^{\text{new}}$ ▷ Update parameters
  end while
  return $\boldsymbol{\theta}^{\text{new}}$
end procedure

Chapter 2

Neural networks

Neural networks represent one of the leading concepts in machine learning, based on the structure and function of the human brain. A neural network, much like the human brain, is a cohesive unit whose proper functioning results from the good connectivity of its basic components—*(artificial) neurons*. Good connectivity is achieved through a chain reaction of weight updates using the *gradient descent* technique in accordance with the training data, mirroring how synapses in the human brain strengthen or weaken based on experience and learning from the observed data. Obviously, to perform these weight updates, we need an efficient technique to calculate the gradient, and this is precisely what we call *backpropagation*. All the mentioned concepts will be elaborated on in detail, leading to a formal definition of a neural network and the process of optimizing its accuracy. Furthermore, we will define a specific type designed for image processing challenges - a *convolutional neural network*. However, before delving into the details, it is necessary to motivate their study, which arises from numerous shortcomings of the assumptions regarding the linearity of data, fixed *basis functions* that do not depend on it, and the famously known *curse of dimensionality*.

2.1 Motivation

Given a n-dimensional vector $\mathbf{x}$, we can define linear basis function models based on linear combination of non-linear basis functions $\phi_j(\mathbf{x})$:

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x})\right) = f\left(\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})\right), \tag{2.1}$$

where f is a non-linear output activation function. Basis functions can take arbitrary forms, which might lead us to believe that such models could solve any classification or regression

problem. In other words, a sufficiently large and diverse set of basis functions could make a linear basis function model capable of approximating any function to an arbitrary level of accuracy. However, a closer look at equation (2.1) reveals a significant drawback: the basis functions are predefined and data-independent. A natural assumption is that larger datasets would require more basis functions, which would also need to be sufficiently flexible. This demands a combination of domain knowledge and trial-and-error, which was a key principle in machine learning for many years. But, as datasets grew larger, this approach became impractical, making it untenable to maintain the independence between basis functions and the dataset. Today, domain knowledge is more relevant in the design of model hyperparameters, while the neural network is tasked with learning the basis functions directly from the data. Nevertheless, as we will see later, the linear basis function model is actually a special case of a neural network.

Even assuming that we have sufficiently efficient basis functions, the problem arises when there are too many of them. Specifically, a linear basis function model can be illustrated by a generalized second-order polynomial:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j.$$

From the above, it is evident that the number of independent coefficients is of the order $O(n^2)$. This concept can be generalized to a polynomial of degree M, resulting in an exponential growth $O(n^M)$ in the number of coefficients. Furthermore, consider the example of a classification problem. Suppose we divide the feature space into a grid by splitting possible values at each coordinate into equidistant intervals, and use a naive algorithm for classification—classifying a given point by identifying its box at the grid and assigning it the class that is most frequent in that box. Therefore, if the box corresponding to the coordinates of x contains 10 members of class A and 5 members of class B, we classify the data x as belonging to class A. The problem with this naive approach is that, for proper classification, as the dimension of the feature space increases, the number of points required grows exponentially, as illustrated in Figure 2.6.

Both examples illustrate the phenomenon known as the *curse of dimensionality*, where increasing dimensions result in computational inefficiency among many other challenges.

2.2 Artificial neuron

An *artificial neuron* (hereinafter referred to as a *neuron*) consists of three key components: linear combination of *weights*, *bias*, and the *activation function*. This can be mathemati-

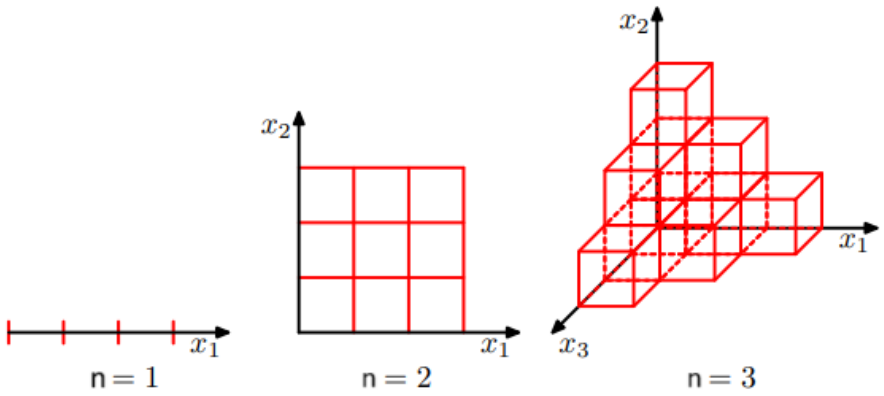


Figure 2.1: The increase in the number of grid elements is exponentially dependent on the dimension n. Intuitively, 10 samples would cover a much larger portion of the space when $n = 1$ than when $n = 3$.

cally expressed as:

$$z = \sum_{i=1}^{n} w_i x_i + b, \tag{2.2}$$

$$a = f(z), \tag{2.3}$$

where $x_1, x_2, \ldots, x_n$ represent the inputs to the neuron, b represents the *bias*, and f is the *activation function* which can be non-linear. Regarding the rest of the terminology, $w_1, w_2, \ldots, w_n$ are called *weights*, the value z is referred to as the *pre-activation*, and the output of the neuron a is called the *activation*. The simple mathematical formulation given by (2.2) and (2.3) has formed the basis of neural network models from the 1960s up to the present day, and can be represented in diagram form as shown in Figure 2.2. The activation function of a neuron is crucial for introducing non-linearity into a neural network. Without it, the network would behave like a linear model, which follows from its definition as the composition of linear functions remains linear. By incorporating non-linear functions, neural networks can capture complex patterns in data, enhancing their ability to learn and model intricate relationships. When it comes to choosing the right activation function, modern deep learning primarily proposes three commonly used options: the sigmoid function, the hyperbolic tangent (tanh), and the rectified linear unit (ReLU). Below are their

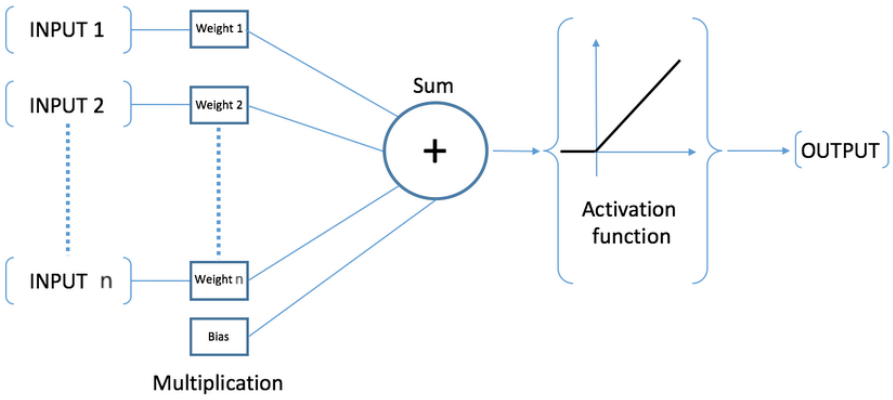


Figure 2.2: Neuron model.

mathematical definitions, presented in the order mentioned:

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}},$$
$$\mathrm{ReLU}(x) = \max(0, x).$$

While the graphs of these activation functions are shown in Figure 2.5, understanding the advantages and disadvantages of each function is essential. These aspects will become clearer after we explain neural network training methods; therefore, further details, along with modifications based on their positive and negative characteristics, are provided in Section 2.6.

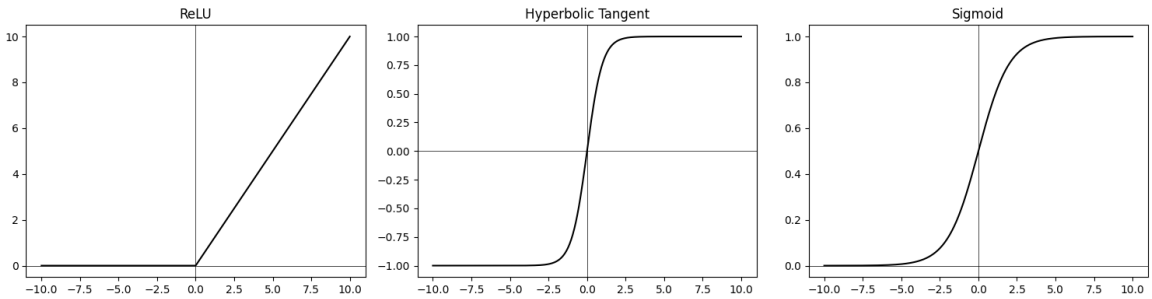


Figure 2.3: Activation functions.

2.3 Neural network

A neural network consists of interconnected layers of neurons that perform transformations on the input data in order to best approximate the elements of the input dataset according to its output value. Regarding the transformation flow of input data, it's important to note that we will be working with *feedforward neural networks*, where connections between the neurons do not form a cycle. This is the simplest form of neural network architecture, where information moves only in a forward direction, sequentially passing through the input, any hidden layers (if they exist), and finally the output layer. Before providing a formal definition, it's essential to define all the mentioned types of layers, specifically the transformations that occur within each group of neurons.

The input layer, as the first layer of the neural network, receives the input vector $\mathbf{x} \in \mathbb{R}^n$ and directly passes it to the next layer without performing any computations or transformations. Next, calculations occur within the hidden layers and the output layer, for which we provide the corresponding mathematical notation. For a layer l with $M^{(l)}$ neurons receiving an input vector $\mathbf{x} \in \mathbb{R}^{N^{(l)}}$, the layer computes an output vector $\mathbf{y}^{(l)} \in \mathbb{R}^{M^{(l)}}$. To calculate each of its component $y_j^{(l)}$, we extend the definition of a single neuron to a set of neurons using matrix notation:

$$z_j^{(l)} = \sum_{i=1}^{N^{(l)}} W_{ji}^{(l)} x_i + b_j^{(l)} \implies \mathbf{z}^{(l)} = W^{(l)}\mathbf{x} + \mathbf{b}^{(l)},$$

where $W^{(l)} \in \mathbb{R}^{M^{(l)} \times N^{(l)}}$ is the weight matrix of the layer, $\mathbf{x} \in \mathbb{R}^{N^{(l)}}$ is the input vector, $\mathbf{b}^{(l)} \in \mathbb{R}^{M^{(l)}}$ is the bias vector, and $\mathbf{z}^{(l)} \in \mathbb{R}^{M^{(l)}}$ is the vector of pre-activation values for the layer. Next, we apply the activation function to each pre-activation value:

$$a_j^{(l)} = f_j^{(l)}(z_j^{(l)}),$$

or in vector form:

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)})$$

where $f^{(l)}$ is the vector of activation functions applied element-wise to the vector $\mathbf{z}^{(l)}$. Thus, the layer transformation can be summarized as:

$$\mathbf{a}^{(l)} = f^{(l)}(W^{(l)}\mathbf{x} + \mathbf{b}^{(l)}),$$

where $\mathbf{a}^{(l)}$ is the output of the layer after applying the activation function to the pre-activation values. This transformation will be written shortly as $f^{(l)}(\mathbf{x}; W^{(l)}, \mathbf{b}^{(l)})$. Using this notation, we can express layer transformations as follows:

- input layer: $f^{(0)}(\mathbf{x}) = \mathbf{x}$,

- hidden layers: $f^{(l)}(\mathbf{x}; W^{(l)}, \mathbf{b}^{(l)}),\ l = 1, 2, ..., L-1,$
- output layer: $f^{(L)}(\mathbf{x}; W^{(L)}, \mathbf{b}^{(L)}),$

where L is the total number of hidden and output layers. It's important to note that the output $\mathbf{a}^{(l)}$ of a layer l serves as part of the input, along with the weight matrix and bias, for the next layer $l + 1$.

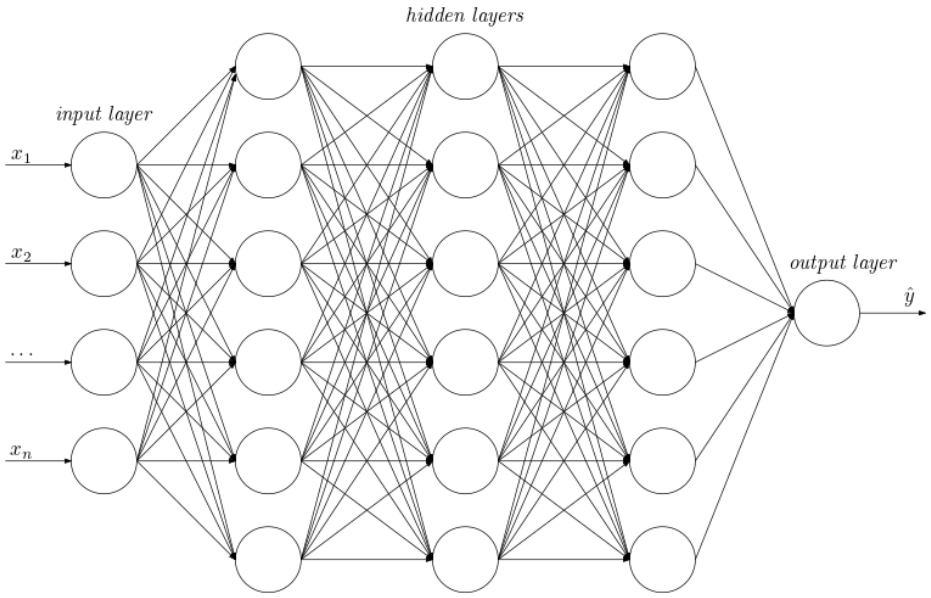


Figure 2.4: A feedforward neural network with an input layer, three hidden layers, and an output layer. Note that the inputs to each neuron come from the outputs of all neurons in the previous layer, although this is not always the case. This type of neural network is called a *fully connected neural network*.

Definition 2.3.1. *A neural network with L layers, where L includes both hidden layers and the output layer, is a function* $f : \mathbb{R}^n \to \mathbb{R}^m$ *defined as:*

$$f(\mathbf{x}) = f^{(L)}(f^{(L-1)}(\dots f^{(2)}(f^{(1)}(f^{(0)}(\mathbf{x}); W_1, \mathbf{b}_1); W_2, \mathbf{b}_2)\dots); W_L, \mathbf{b}_L),$$

where $N^{(0)} = n$, $M^{(l)} = N^{(l-1)}$ *for* $l = 1, \dots, L-1$, *and* $M^{(L)} = m$.

Convolutional neural network

Feedforward neural networks can face challenges with an excessive number of parameters resulting from complex input data, such as images, without additional preprocessing.

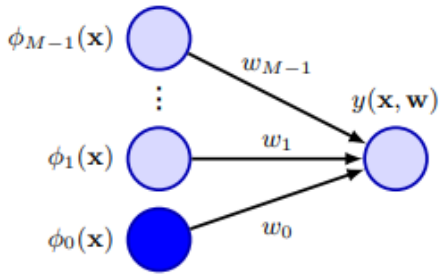


Figure 2.5: By defining f in (2.1) as identity function, we obtain the linear model $y(\mathbf{x}, \mathbf{w})$, which can be expressed with a single layer of parameters in a neural network. In this case, the basis functions are independent of the data because the first layer of neurons remains unchanged during training and parameter optimization. On the other hand, a general neural network incorporates implicit basis functions that are data-dependent and their output for the same input changes during the training as weights and biases are affected.

Convolutional neural networks (CNNs) address this issue by introducing *convolutional* and *pooling* layers before the layers mentioned in the previous section. Motivated by the mathematical concept of convolution, CNNs have been instrumental in driving significant advancements in the field of computer vision, as detailed below.

The convolutional layer is the fundamental building block of a CNN, inspired by the idea that nearby pixels are more strongly correlated than those farther apart. It introduces *filters* or *kernels*, typically represented as $H \times W$ matrices, where H and W are the height and width of the filter, respectively. Each element of these matrices corresponds to a trainable parameter (weight) that is adjusted during the training process. These filters perform element-wise multiplication (Hadamard product) followed by summation with corresponding sections of the input data across its entire spatial dimensions (width and height). This approach significantly reduces the number of parameters and improves computational efficiency, even though the filters themselves consist of learnable parameters updated during training. The output of this layer is referred to as a *feature map*, also known as a *channel*. Empirical evidence shows that deeper convolutional layers recognize more complex patterns, while earlier layers detect simpler ones. A fascinating simulator that visually demonstrates this concept can be found in [8].

Example 2.3.2. *Let us compare the number of parameters in a convolutional layer and a fully connected (FC) layer. A convolutional layer with F filters (kernels), each of size $H \times W$, applied to an input of size $C_{in} \times H_{in} \times W_{in}$ (where C_{in} is the number of input channels, and H_{in}, W_{in} are the height and width of the input) has:*

$$\textit{Parameters (CNN)} = (H \times W \times C_{in} + 1) \times F,$$

where +1 *accounts for the bias for each filter. For* $C_{in} = 3$, $H_{in} = W_{in} = 32$, $H = W = 3$, *and* $F = 64$*:*

$$\textit{Parameters (CNN)} = (3 \times 3 \times 3 + 1) \times 64 = 1,792.$$

An FC layer with input size $C_{in} \times H_{in} \times W_{in}$ *and output size* F *has:*

$$\textit{Parameters (FC)} = (\textit{Input size}) \times (\textit{Output size}) + (\textit{Output size}).$$

For $C_{in} = 3$, $H_{in} = W_{in} = 32$, *and* $F = 64$*:*

$$\textit{Parameters (FC)} = (3 \times 32 \times 32) \times 64 + 64 = 196,736.$$

Thus, the convolutional layer has significantly fewer parameters (1,792) compared to the fully connected layer (196,736).

The pooling layer further simplifies the output from the convolutional layers and reduces the number of parameters by focusing the network on important features. It operates by sliding a window across the feature map and applying a specific operation to each region covered by the window. The most common types are max/min and average pooling. An additional benefit of this layer is translation invariance (or equivariance), which enables the network to recognize an object or feature regardless of its position, orientation, size, etc.

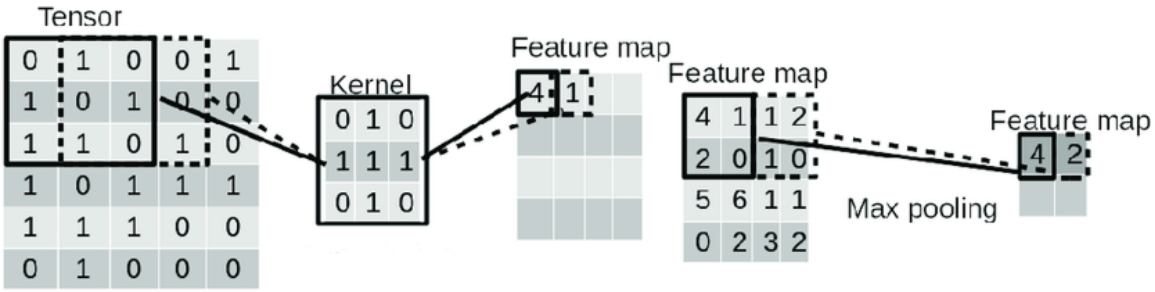


Figure 2.6: Illustration of the use of convolutional and max pooling layers on input data represented as a tensor. The **4** in the feature map is computed by performing element-wise multiplication between the kernel and the top-left 3×3 section of the input tensor, followed by summing the results. Similarly, **1** is obtained by applying the same operation to the next 3×3 section of the input, shifted by one step, and summing the results. After max pooling, the value **4** in the feature map is derived by selecting the maximum value from the 2×2 sliding window applied to the feature map, which covers the region $\{4, 1, 2, 0\}$ in this step.

Alongside convolution, we will use its reverse operation during implementation. Specifically, the flow of our model will involve creating a lower-dimensional latent representation

of the input using convolution. However, it will also be necessary to reconstruct the image from the latent space, which requires increasing the dimensionality of the latent variable. This is achieved using the *transposed convolution operation*, which employs all the techniques seen so far but with the purpose of increasing dimensionality. It is best explained visually, as shown in Figure 2.7.

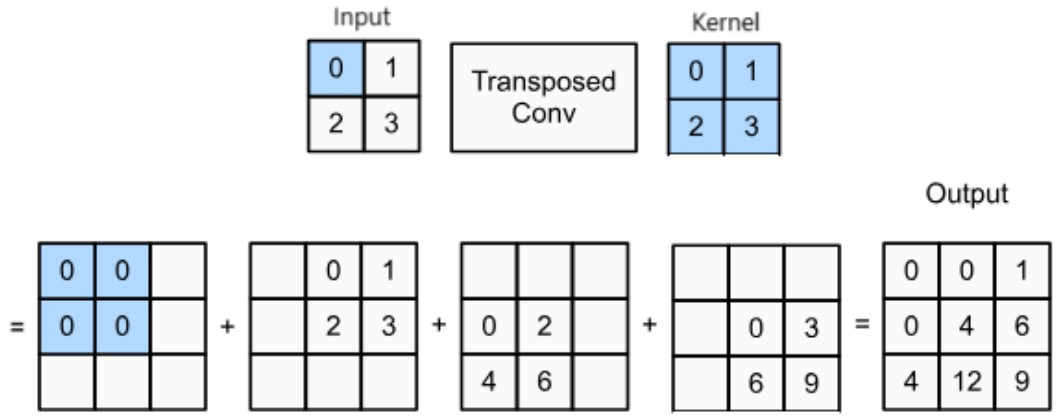


Figure 2.7: Transposed convolution: applies the kernel to the input matrix by sliding and performing element-wise multiplication and summation, producing an upsampled output matrix with larger dimensions.

2.4 Gradient descent

Neural networks can be understood as universal function approximators, capable of approximating complex, non-linear functions that map input data to output data. Accordingly, the network can contain numerous parameters, raising the question of how to optimize them to ultimately become a better approximator. The answer lies in the gradient descent algorithm. Gradient descent is an optimization algorithm used to minimize a function by iteratively moving towards the direction of steepest descent, which can be shown to be in the direction of the gradient but with opposite orientation. The algorithm starts from an initial point (typically random) and updates parameters to move closer to the function's minimum. Let us denote all the parameters of the neural network as $\boldsymbol{\theta} = (\mathbf{w}, \mathbf{b})$, where $\mathbf{w}$ represents the set of all weights, and $\mathbf{b}$ represents the set of all biases in the neural network. The update rule for parameters $\boldsymbol{\theta}$ at iteration t is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t),$$

where η is the learning rate, a hyperparameter that controls the step size for each update, and $\nabla\mathcal{L}(\boldsymbol{\theta}_t)$ is the gradient of the loss function with respect to $\boldsymbol{\theta}$ at iteration t. Each iteration of gradient descent aims to reduce the value of the loss function, ideally leading to a local (or global) minimum. In convex optimization problems, this approach guarantees convergence to the global minimum. However, in neural networks, the loss function is generally non-convex, meaning it has multiple local minima. Despite this, gradient descent often identifies a solution that is sufficiently effective, as many local minima tend to result in comparable performance in practice. This algorithm can be implemented in three variations based on the data size: batch, stochastic, and mini-batch. Batch gradient descent is ideal for smaller datasets that fit entirely into memory. It computes the gradient of the loss function using the entire dataset in each iteration, providing stable and accurate gradient estimates that promote smooth convergence. However, this approach can be computationally expensive and slow for large datasets. Stochastic gradient descent (SGD), on the other hand, is better suited for large datasets or online learning scenarios. Instead of using the entire dataset, SGD computes the gradient using only a single randomly selected data point per iteration. This frequent updating makes SGD computationally efficient and capable of escaping local minima due to its high variance, but this noise can also lead to less stable convergence. Mini-batch gradient descent offers a compromise, processing small batches to reduce memory usage while maintaining more stable convergence than SGD, making it widely used in deep learning applications. In accordance with the above, we provide the formal definition of the aforementioned variant of gradient descent, with a particular emphasis on its application in the context of supervised learning, as this will be our primary focus.

Definition 2.4.1. *Let $\mathcal{L}$ be the loss function, $\boldsymbol{\theta}$ be the vector of values of all parameters of the neural network, and $B = \{(\mathbf{x_i}, \mathbf{y_i}) \mid i = 1, 2, \ldots, n\}$ be a randomly selected subset (mini-batch) of training data, where $\mathbf{x_i}$ is the input to the model and $\mathbf{y_i}$ is the actual output. Then the parameter update for gradient descent with mini-batches is given by:*

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_\theta \left(\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\mathbf{y_i}, \mathbf{x_i}, \boldsymbol{\theta})\right). \tag{2.4}$$

In deep learning, η denotes the *learning rate*. A new mini-batch is randomly selected in each iteration from yet unused data pairs until the entire training set is exhausted, with one complete pass referred to as an *epoch*. The optimization continues until the convergence criteria are met or the maximum number of epochs is reached. The first stopping criterion does not always occur. There are several gradient descent optimization techniques that address the issue of convergence, such as *momentum, Nesterov accelerated gradient, AdaGrad, RMSprop*, and *Adam*. In this thesis, the Adam optimizer will be used, and it will be explained in the following section. For information on other optimization techniques, please refer to Section 7.3 in [2].

The Adam optimizer

The Adam (Adaptive Moment Estimation) optimizer is a widely used optimization algorithm in deep learning, which combines the benefits of two other extensions of stochastic gradient descent (SGD): *AdaGrad* and *RMSProp*. Adam adapts the learning rate for each parameter based on the first and second moments of the gradients, which helps in improving convergence rates, especially in high-dimensional spaces. The key features of Adam include adaptive learning rates, where each parameter has its own learning rate that is dynamically adjusted based on past gradients, and bias correction, which compensates for the initialization bias in the moving averages during the early stages of optimization. So, Adam optimizer maintains two moving averages for each parameter: the first moment estimate m_t (the mean of the gradients) and the second moment estimate v_t (the uncentered variance of the gradients). The first moment helps accelerate in relevant directions, while

Algorithm 2 Adam optimizer

procedure AdamOptimizer
  Input: $\eta, \beta_1, \beta_2, \epsilon$
  Initialize parameters $\boldsymbol{\theta}$
  Initialize $\mathbf{m_0} \leftarrow 0$, $v_0 \leftarrow 0$, and $t \leftarrow 0$
  while not converged **do**
    $t \leftarrow t + 1$
    $\mathbf{g_t} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ ▷ Compute gradient
    $\mathbf{m_t} \leftarrow \beta_1 \mathbf{m_{t-1}} + (1 - \beta_1)\mathbf{g_t}$ ▷ Update first moment
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)\mathbf{g_t} \odot \mathbf{g_t}$ ▷ Update second moment
    $\hat{\mathbf{m}}_\mathbf{t} \leftarrow \frac{\mathbf{m_t}}{1-\beta_1^t}$ ▷ Bias-corrected first moment
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ ▷ Bias-corrected second moment
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{\mathbf{m}}_\mathbf{t}$ ▷ Update parameters
  end while
end procedure

where η is the learning rate, β_1 and β_2 are decay rates (commonly set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$), ϵ is a small constant to prevent division by zero, and $\odot$ is Hadamard product.

the second moment stabilizes updates by accounting for the variability of gradients. This balance makes Adam particularly adept at managing *sparse gradients*, characterized by numerous zero or near-zero values in the gradient of the loss function, thereby facilitating faster convergence in a wide range of deep learning scenarios. Large gradients can cause unstable updates, while very small gradients can stall progress. The variance-based adjustment helps control these extremes by moderating the impact of outlier gradients, which is particularly useful in deep networks prone to vanishing or exploding gradient issues. Pseudocode is provided in Algorithm 2.

2.5 Backpropagation

In the previous section, we saw that finding the minimum of the loss function requires its gradient. However, this calculation can be quite demanding if approached naively, using a brute-force algorithm that follows its mathematical definition. Backpropagation is an extremely efficient way to calculate the gradient, leveraging the structure of the network and the *chain rule*. Therefore, this algorithm requires activation functions to be differentiable. However, the ReLU function is not differentiable at 0. Despite this, the non-differentiability at a single point has not proven to be problematic in practice. This situation rarely occurs, and the gradient is typically defined to be either 0 or 1 at that point.

The objective of backpropagation is to compute the gradient of the loss function using the *chain rule*, specifically to determine the partial derivatives:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}, \quad \frac{\partial \mathcal{L}}{\partial b_i^{(l)}}, \tag{2.5}$$

where $\mathcal{L}$ represents the cost function, $b_i^{(l)}$ is the bias of the i-th neuron in layer l, and $w_{ij}^{(l)}$ is the weight factor by which the activation value of the j-th neuron in layer $(l-1)$ enters the i-th neuron in layer l. The activation value of the i-th neuron in layer l is denoted by $a_i^{(l)}$. Furthermore, $a_i^{(l)}$ is related to the activation values of layer $(l-1)$ through the equation:

$$a_i^{(l)} = f^{(l)}\left(\sum_j w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}\right). \tag{2.6}$$

Equation (2.6) can be rewritten in matrix form:

$$\mathbf{a}^{(l)} = f^{(l)}\left(W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right),$$

where $W^{(l)} = (w_{ij}^{(l)})$ represents the weight matrix of layer l, $\mathbf{b}^{(l)}$ is the bias vector of layer l, and $\mathbf{a}^{(l)}$ is the activation vector of layer l.

To calculate the required partial derivatives, we introduce an intermediate variable

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}}$$

which we call the *error* of the i-th neuron in layer l, in accordance with the previous paragraph. Backpropagation provides us with a recursive way of calculating $\delta_i^{(l)}$ in terms of $\delta_i^{(l+1)}$ using the *chain rule*. First we compute the error at the neurons in the last (output) layer L:

$$\delta_i^{(L)} = \frac{\partial \mathcal{L}}{\partial z_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(L)}} (f^{(L)})'(z_i^{(L)})$$

or in vector form,

$$\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{a}}\mathcal{L} \odot (f^{(l)})'(\mathbf{z}^{(L)}), \tag{2.7}$$

where $\odot$ represents the Hadamard product $(A \odot B)_{ij} = (A)_{ij} \cdot (B)_{ij}$, and $\nabla_{\mathbf{a}}\mathcal{L}$ is the vector of partial derivatives $\frac{\partial \mathcal{L}}{\partial a_i^{(L)}}$. Now we want to calculate the error $\delta_i^{(l)}$ at the neurons in layer l in terms of the error in layer $l+1$:

$$\begin{aligned}
\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} &= \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} \\
&= \sum_j \delta_j^{(l+1)} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} \\
&= \sum_j \delta_j^{(l+1)} \frac{\partial}{\partial z_i^{(l)}} \left(\sum_k w_{jk}^{(l+1)} f^{(l)}(z_k^{(l)}) + b_j^{(l+1)} \right) \\
&= \sum_j w_{ji}^{(l+1)} \delta_j^{(l+1)} (f^{(l)})'(z_i^{(l)}).
\end{aligned}$$

Declaring $\delta_j^{(l)}$ as a component of the vector $\boldsymbol{\delta}^{(l)}$, we get:

$$\boldsymbol{\delta}^{(l)} = \left(W^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)} \odot (f^{(l)})'(\mathbf{z}^{(l)}). \tag{2.8}$$

Using equations (2.7) and (2.8) we can determine the error at each neuron in the network, which is the primary objective of backpropagation (2.5). By applying the *chain rule*, we obtain

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)},$$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}.$$

In matrix form, this becomes

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \quad \frac{\partial \mathcal{L}}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)} \left(\mathbf{a}^{(l-1)}\right)^T.$$

Here, $\mathbf{a}^{(l-1)}$ denotes the activation vector of the neurons in layer $(l-1)$, and $(\boldsymbol{\delta}^{(l)})^T$ is the transposed error vector for the neurons in layer l. Now we have all the neccessary equations for the backpropagation described in Algorithm 3. A key feature of Algorithm 3 is its ability to efficiently compute all partial derivatives through a two-phase process: an initial forward pass, where inputs are propagated through the network, followed by a backward pass, where the model weights are updated based on the backward propagation of the loss function's error.

Algorithm 3 Backpropagation in neural networks

procedure BACKPROPAGATION
  Input: $W^{(l)}$, $\mathbf{b}^{(l)}$, activation function $f^{(l)}$ and its derivative $(f^{(l)})'$, for $l = 1, 2, \ldots, L$
  Forward Propagation:
  for $l = 1$ to L **do**
    $\mathbf{z}^{(l)} \leftarrow W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$ ▷ Compute weighted sum
    $\mathbf{a}^{(l)} \leftarrow f^{(l)}(\mathbf{z}^{(l)})$ ▷ Apply activation function
  end for
  Backward Propagation:
  Compute the output layer gradient: $\delta_L = \nabla_{\mathbf{a}}\mathcal{L} \odot (f^{(L)})'(\mathbf{z}^{(L)})$
  for $l = L - 1$ to 1 **do**
    $\boldsymbol{\delta}^{(l)} \leftarrow (W^{(l+1)})^T\boldsymbol{\delta}^{(l+1)} \odot (f^{(l)})'(\mathbf{z}^{(l)})$ ▷ Backpropagate the error
  end for
  Gradient Computation:
  for $l = 1$ to L **do**
    $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} \leftarrow \boldsymbol{\delta}^{(l)}$ ▷ Compute gradient for biases
    $\frac{\partial \mathcal{L}}{\partial W^{(l)}} \leftarrow \mathbf{a}^{(l-1)}(\boldsymbol{\delta}^{(l)})^T$ ▷ Compute gradient for weights
  end for
end procedure

2.6 Challenges

In the initialization phase of the backpropagation algorithm (Algorithm 3), it is crucial to set initial values for the weights and select an activation function for each layer. The choice of these components is interconnected and presents a non-trivial challenge, ultimately determining the quality of the network that emerges after training.

Activation functions

After deriving all the necessary equations for the backpropagation algorithm, it becomes evident that we need to compute the derivatives of the activation functions. Therefore, it is crucial to examine these functions' behavior and, based on this analysis, identify potential numerical issues that may arise. *The vanishing gradient problem* refers to the issue where the gradients of the activation function become very small during backpropagation. This can prevent the network from learning effectively. It happens when the derivative of the activation function approaches zero for large positive or negative inputs, as is the case with sigmoid and tanh derivatives which can be seen on Figure 2.6. Such functions are often described as *saturating* around certain values, leading to slow or unstable learning because the neural network becomes insensitive to input changes.

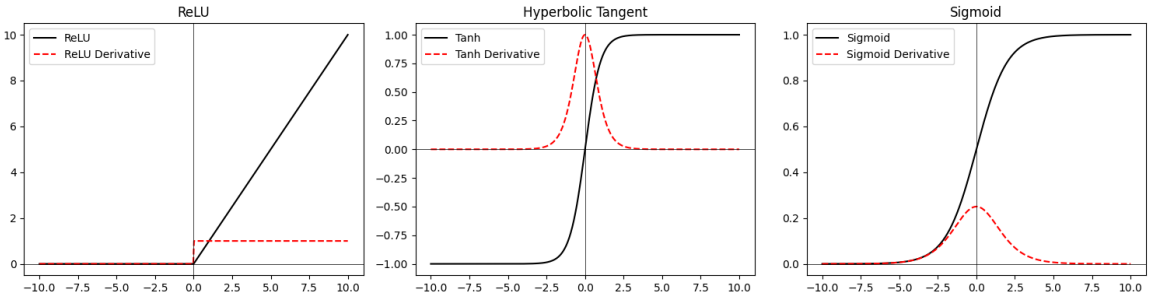


Figure 2.8: Activation functions and their derivatives.

While the sigmoid and tanh functions have certain drawbacks, such as vanishing gradients and saturation, they continue to be useful in specific scenarios due to their unique properties. For example, the sigmoid function outputs values in the range $[0, 1]$, which is particularly beneficial for tasks like binary classification or when activations are required to represent probabilities. This makes the sigmoid function especially valuable in the output layer of a network, where we need a probabilistic interpretation. In contrast, the tanh function produces outputs in the range $[-1, 1]$, which are zero-centered. This characteristic is advantageous in the sense that it reduces the risk of biased gradient updates towards the positive values, which is particularly important for the hidden layers. Moreover, both activation functions introduce essential non-linearity into the network, along with their smooth and differentiable behavior which also makes them mathematically convenient, aiding in gradient-based optimization. Although they face the vanishing gradient problem, sigmoid and tanh functions can still be useful in shallow networks, where this issue is less pronounced. Finally, it's also important to note that many pre-trained models and legacy systems rely on sigmoid and tanh due to their historical prevalence. Transitioning to alternatives such as ReLU may not always be feasible, especially when compatibility with these existing systems is a concern.

The ReLU function clearly avoids the problems of vanishing gradients and saturation, but it faces the problem of *dead neurons*. Dead neurons refer to neurons that consistently output zero or a constant value for all inputs, regardless of the learning process. This occurs when a neuron's activation function becomes trapped in a state where its output no longer changes, effectively removing its contribution to the learning process. This drawback of the ReLU function has been mitigated by its modifications, which can be seen in the Table 2.1. However, ReLU is widely used due to its simplicity and effectiveness. Some of its beneficial characteristics include enabling sparse activation within a neural network, where only a subset of neurons are active at any given time.

So far, we have observed that activation functions face challenges such as vanishing

Activation Function	**Formula**	**Description**
Leaky ReLU	$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0, \\ \alpha x, & \text{if } x < 0 \end{cases}$	Allows a small, non-zero gradient α for negative values, avoiding dead neurons.
Parametric ReLU (PReLU)	$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0, \\ \alpha x, & \text{if } x < 0 \end{cases}$	Similar to Leaky ReLU, but α is a learned parameter during training.
Exponential Linear Unit (ELU)	$\text{ELU}(x) = \begin{cases} x, & \text{if } x \geq 0, \\ \alpha(\exp(x) - 1), & \text{if } x < 0 \end{cases}$	Smooth transition between positive and negative values, controlled by α, which improves gradient flow.

Table 2.1: Variations of ReLU activation functions.

gradients, saturation, and dead neurons. While dead neurons can be addressed through modifications of the ReLU function, weight initialization techniques help mitigate the issues of vanishing gradients and saturation.

Weight initialization

Improper weight initialization can significantly damage the neural network training process. If weights are too small, it can lead to vanishing gradients, where the network fails to learn effectively, particularly in deep networks. On the other hand, large weights can cause exploding gradients, leading to instability and diverging values during training. In the case of activation functions like ReLU, poor initialization can result in dead neurons. Additionally, poorly initialized weights can cause slow convergence, prevent effective optimization, and lead to the symmetry problem, where neurons in the same layer learn the same features. Proper weight initialization techniques, such as Xavier or He initialization, are crucial to overcome these issues and ensure efficient network training. Interested reader can find more in papers [3], [6] and [11]. The first paper [3] examines recent advancements in techniques to improve model performance, stability, and convergence, particularly in remote sensing applications. Kumar's work [11] develops proper weight initialization theory with non-linear activations, while Glorot and Bengio [6] discuss the importance of initialization schemes that maintain the variance of activations and gradients across layers, addressing

issues like saturation and improving training efficiency in deep neural networks.

Chapter 3

Variational autoencoder

An *auto-associative neural network*, or *autoencoder*, is a model designed to learn internal representations of data by reconstructing its input. It has the same number of output units as input units and is trained to generate an output $\tilde{\mathbf{x}}$ that closely matches the input $\mathbf{x}$. The network consists of two components: an *encoder*, which maps the input $\mathbf{x}$ to a hidden representation $\mathbf{z}(\mathbf{x})$ within a latent space of significantly lower dimensionality than the original feature space, and a *decoder*, which maps this representation back to the output $\tilde{\mathbf{x}}(\mathbf{z})$. Once trained, the encoder's output $\mathbf{z}(\mathbf{x})$ serves as a compact, low-dimensional representation of the input, which is later used for generating new data. We will first introduce the *deterministic autoencoder*, a straightforward approach focused on direct reconstruction of the input, and discuss its properties. While simple autoencoders are rarely used directly in modern deep learning — due to their inability to provide semantically meaningful latent representations or to generate new examples from the data distribution — they form an important conceptual foundation for more advanced generative models. Building on this foundation, we will explore the *variational autoencoder (VAE)* — a latent variable model that extends the autoencoder into a probabilistic framework and serves as a central focus of this thesis. VAEs address the limitations of deterministic autoencoders by structuring the latent space probabilistically, enabling both meaningful representations and powerful generative capabilities. This is done by learning an encoder distribution $q(\mathbf{z}|\mathbf{x})$ together with a decoder distribution $p(\mathbf{x}|\mathbf{z})$. The key differences between these two approaches, such as deterministic versus probabilistic latent spaces, will be discussed in detail.

3.1 Deterministic autoencoder

Autoencoders extend classical dimensionality reduction techniques, such as *principal component analysis (PCA)*. In PCA, a linear transformation maps the input data onto a lower-dimensional manifold, which can then be approximately reconstructed back to the original

data space using another linear transformation. For further details on PCA, see [2, Section 16.1], although the main ideas have already been outlined in the geometric interpretation of the Mahalanobis distance, where we defined the principal components. Autoencoders extend PCA by leveraging neural network non-linearity to model complex, nonlinear data subspaces, where the encoder compresses the input into a latent representation, and the decoder reconstructs it by optimizing network weights to minimize reconstruction error over the dataset. This framework not only extends PCA to non-linear transformations but also serves as a foundational approach for learning efficient data representations in modern machine learning. In the following, we present some of the most common types of simple autoencoders.

Linear autoencoders

Linear autoencoders perform dimensionality reduction by projecting data onto a linear subspace, similar to PCA. The network maps input vectors onto themselves, learning an *auto-associative mapping* that minimizes the sum-of-squares reconstruction error:

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{n=1}^{N} \|f(\mathbf{x_n}, \boldsymbol{w}) - \mathbf{x_n}\|^2.$$

Despite using neural networks, linear activation functions restrict the representation to linear mappings. It can be shown that such autoencoders are equivalent to PCA when they consist of a single hidden layer and the number of hidden units matches the number of principal components to be extracted.

Deep autoencoders

Deep autoencoders extend the linear autoencoder framework by incorporating non-linear layers. A typical architecture consists of multiple layers, with non-linear activation functions such as sigmoid or ReLU applied in the hidden layers. The network maps the input data through successive nonlinear transformations, enabling the latent representation to capture more complex, non-linear relationships in the data:

$$\underbrace{F_1 : \text{Input} \to \text{Latent space}}_{\text{encoder}}, \quad \underbrace{F_2 : \text{Latent space} \to \text{Output}}_{\text{decoder}}.$$

This flexibility allows deep autoencoders to generalize beyond linear transformations, effectively performing a non-linear form of PCA.

Sparse autoencoders

Sparse autoencoders introduce a regularization term to encourage sparsity in the latent representation. For instance, an L_1 regularizer is added to the reconstruction error to enforce a sparse activation of the hidden units:

$$\tilde{E}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \|f(\mathbf{x_n}, \boldsymbol{w}) - \mathbf{x_n}\|^2 + \lambda \sum_{k=1}^{K} |z_k|,$$

where z_k are the activations of the hidden units. Sparsity helps the model learn a compressed representation with fewer active features, leading to better interpretability and reduced dimensionality. Sparse autoencoders are particularly useful for feature selection and representation learning.

Denoising autoencoders

Denoising autoencoders aim to learn robust representations by reconstructing input data from a corrupted version. The input data $\mathbf{x_n}$ is perturbed to produce a noisy version $\tilde{\mathbf{x}}_\mathbf{n}$, and the autoencoder is trained to minimize the reconstruction error between the clean input $\mathbf{x_n}$ and the reconstructed output from its noisy version:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \|f(\tilde{\mathbf{x}}_\mathbf{n}, \boldsymbol{w}) - \mathbf{x_n}\|^2.$$

By learning to denoise the input, the model captures meaningful structures in the data. Denoising autoencoders are effective in discovering dependencies and robustly encoding features, even in noisy or incomplete datasets.

Applications

As previously mentioned, simple autoencoders are rarely utilized in modern deep learning because they primarily focus on minimizing reconstruction loss and data compression, often failing to capture meaningful relationships within the data. However, sparse autoencoders have gained popularity due to the emergence of *large language models (LLMs)*. LLMs are highly complex, consisting of a vast number of parameters, making meaningful interpretation challenging. Let's suppose we want to interpret an LLM based on sentences that differ in tense. For example, we can use the LLM to generate a dataset of such sentences and record the hidden states during their generation. These states can then train a sparse autoencoder, which creates a latent representation while minimizing the number of activated neurons due to its inherent design. As a result, we might find that a specific

neuron h in the latent layer activates most strongly for sentences in the past tense. By identifying the hidden states that most influence this neuron, we can pinpoint the parts of the LLM responsible for generating such sentences, providing valuable insights for fine-tuning the model. For further research and a simulation of the example above, refer to [4].

3.2 Architecture

In the following, we explain the structure of the variational autoencoder. If we examine deterministic autoencoders, we see that they focus solely on minimizing the reconstruction loss. The main issue with this approach is the lack of generative capability, as random sampling from the latent space does not produce meaningful outputs. The variational autoencoder seeks to address this by introducing, in addition to the reconstruction loss, a penalty for the lack of compactness in the latent representation in the form of KL divergence. This additional metric encourages the model to learn a compact and dense latent space, which remains meaningful due to the reconstruction loss. As we have previously mentioned, the encoder learns the latent space as a distribution (later we will explain why neural networks are suitable for this task). In line with the Bayesian approach, the question arises of which distribution the encoder should target. Specifically, it involves determining which distribution of the latent variables should be used as the prior and subsequently measured against the current encoder distribution using KL divergence. In practice, this prior distribution p is most commonly chosen as a zero-mean, unit-variance Gaussian $p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid \mathbf{0}, \mathbf{I})$. Additionally, we observe that if the loss function contained only the KL divergence, the encoder would indeed learn a compact representation, but it would not be meaningful. The relationship between the three cases of the loss function is illustrated in Figure 3.2. In the context of neural networks, the encoder is often referred to as the *inference model*. The decoder models the distribution $p_{\theta}(\mathbf{x} \mid \mathbf{z}) = p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the parameters of the neural network, and its output can be the parameters of a Gaussian distribution conditioned on $\mathbf{z}$ and $\boldsymbol{\theta}$. For simplicity, suppose that both x and z are scalars. The output is then calculated as:

$$\ln p(x \mid z, \boldsymbol{\theta}) = -\frac{1}{2} \ln (2\pi\sigma) - \frac{(x-\mu)^2}{2\sigma^2},$$

where μ and σ represent the decoder's output for a given z and $\boldsymbol{\theta}$.

The development of our model typically starts in the frequentist direction with the maximization of the log-likelihood expectation:

$$p(\mathcal{D} \mid \boldsymbol{\theta}) = \sum_{n=1}^{N} \ln p(\mathbf{x_n} \mid \boldsymbol{\theta})$$

under the assumption that the data are independent and identically distributed. However, it will be shown that this expression in our model is not computable due to the introduction of

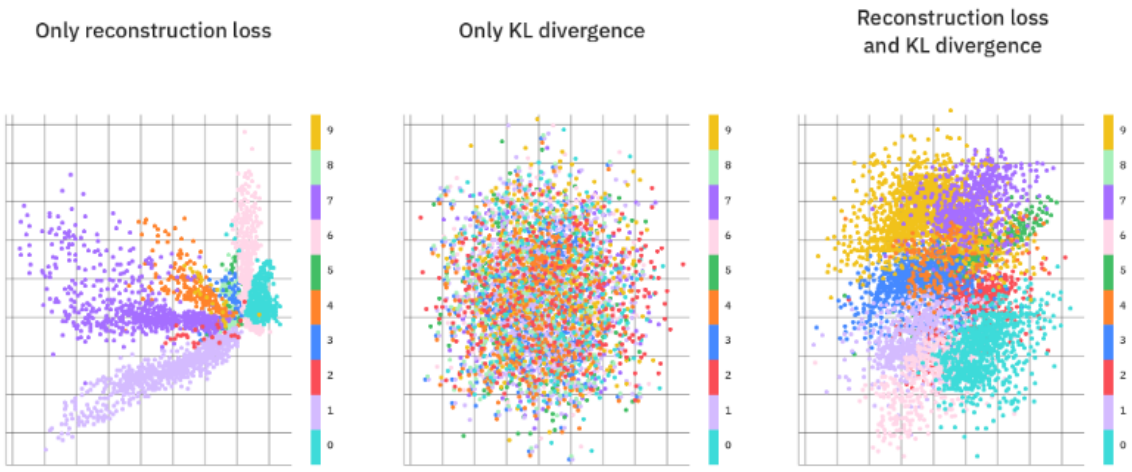


Figure 3.1: We examine the latent representation of the MNIST dataset, consisting of black-and-white images of digits, in relation to the loss function. In the first case, where only the reconstruction loss is used, we observe a high probability of selecting meaningless outputs when randomly sampling. In the second case, where only the KL divergence is used, we see a compact distribution, but without clear boundaries between classes, making it uninformative. Naturally, the optimal case is represented in the third image.

latent variables, which implement the Bayesian approach. Nevertheless, the problem will be solved using the *amortized (variational) inference*, a technique in Bayesian statistics for approximating complex probability distributions, which turns inference problems into optimization problems. To achieve this, we will begin by reformulating the maximum log-likelihood in a form that is amenable to optimization.

To summarize, the architecture of a variational autoencoder consists of two neural networks that generate parameterized distributions. A latent variable model $p_\theta(\mathbf{x}, \mathbf{z})$, whose distribution is parameterized by a neural network, is referred to as a *deep latent variable model (DLVM)*. The most common form of such models, which we will also employ, is represented as follows:

$$p_\theta(\mathbf{x}, \mathbf{z}) = \underbrace{p_\theta(\mathbf{z})}_{\text{Gaussian prior}} \underbrace{p_\theta(\mathbf{x} \mid \mathbf{z})}_{\text{decoder}}, \tag{3.1}$$

where $p_\theta(\mathbf{z})$ and $p_\theta(\mathbf{x} \mid \mathbf{z})$ are predefined. A major advantage of these models is that the marginal likelihood $p_\theta(\mathbf{x})$ can exhibit significant complexity, even when each factor in (3.1) is simple. This makes them powerful tools for estimating complex distributions $p^*(\mathbf{x})$.

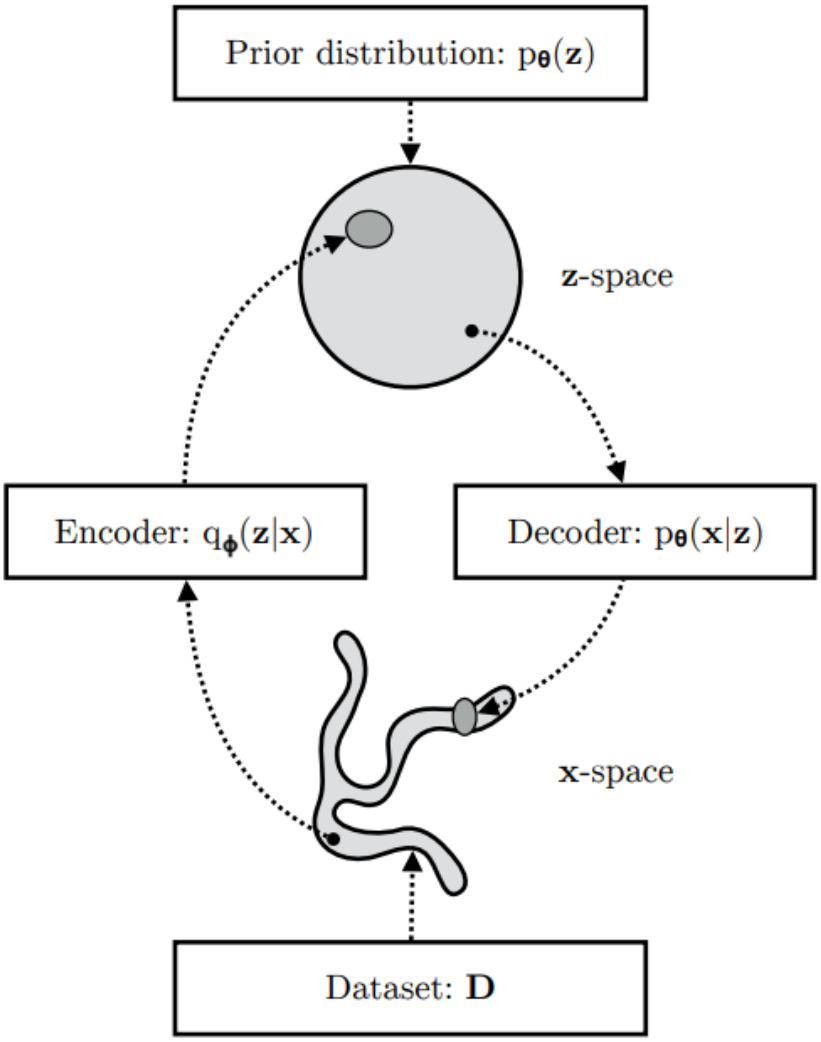


Figure 3.2: The architecture of a variational autoencoder (VAE). We observe a complex and irregular data-generating distribution compared to a much simpler prior distribution $p_\theta(\mathbf{z})$ in the latent space. The generative model defines a joint distribution $p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{z})p_\theta(\mathbf{x} \mid \mathbf{z})$ where $p_\theta(\mathbf{z})$ is the prior distribution over the latent space, and $p_\theta(\mathbf{x} \mid \mathbf{z})$ is the decoder. The encoder $q_\phi(\mathbf{z} \mid \mathbf{x})$, also known as the inference model, approximates the true but intractable posterior $p_\theta(\mathbf{z} \mid \mathbf{x})$ of the generative model. The previously mentioned statements and notations will be introduced later in the chapter; they are presented here for easier reference and understanding.

3.3 Evidence lower bound (ELBO)

Recall that the likelihood function for the latent-variable model is given by:

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \int p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})\, d\mathbf{z} = \int p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z} \mid \boldsymbol{\theta})\, d\mathbf{z}, \tag{3.2}$$

in which $p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$ is defined by decoder neural network. Due to the complex transformations from the neural network, the integral in (3.2) becomes difficult to compute and lacks a closed-form solution. As a result, we rely on numerical approximations, but traditional grid-based methods like Trapezoidal or Simpson's Rule face challenges in high dimensions due to the curse of dimensionality. As the number of dimensions increases, the computational resources required to improve precision and reduce variance grow exponentially, making these methods infeasible for higher-dimensional problems.

To overcome these difficulties, we reformulate equation (3.2) as an optimization problem, seeking an extremum. This approach is advantageous due to the availability of efficient algorithms for solving such problems, providing a computationally feasible solution.

Let q be an arbitrary density function over the latent space. Then, the following holds:

$$\begin{aligned}
\ln p_{\boldsymbol{\theta}}(\mathbf{x}) &= \mathbb{E}_{q(\mathbf{z})}[\ln p_{\boldsymbol{\theta}}(\mathbf{x})] \\
&= \mathbb{E}_{q(\mathbf{z})}\left[\ln \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{q(\mathbf{z})}\left[\ln \left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \frac{q(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right] \\
&= \mathbb{E}_{q(\mathbf{z})}\left[\ln \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} + \ln \frac{q(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right] \\
&= \mathbb{E}_{q(\mathbf{z})}\left[\ln \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})}\right] + \underbrace{\mathbb{E}_{q(\mathbf{z})}\left[\ln \frac{q(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right]}_{D_{\mathrm{KL}}(q(\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))}.
\end{aligned}$$

Thus, the log-likelihood can be expressed as:

$$\ln p_{\boldsymbol{\theta}}(\mathbf{x}) = \mathcal{L}(q, \boldsymbol{\theta}) + D_{\mathrm{KL}}(q(\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})), \tag{3.3}$$

where

$$\mathcal{L}(q, \boldsymbol{\theta}) = \int q(\mathbf{z}) \ln \left(\frac{p(\mathbf{x}, \mathbf{z} \mid \boldsymbol{\theta})}{q(\mathbf{z})}\right).$$

The Kullback-Leibler divergence is non-negative, so $\ln p(\mathbf{x} \mid \boldsymbol{\theta}) \geq \mathcal{L}(q, \boldsymbol{\theta})$. Therefore, the quantity $\mathcal{L}$ is called the *evidence lower bound* (ELBO), also known as the *variational lower bound*. Additionally, we observe that increasing the ELBO reduces the KL divergence of

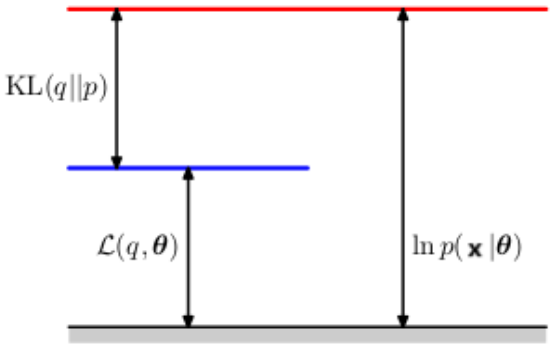


Figure 3.3: The decomposition of the log-likelihood into the KL divergence and ELBO.

the approximation q from the true posterior distribution, thereby improving the accuracy of q.

Assume that the sample set $\mathcal{D} = \{\mathbf{x_1}, \ldots, \mathbf{x_N}\}$ for training is independent and identically distributed. Then, from (3.3), we obtain the log-likelihood function for this dataset:

$$\ln p(\mathcal{D} \mid \boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}_n + \sum_{n=1}^{N} \mathrm{KL}\left(q_n(\mathbf{z}_n) || p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta})\right) \tag{3.4}$$

where

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

The motivation for choosing the distribution q in (3.4) comes directly from the expectation-maximization algorithm. To mimic the E-step, we should define $q(\mathbf{z}_n) = p(\mathbf{z}_n \mid \mathbf{x}_n, \boldsymbol{\theta})$:

$$\ln p_{\boldsymbol{\theta}}(\mathbf{x_n}) = \mathcal{L}(p_{\boldsymbol{\theta}}(\mathbf{z_n}|\mathbf{x_n}), \boldsymbol{\theta}) + D_{\mathrm{KL}}(p_{\boldsymbol{\theta}}(\mathbf{z_n}|\mathbf{x_n}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z_n}|\mathbf{x_n})) = \mathcal{L}(p_{\boldsymbol{\theta}}(\mathbf{z_n}|\mathbf{x_n}), \boldsymbol{\theta})$$

However, the exact posterior distribution of $\mathbf{z}_n$:

$$p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta}) = \frac{p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta}) p(\mathbf{z}_n)}{p(\mathbf{x}_n|\boldsymbol{\theta})}$$

introduces intractable likelihood function in the denominator. Therefore we need to find an approximation to the posterior distribution. Another problem is that the log-likelihood function (3.4) introduces a separate latent variable $\mathbf{z_n}$ for each sample $\mathbf{x_n}$, with its own distribution $q(\mathbf{z_n})$ that could be numerically optimized separately. However, this is computationally expensive, particularly for large datasets.

3.4 Amortized inference

In the previous section, we observed the issues related to the posterior distribution in terms of computational cost. Now, we aim to amortize this cost across the entire dataset using an approximation distribution. Instead of learning the distribution $p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta})$ for each sample individually, we will define a shared distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$, which will be approximated by a neural network—the encoder. This leads us to the final structure of the variational autoencoder, where the first neural network learns the latent representation of the samples, and the second one attempts to reconstruct the input based on this compressed information.

A common approach for defining the encoder is to assume a Gaussian distribution with a diagonal covariance matrix. The parameters of this distribution, namely the mean μ_j and variance σ_j^2, are determined as outputs of a neural network that takes $\mathbf{x}$ as input. The distribution can be expressed as:

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

where M denotes the dimension of the latent space. Note that the means $\mu_j(\mathbf{x}, \boldsymbol{\phi})$ can take any finite value, providing flexibility in the choice of activation functions for the output neurons. In contrast, the variances $\sigma_j^2(\mathbf{x}, \boldsymbol{\phi})$ are required to be non-negative, which is typically ensured by using an exponential activation function.

We observed that the E-step in the EM algorithm is achieved by choosing $q(\mathbf{z}) = p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta}^{\text{old}})$. However, we noted that instead of computing the exact posterior distribution, we approximate it with $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$ using a neural network. As a result, optimizing the loss function with respect to $\boldsymbol{\phi}$ generally does not reduce the KL divergence to zero. Instead, there will always be a residual gap between the true log-likelihood and the ELBO. This is due to several factors. While neural networks are capable of modeling highly complex distributions, we cannot expect them to represent the posterior distribution exactly for several reasons: the true conditional posterior is generally not a factorized Gaussian, neural networks have their own inherent limitations, and the training process itself provides only an approximate solution to the optimization problem.

By incorporating two neural networks with parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, it follows that the loss function of the variational autoencoder will depend on these parameters. In the following, we will highlight the challenge of performing backpropagation with respect to $\boldsymbol{\phi}$ due to the stochastic nature of selecting the latent representation of the input. This challenge is addressed using the *reparameterization trick*, which enables gradient-based optimization despite the stochasticity.

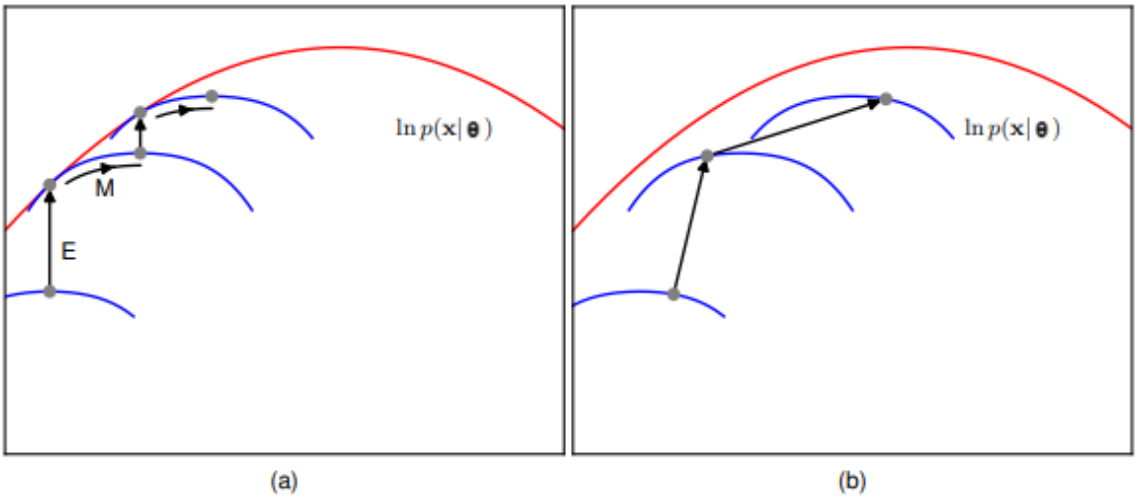


Figure 3.4: Comparison of the EM algorithm with ELBO optimization. We observe that by optimizing the encoder parameters, a residual gap remains between the log-likelihood and the ELBO. In contrast, in the EM algorithm, this is not the case because the posterior distribution is exact, not approximated. Additionally, we note that the M-step corresponds to optimizing the decoder parameters.

3.5 Reparametrization trick

Contribution of each sample $\mathbf{x_n}$ to the loss function (3.4) is given by:

$$\begin{aligned}\mathcal{L}_n(\boldsymbol{\phi}, \boldsymbol{\theta}) &= \int q(\mathbf{z_n}|\mathbf{x_n}, \boldsymbol{\phi}) \ln \left\{ \frac{p(\mathbf{x_n}|\mathbf{z_n}, \boldsymbol{\theta}) p(\mathbf{z_n})}{q(\mathbf{z_n}|\mathbf{x_n}, \boldsymbol{\phi})} \right\} \mathbf{dz_n} \\ &= \underbrace{\int q(\mathbf{z_n}|\mathbf{x_n}, \boldsymbol{\phi}) \ln p(\mathbf{x_n}|\mathbf{z_n}, \boldsymbol{\theta}) \, \mathbf{dz_n}}_{\text{reconstruction loss}} - \underbrace{\mathrm{KL}(q(\mathbf{z_n}|\mathbf{x_n}, \boldsymbol{\phi}) \| p(\mathbf{z_n}))}_{\text{regularization term}}. \end{aligned} \tag{3.5}$$

The KL divergence between two multivariate Gaussian distributions $q(\mathbf{z}) = \mathcal{N}(\boldsymbol{\mu}_q, \boldsymbol{\Sigma}_q)$ and $p(\mathbf{z}) = \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ is given by:

$$\mathrm{KL}(q\|p) = \frac{1}{2}\left(\ln \frac{\det \boldsymbol{\Sigma}_q}{\det \boldsymbol{\Sigma}_p} - d + \mathrm{tr}\left(\boldsymbol{\Sigma}_p^{-1}\boldsymbol{\Sigma}_q\right) + (\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)^\top \boldsymbol{\Sigma}_p^{-1}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q) \right), \tag{3.6}$$

where d is the dimensionality of the random variable. The proof of the previous statement is highly technical and will not be provided here. However, it can be found in detail in [7]. According to the assumptions of our model, we have $q(\mathbf{z}_n \mid \mathbf{x}_n, \boldsymbol{\phi}) = \mathcal{N}(\boldsymbol{\mu}(\mathbf{x}_n, \boldsymbol{\phi}), \mathrm{diag}(\boldsymbol{\sigma}^2(\mathbf{x}_n, \boldsymbol{\phi})))$

and $p(\mathbf{z}_n) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. By substituting those Gaussian distributions into (3.6), we obtain:

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

So, we found the closed-form solution of the second term in the (3.5). The first term can be approximated with a simple Monte Carlo estimator:

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_n|\mathbf{x}_n)}[\ln p(\mathbf{x}_n \mid \mathbf{z}_n, \boldsymbol{\theta})] = \int q(\mathbf{z}_n \mid \mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{x}_n \mid \mathbf{z}_n, \boldsymbol{\theta}) \, \mathrm{d}\mathbf{z}_n \approx \frac{1}{L} \sum_{l=1}^{L} \ln p(\mathbf{x}_n \mid \mathbf{z}_n^{(l)}, \boldsymbol{\theta}).$$

This is easily differentiated with respect to $\boldsymbol{\theta}$:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_n|\mathbf{x}_n)}[\ln p(\mathbf{x}_n \mid \mathbf{z}_n, \boldsymbol{\theta})] &= \nabla_{\boldsymbol{\theta}} \left(\int q(\mathbf{z}_n \mid \mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{x}_n \mid \mathbf{z}_n, \boldsymbol{\theta}) \, \mathrm{d}\mathbf{z}_n \right) \\ &= \int q(\mathbf{z}_n \mid \mathbf{x}_n, \boldsymbol{\phi}) \nabla_{\boldsymbol{\theta}} \ln p(\mathbf{x}_n \mid \mathbf{z}_n, \boldsymbol{\theta}) \, \mathrm{d}\mathbf{z}_n \\ &\approx \frac{1}{L} \sum_{l=1}^{L} \nabla_{\boldsymbol{\theta}} \ln p(\mathbf{x}_n \mid \mathbf{z}_n^{(l)}, \boldsymbol{\theta}). \end{aligned}$$

However, the gradient of the expectation with respect to $\boldsymbol{\phi}$ is a bit more complicated. To illustrate this problem, we define:

$$f_{\boldsymbol{\phi}}(\mathbf{z_n}) = \ln p(\mathbf{x_n} \mid \mathbf{z_n}, \boldsymbol{\theta}), \tag{3.7}$$

where it is important to note that $\mathbf{z_n}$ implicitly depends on $\boldsymbol{\phi}$ because it is sampled from a distribution defined by the encoder which is parametrized by $\boldsymbol{\phi}$. We then arrive at the following:

$$\begin{aligned} \nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_n|\mathbf{x}_n)}[f_{\boldsymbol{\phi}}(\mathbf{z})] &= \nabla_{\boldsymbol{\phi}} \left[\int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}_n \mid \mathbf{x}_n) f_{\boldsymbol{\phi}}(\mathbf{z}) \, d\mathbf{z} \right] \\ &= \int_{\mathbf{z}} f_{\boldsymbol{\phi}}(\mathbf{z}) \nabla_{\boldsymbol{\phi}} q_{\phi}(\mathbf{z}_n \mid \mathbf{x}_n) \, d\mathbf{z} + \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}_n \mid \mathbf{x}_n) \nabla_{\boldsymbol{\phi}} f_{\boldsymbol{\phi}}(\mathbf{z}) \, d\mathbf{z} \\ &= \underbrace{\int_{\mathbf{z}} f_{\boldsymbol{\phi}}(\mathbf{z}) \nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(\mathbf{z}_n \mid \mathbf{x}_n) \, d\mathbf{z}}_{\text{not an expectation in general}} + \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_n|\mathbf{x}_n)} \left[\nabla_{\boldsymbol{\phi}} f_{\boldsymbol{\phi}}(\mathbf{z}) \right] \end{aligned}$$

Thus, we arrive at an expression that, in general, is not an expectation and, as such, cannot be directly approximated using a Monte Carlo estimator. To address this, we apply

the previously mentioned *reparameterization trick*. Instead of directly sampling $\mathbf{z_n}$ from $\mathcal{N}(\boldsymbol{\mu}(\mathbf{x}_n, \boldsymbol{\phi}), \text{diag}(\boldsymbol{\sigma}^2(\mathbf{x}_n, \boldsymbol{\phi})))$, we sample $\epsilon \sim \mathcal{N}(0, 1)$ and then define:

$$\mathbf{z_n} = \boldsymbol{\mu}(\mathbf{x}_n, \boldsymbol{\phi}) + \boldsymbol{\sigma}^2(\mathbf{x}_n, \boldsymbol{\phi}))\epsilon.$$

Generating random samples for ϵ decouples the stochasticity from $\boldsymbol{\phi}$, ensuring that the dependence on $\boldsymbol{\phi}$ remains explicit which enables the efficient computation of gradient:

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z_n}|\mathbf{x_n})}\left[f_{\boldsymbol{\phi}}(\mathbf{z_n})\right] = \mathbb{E}_{p(\epsilon)}\left[f_{\boldsymbol{\phi}}(\mathbf{z_n})\right]$$

where $\mathbf{z_n} = \mathbf{g}(\epsilon, \boldsymbol{\phi}, \mathbf{x_n}) = \boldsymbol{\mu}(\mathbf{x}_n, \boldsymbol{\phi}) + \boldsymbol{\sigma}^2(\mathbf{x}_n, \boldsymbol{\phi}))\epsilon$. Now, we can finally form a simple Monte Carlo estimator:

$$\nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z_n}|\mathbf{x_n})}\left[f_{\boldsymbol{\phi}}(\mathbf{z_n})\right] = \nabla_{\boldsymbol{\phi}}\mathbb{E}_{p(\epsilon)}\left[f_{\boldsymbol{\phi}}(\mathbf{z_n})\right] = \mathbb{E}_{p(\epsilon)}\left[\nabla_{\boldsymbol{\phi}} f_{\boldsymbol{\phi}}(\mathbf{z_n})\right] \approx \frac{1}{L}\sum_{l=1}^{L} \nabla_{\boldsymbol{\phi}} f_{\boldsymbol{\phi}}(\mathbf{z_n}).$$

Now, we have precisely defined the entire flow of an input within the variational autoencoder and ensured the smooth operation of backpropagation, allowing us to finally define the total loss function:

$$\mathcal{L} = \sum_{n=1}^{N}\left(\frac{1}{2}\sum_{j=1}^{M}(1 + \ln\sigma_j^2(\mathbf{x_n}, \boldsymbol{\phi}) - \mu_j^2(\mathbf{x_n}, \boldsymbol{\phi}) - \sigma_j^2(\mathbf{x_n}, \boldsymbol{\phi})) + \frac{1}{L}\sum_{l=1}^{L} \ln p(\mathbf{x_n} \mid \mathbf{z}_n^{(l)}, \boldsymbol{\theta})\right).$$

where N is the number of samples, M is the dimension of the latent space, and L is the number of sampled latent representations of $\mathbf{x_n}$ for an approximation of the expectation.

A crucial question in applications is how to compute the reconstruction loss, $\ln p(\mathbf{x_n} \mid \mathbf{z}_n^{(l)}, \boldsymbol{\theta})$. Typically, the number of latent samples is set to $L = 1$, as sampling from the distribution inherently introduces noise, and using a single sample per input data point generally leads to better optimization. For simplicity of notation, we will consider $\ln p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$, implicitly assuming that $\mathbf{z}$ depends on $\mathbf{x}$ and $\boldsymbol{\phi}$. In probabilistic models, the form of the reconstruction loss is dictated by the assumed distribution of $\ln p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$. In [9], the authors assume that $\ln p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ follows a multivariate Gaussian distribution with a diagonal covariance matrix, given by $C = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_k^2)$. From equation (1.3), we directly obtain

$$\begin{aligned} p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) &= \frac{1}{(2\pi)^{k/2}\prod_{i=1}^{k}\sigma_i} \exp\left(-\frac{1}{2}\sum_{i=1}^{k}\frac{(x_i - \mu_i)^2}{\sigma_i^2}\right) \\ &= -\frac{k}{2}\ln(2\pi) - \sum_{i=1}^{k}\ln\sigma_i - \frac{1}{2}\sum_{i=1}^{k}\frac{(x_i - \mu_i)^2}{\sigma_i^2}, \quad \mathbf{x} \in \mathbb{R}^k. \end{aligned}$$

Algorithm 4 Variational autoencoder training

Input: Training data set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
  Encoder network $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$
  Decoder network $p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$
  Initial weight vectors $\boldsymbol{\theta}, \boldsymbol{\phi}$
  Learning rate η
Output: Final weight vectors $\boldsymbol{\theta}, \boldsymbol{\phi}$
repeat
  $\mathcal{L} \leftarrow 0$
  for $n \in \{1, \ldots, N\}$ **do**
    for $j \in \{1, \ldots, M\}$ **do**
      $\epsilon_{nj} \sim \mathcal{N}(0, 1)$
      $z_{nj} \leftarrow \mu_j(\mathbf{x}_n, \boldsymbol{\phi}) + \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})\epsilon_{nj}$
      $\mathcal{L} \leftarrow \mathcal{L} + \frac{1}{2}(1 + \ln \sigma_j^2(\mathbf{x_n}, \boldsymbol{\phi}) - \mu_j^2(\mathbf{x_n}, \boldsymbol{\phi}) - \sigma_j^2(\mathbf{x_n}, \boldsymbol{\phi}))$
    end for
    $\mathbf{z}_n \leftarrow (z_{n1}, \ldots, z_{nM})$
    $\mathcal{L} \leftarrow \mathcal{L} + \ln p(\mathbf{x}_n \mid \mathbf{z}_n, \boldsymbol{\theta})$
  end for
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$ ▷ Update decoder weights
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \eta \nabla_{\boldsymbol{\phi}} \mathcal{L}$ ▷ Update encoder weights
until converged
return $\boldsymbol{\theta}, \boldsymbol{\phi}$

In practical applications, assuming σ_i^2 to be constant simplifies the objective, making the maximization of $\ln p_{\mathbf{w}}(\mathbf{x} \mid \mathbf{z})$ equivalent to minimizing the mean squared error (MSE) which we observe in summation term. Consequently, MSE will be used as the reconstruction loss in the implementation described in Section 4.3. It is also worth noting that, in practice, the log-variance is often computed to ensure its positivity and numerical stability. The variance can then be easily obtained using the formula $\sigma_i = \exp\left(\frac{1}{2} \log \sigma_i^2\right)$.

A different variant of the variational autoencoder, discussed in Section 4.4, employs an alternative reconstruction loss defined in [15]. In this approach, the authors modify the ELBO objective to accommodate an arbitrary differential loss $\Delta(\mathbf{x}, \hat{\mathbf{x}})$ by replacing the probabilistic decoder with a deterministic mapping from the latent representation:

$$\hat{\mathbf{x}} = f_{\mathbf{w}}(\mathbf{z}).$$

The resulting objective function consists of a weighted sum of the expected loss of $\hat{\mathbf{x}}$ under the encoder's distribution over $\mathbf{z}$ and the KL regularization term:

$$\hat{\mathcal{L}} = C \cdot \mathbb{E}_{q_\phi}[\Delta(\mathbf{x}, \hat{\mathbf{x}})] + D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) || p(\mathbf{z})),$$

where the constant C controls the trade-off between the image-specific loss and the regularization term.

Score function estimator - REINFORCE

Note that the generative model creates a parameterized continuous distribution. In the past, discrete distributions were commonly used, and in line with that, we will present an additional technique that is applicable to discrete distributions, unlike the reparameterization trick. Specifically, we aim to enable backpropagation through the sampling process of $\mathbf{z}$, i.e., to approximate the gradient

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

where f is an example function, such as the one in Equation (3.7).

To find the gradient with respect to $\boldsymbol{\phi}$, we apply the gradient operator to the expectation and product rule to the integrand:

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] &= \nabla_{\boldsymbol{\phi}} \int f(\mathbf{z}) q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})\, d\mathbf{z} = \int \nabla_{\boldsymbol{\phi}} \Big(f(\mathbf{z}) q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \Big)\, d\mathbf{z} \\
&= \int f(\mathbf{z}) \nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})\, d\mathbf{z} + \int q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \nabla_{\boldsymbol{\phi}} f(\mathbf{z}) d\mathbf{z} \\
&= \int f(\mathbf{z}) \nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})\, d\mathbf{z} + \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\nabla_{\boldsymbol{\phi}} f(\mathbf{z})]}_{\text{MC estimator}}.
\end{aligned}$$

Now we focus on the first term of the equation above because the second one is easily approximated with the Monte Carlo estimator. The gradient of $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ with respect to $\boldsymbol{\phi}$ is related to the gradient of the log-likelihood:

$$\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = \frac{\nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})} \implies \nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}).$$

Substituting this back into the integral, we obtain the gradient expressed as a sum of expectations, which can then be easily approximated:

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] &= \int f(\mathbf{z}) q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})\, d\mathbf{z} + \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\nabla_{\boldsymbol{\phi}} f(\mathbf{z})] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[f(\mathbf{z}) \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})] + \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\nabla_{\boldsymbol{\phi}} f(\mathbf{z})] \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})\ \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) + \nabla_{\boldsymbol{\phi}} f(\mathbf{z})].
\end{aligned}$$

This gives us an alternative gradient estimator for the expectation, which supports the modeling of discrete distributions by the generative model. This estimator is also known as the

score function estimator or the *REINFORCE gradient estimator*. In the literature, it is most commonly presented in the following form:

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z})}[f(\mathbf{z})] = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z})}[f(\mathbf{z}) \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z})].$$

However, the reason we will not use this estimator when approximating the ELBO is that it has been experimentally shown in several studies to have high variance when compared to the reparametrization trick. This can be observed in Figure 3.5, where the example is shown using a simple function. Despite the substantial empirical evidence demonstrating the high variance of this estimator, there are relatively few studies that rigorously explain why the reparameterization trick is so effective. In the paper [17], appropriate assumptions on the parameterized distribution $q_{\boldsymbol{\phi}}$ are introduced, and the claim is rigorously proven. However, the doctoral dissertation [5, Section 3.1.2] provides examples where the score function estimator yields lower variance than the reparameterization trick. Nonetheless, it also presents a proposition and specific conditions on the distribution that demonstrate the opposite, explaining when and why the reparameterization trick outperforms other approaches.

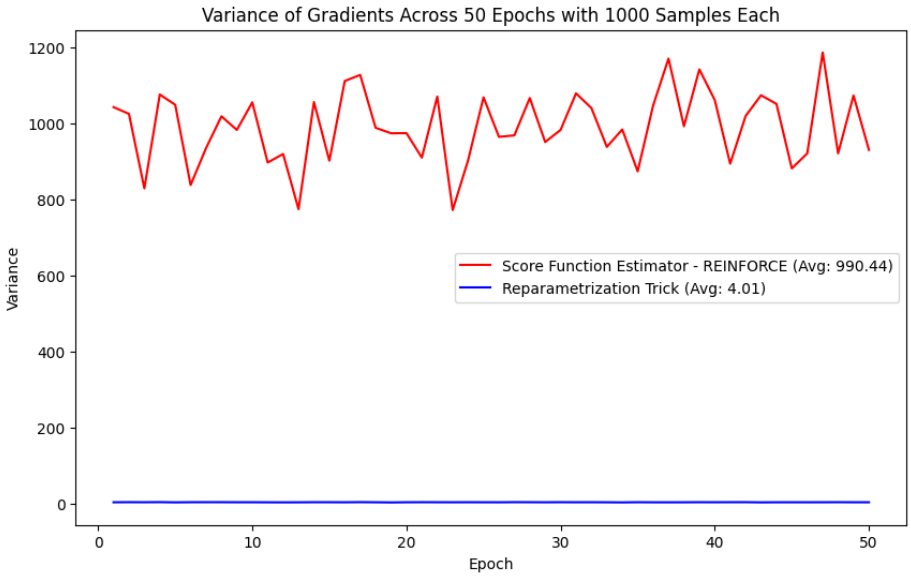


Figure 3.5: Variance comparison of two different approaches for approximating the gradient of an expectation. The testing was performed on the function $f(z) = z^2$ and 1000 samples were used over 50 epochs.

Chapter 4

Application - Image synthesis

Variational autoencoders can be applied across various domains. They are used in anomaly detection (comparing the reconstruction error and the likelihood of observed data), natural language processing (creating coherent sentences by sampling from the latent space), and recommender systems (learning latent factors for user-item interactions and generating recommendations). However, their primary application lies in image processing and synthesis, which is the focus of the following discussion. We will test the standard VAE and its variant, *multiscale structural similarity VAE (MSSIM-VAE)*, which integrates the *multiscale structural similarity index (MSSIM)* into its loss function to improve the quality of generated images.

4.1 Dataset

During the training, validation, and testing of the models, the *Large-scale CelebFaces Attributes (CelebA) Dataset* was utilized. This dataset is widely used in computer vision tasks such as recognition, detection, editing, and generation of human faces. It is characterized by high diversity among the images and a wealth of useful features for each photograph. The dataset, with a size of 1.37 GB, consists of 202,599 images with dimensions of 178 x 218 pixels, featuring 10,177 individual celebrities. Each image includes 5 landmark locations for identifying facial features (left eye, right eye, nose tip, left mouth corner, right mouth corner) and 40 binary attributes, such as those indicating characteristics like bald, big lips, mustache, eyeglasses, male, etc. Examples of images from the dataset can be seen in Figure 4.1, while for additional details and similar datasets, refer to [12].

However, for our purposes, we will only use images without facial features or attributes, along with the file `list_eval_partition.txt`, which contains information on whether an image is intended for training, validation, or testing. This file is necessary because a `LightningDataModule` for automatically configuring datasets from `pytorch_lightning`

library, requires information to correctly assign data to training, validation, and testing sets. This module also supports initialization using the pre-defined `CelebA` module from `torchvision.datasets`, which only requires local download of the dataset for use. Upon closer inspection, it becomes evident that some images contain noise around the

Figure 4.1: Three randomly selected photos from the dataset.

human portrait. Therefore, we applied a center crop of 148 x 148 pixels on each image, effectively cutting out the surrounding noise while preserving the main content—the human portrait. Additionally, to prevent overfitting and improve the model's learning, we applied random horizontal flipping to each image. Finally, to reduce the number of model parameters and speed up the process, we resized each image to 64x64 pixels and normalized their pixel intensities to the range [0, 1]. An example of such transformations can be seen in Figure 4.2. Overall, a total of 202,599 images were used to create the model, of which 162,770 were used for training, 19,867 for validation, and 19,962 for testing.

Figure 4.2: The first picture shows the original image, where some unexplained noise is visible around the hair. In the second picture, we apply the center crop and horizontal flip transformations. Finally, the last picture displays the image after it has been resized.

4.2 Model architecture

As mentioned earlier, we will compare two types of variational autoencoders that primarily differ in their loss functions, specifically the reconstruction loss, while the KL divergence term remains the same. In order to make a fair comparison, everything except the loss functions is identical, so the configuration outlined below applies to both types of models.

The encoder processes input images with 3 channels (RGB) through a series of convolutional layers, each increasing the number of output channels which are set by default to `[32, 64, 128, 256, 512]`, while the spatial dimensions of the image are halved at each layer. Each convolutional layer is followed by a `LeakyReLU` activation function, which allows small negative values for inactive neurons, in contrast to the standard `ReLU`, which outputs zero for negative inputs. The encoder outputs two 128-dimensional vectors representing the mean and log-variance of the latent space distribution, which are used to sample the latent variable. These vectors are produced by two fully connected layers, which take the output of the final convolutional layer. The decoder reconstructs the input image from the latent variable by first mapping it to a higher-dimensional feature representation using a fully connected layer. This feature map is then reshaped and progressively upsampled through a series of transposed convolutional layers, where the number of output channels decreases in the reverse order of the encoder's hidden dimensions, set by default to `[512, 256, 128, 64, 32]`. The final layer applies a transposed convolution followed by a standard convolution to generate the reconstructed image with 3 channels. A `Tanh` activation function is used to scale the pixel values between -1 and 1, ensuring consistency with the normalized input data. Recall that during the dataset preprocessing, each image was converted into a tensor with pixel values ranging from 0 to 1. The reason we can still use `Tanh` in the final layer is that, when generating or reconstructing images that will be shown later, we utilize a built-in function that ensures proper scaling from [-1, 1] to [0, 1]. Finally, with this setup, the models resulted in a total of 3,937,635 learnable parameters.

Regarding the remaining configuration, each model was trained for 50 epochs with a batch size of 64 images, a learning rate of 0.005, and a latent space dimension of 128. The training was performed on the Kaggle platform [1], utilizing two T4 GPUs in parallel, which significantly improved the overall efficiency of the training process. The entire process was carried out using `PyTorch Lightning`, a high-level framework built on top of `PyTorch`, which significantly simplified the workflow, particularly in terms of logging, checkpointing, and multi-GPU scaling. The most important components were the `LightningModule` and `Trainer`, which enabled efficient and modular code, allowing the focus to remain primarily on improving model quality rather than dealing with the setup of training loops, low-level operations, and other repetitive tasks.

The original code used to reproduce the examples from this chapter is available at GitHub Repository.

4.3 Variational autoencoder (VAE)

We apply all the theoretical considerations related to variational autoencoders, where the reconstruction loss is defined as the mean squared error between the input image and the reconstructed image:

$$\text{Reconstruction Loss} = \frac{1}{k}\sum_{i=1}^{k}(x_i - \hat{x}_i)^2,$$

where:

- x_i is the i-th pixel of the input image,
- $\hat{x}_i$ is the i-th pixel of the reconstructed image,
- k is the total number of pixels in the image.

In Figure 4.3, we observe the behavior of the training and validation losses as a function of the batches, while Figures 4.4 and 4.5 show the reconstructed and generated images over epochs. The reconstructed and generated images lack detail and appear somewhat blurry, with improvements expected in the MSSIM variant of the variational autoencoder.

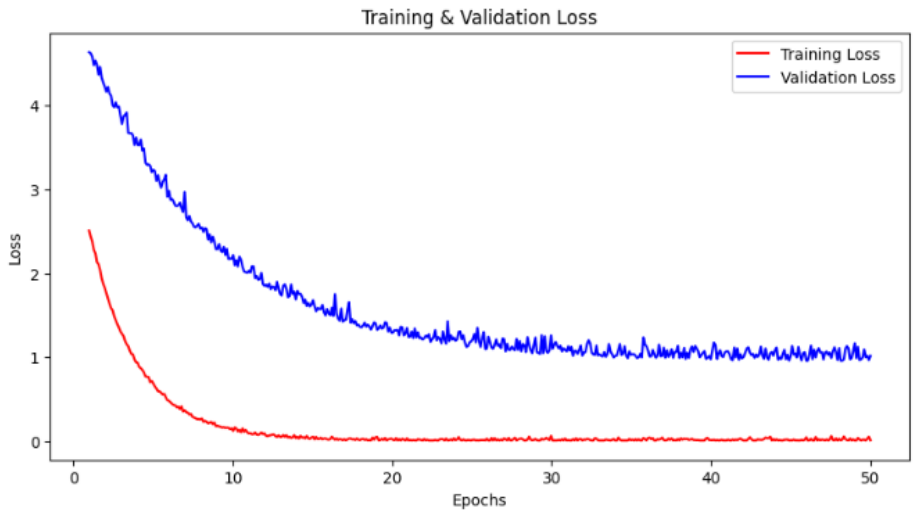


Figure 4.3: Training and validation losses based on the number of epochs.

Figure 4.4: The first column illustrates the input images, while the second column presents their corresponding reconstructions during the training process. The first row depicts the results from the initial epoch, whereas the second row showcases the outcomes from the final, fiftieth epoch.

Figure 4.5: Generated images over the epochs. The first row shows the 1st and 25th epochs, while the second row shows the 37th and final 50th epochs.

4.4 MSSIM - VAE

The structural similarity index measure (SSIM) evaluates the perceived quality of images based on three components: luminance, contrast, and structure. For the two images x and

y being compared, the formula is given by:

$$\text{SSIM}(x, y) = [l(x, y)]^{\alpha} \cdot [c(x, y)]^{\beta} \cdot [s(x, y)]^{\gamma},$$

where the individual components are defined as follows:

$$\underbrace{l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}}_{\text{luminace}}, \underbrace{c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}}_{\text{contrast}}, \underbrace{s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}}_{\text{structure}}.$$

Setting the weights α, β, γ to 1, the formula can be reduced to the form shown below:

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}.$$

Note that the constant C_3 is absent in the equation above. It is commonly set as $C_3 = \frac{C_2}{2}$ to obtain a simpler expression. The means μ_x and μ_y represent the average intensities of the pixels, while σ_x^2 and σ_y^2 are the variances, indicating intensity spread. The covariance σ_{xy} shows the correlation between pixel intensity changes in both images, while the constants C_1 and C_2 ensure numerical stability, preventing division by zero in the calculations. These constants are typically derived based on the dynamic range L of pixel values in the input images. The general equations are given as:

$$C_1 = (K_1 \cdot L)^2, \quad C_2 = (K_2 \cdot L)^2,$$

where K_1 and K_2 are small constants, often set to $K_1 = 0.01$ and $K_2 = 0.03$, and L is the dynamic range of the pixel values (e.g., 255 for 8-bit images). In the provided implementation, the dynamic range is normalized to $L = 1$, so the constants are computed as:

$$C_1 = (0.01 \cdot 1)^2 = 0.0001, \quad C_2 = (0.03 \cdot 1)^2 = 0.0009.$$

These specific values are suitable for our normalized images with a pixel value range of $[0, 1]$.

SSIM is a perception-based metric that evaluates image quality by assessing changes in structural information, unlike other metrics, like MSE, which focus on absolute error measurement. It accounts for human visual perception elements such as luminance and contrast masking. Structural information reflects strong inter-dependencies among nearby pixels, integral to understanding object structures in a scene. Luminance masking refers to the reduced visibility of distortions in brighter areas, while contrast masking describes the decreased visibility of distortions amidst active or textured regions.

Next, we define a more advanced variant of SSIM, known as multiscale SSIM (MS-SSIM). This approach involves the application of SSIM across multiple image scales, allowing for a more comprehensive comparison of image structures at various resolutions.

Definition 4.4.1. *Let x and y be two images, S be the total number of scales, and $\{\alpha_1, \alpha_2, \ldots, \alpha_S\}$ be the weights assigned to each scale. The multiscale SSIM (MS-SSIM) is given by:*

$$MS\text{-}SSIM(x, y) = \prod_{j=1}^{S} \left[SSIM_j(x, y)\right]^{\alpha_j},$$

where $SSIM_j$ is an SSIM calculated at the j-th scale of the images.

Example 4.4.2 (MS-SSIM calculation)**.** *Suppose we have two grayscale images x and y:*

$$x = [50, 51, 49, 52], \quad y = [48, 50, 47, 51]$$

We calculate the following:

$$\mu_x = 50.5, \quad \mu_y = 49, \quad \sigma_x^2 = 1.25, \quad \sigma_y^2 = 2, \quad \sigma_{xy} = 1.$$

Let $C_1 = 0.01$ and $C_2 = 0.03$. Then:

$$SSIM(x, y) = \frac{(2 \cdot 50.5 \cdot 49 + C_1)(2 \cdot 1 + C_2)}{(50.5^2 + 49^2 + C_1)(1.25 + 2 + C_2)} = \frac{(4950.01)(2.03)}{(5050.01)(3.28)} \approx 0.98$$

This value is close to 1, indicating high similarity which completes the SSIM calculation. Calculating MS-SSIM involves the sequential application of SSIM across multiple image scales. For simplicity, we outline only the calculation steps:

(i) start with two 256×256 images I_1 and I_2,

(ii) compute SSIM at full resolution,

(iii) downsample the images to 128×128, and compute SSIM again,

(iv) repeat for 64×64, 32×32, and 16×16 resolutions,

(v) combine SSIM scores using predefined weights, e.g., $[0.0448, 0.2856, 0.3001, 0.2363, 0.1333]$.

The coefficients for the SSIM scores in the previous example are not arbitrary; they were derived through empirical evaluation, as described in the paper [16, Section 3.2].

In the MSSIM-VAE implementation, the coefficients mentioned earlier were utilized, and the reconstruction loss was formulated based on the MS-SSIM metric. To ensure differentiability, we employed its variant, developed following the approach outlined in [14]. Analogously to the previous variant, in Figure 4.6, the behavior of the training and validation losses as a function of the batches is observed, while Figures 4.7 and 4.8 display the reconstructed and generated images over the epochs. What can be observed is a significant improvement in the quality of both the reconstructed and generated images, which aligns with the newly introduced metric, providing a more accurate assessment of the similarity between the two images.

Figure 4.6: Training and validation losses based on the number of epochs.

4.5 Conclusion

We compared two different implementations of a variational autoencoder and examined their inference and generative capabilities. Both models produced meaningful reconstructions and generated examples, indicating that *posterior collapse* did not occur. At the beginning of training, the term $\ln p_\theta(\mathbf{x} \mid \mathbf{z})$ is relatively weak, making the state where $q_\phi(\mathbf{z} \mid \mathbf{x}) \approx p_\theta(\mathbf{z})$ desirable for maximizing the ELBO. This undesirable stable equilibrium is known as *posterior collapse*, and escaping from it is generally quite difficult. However, several methods exist to prevent such behaviour. One common approach is *KL annealing*, where the latent cost $\mathrm{D}_{KL}(q_\phi(\mathbf{z} \mid \mathbf{x}) \parallel p_\theta(\mathbf{z}))$ is introduced gradually during training. This is achieved by multiplying it with a coefficient α, which increases from 0 to 1 as the number of training epochs progresses.

By visually inspecting the generated images, we observe that MSSIM-VAE outperforms the standard VAE due to its reconstruction loss function being better adapted to the image dataset. However, both models exhibit a certain degree of blurriness. This primarily occurs because we assume a relatively simple Gaussian posterior distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$, which, due to its characteristics (e.g., unimodality), cannot fully capture all details—both due to dimensionality constraints and amortized inference. An improvement involves increasing the flexibility of the inference model (encoder) by further transforming the posterior distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ through a series of invertible transformations, thereby obtaining a more complex distribution. This concept is known as *normalizing flows*, and more details can be found in [10, Chapter 3].

Regardless of all considerations, we have observed that a remarkably simple mecha-

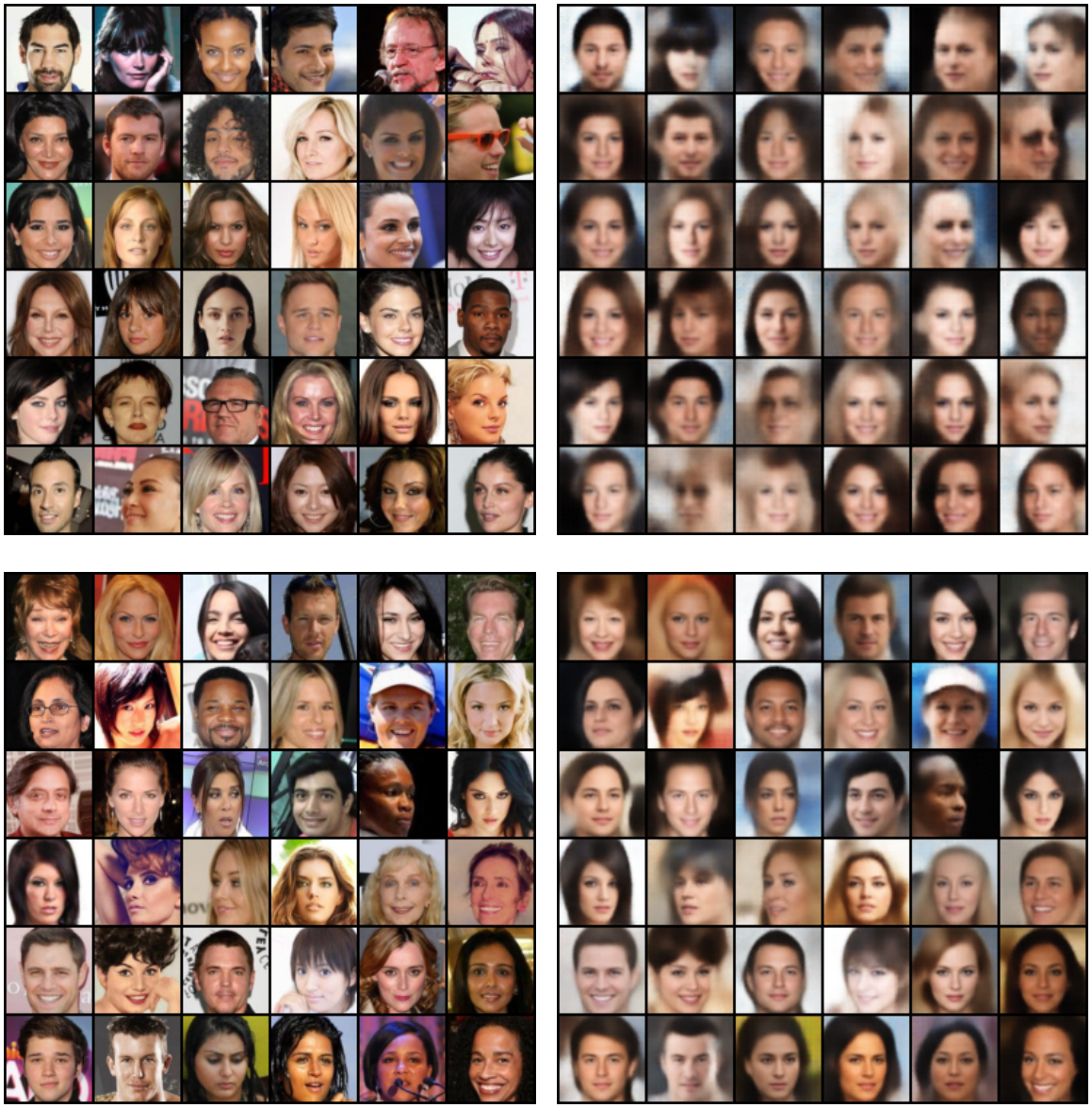


Figure 4.7: The first column illustrates the input images, while the second column presents their corresponding reconstructions during the training process. The first row depicts the results from the initial epoch, whereas the second row showcases the outcomes from the final, fiftieth epoch.

nism for approximating intractable distributions using parameterized neural networks can lead to highly effective generative modeling. The explained flow of collaboration between two neural network models has served as both an introduction and a foundation for the

Figure 4.8: Generated images over the epochs. The first row shows the 1st and 25th epochs, while the second row shows the 37th and final 50th epochs.

future development of more complex models based on the same principle. One of the most well-known examples is *generative adversarial networks* (GANs), where the roles of the encoder and decoder are taken over by the discriminator and generator, respectively.

Bibliography

[1] *Kaggle*, https://www.kaggle.com/, Accessed: 2025-01-04.

[2] Christopher Michael Bishop and Hugh Bishop, *Deep learning - foundations and concepts*, 1st ed., 2023, ISBN 978-3-031-45468-4 (english).

[3] Wadii Boulila, Maha Driss, Eman Alshanqiti, Mohammed Al-Sarem, Faisal Saeed, and Moez Krichen, *Weight initialization techniques for deep learning algorithms in remote sensing: Recent trends and future perspectives*, pp. 477–484, January 2022, ISBN 978-981-16-5558-6.

[4] DeepMind, *Gemma scope: Helping the safety community shed light on the inner workings of language models*, DeepMind Blog (2023), https://deepmind.google/discover/blog/gemma-scope-helping-the-safety-community-shed-light-on-the-\inner-workings-of-language-models/, Accessed: 2024-12-28.

[5] Yarin Gal, *Uncertainty in deep learning*, PhD thesis, University of Cambridge, 2016, https://www.cs.ox.ac.uk/people/yarin.gal/website/thesis/thesis.pdf.

[6] Xavier Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, Journal of Machine Learning Research - Proceedings Track **9** (2010), 249–256.

[7] Rishabh Gupta, *KL Divergence Between Two Gaussian Distributions*, 2020, https://mr-easy.github.io/2020-04-16-kl-divergence-between-2-gaussian-distributions/, Accessed: 2024-12-20.

[8] Adam Harley, *Interactive visualization of CNNs*, https://adamharley.com/nn_vis/cnn/3d.html, n.d., Accessed: 2025-01-19.

[9] Diederik P. Kingma and Max Welling, *Auto-encoding variational Bayes*, CoRR **abs/1312.6114** (2013).

[10] _____, *An introduction to variational autoencoders*, Found. Trends Mach. Learn. **12** (2019), 307–392.

[11] Siddharth Krishna Kumar, *On weight initialization in deep neural networks*, ArXiv **abs/1704.08863** (2017).

[12] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang, *Deep learning face attributes in the wild*, Proceedings of International Conference on Computer Vision (ICCV), December 2015.

[13] Mathematical Musings, *How gaussian distribution maximizes entropy: The proof*, 2023, https://medium.com/mathematical-musings/how-gaussian-distribution-maximizes-entropy-the-proof-7f7dcb2caf4d, Accessed: 2025-01-25.

[14] Jorge Pessoa, *pytorch-msssim*, https://github.com/jorge-pessoa/pytorch-msssim/tree/dev?tab=readme-ov-file, 2018, Accessed: 2025-01-26.

[15] Jake Snell, Karl Ridgeway, Renjie Liao, Brett D. Roads, Michael C. Mozer, and Richard S. Zemel, *Learning to generate images with perceptual similarity metrics*, 2017 IEEE International Conference on Image Processing (ICIP) (2015), 4277–4281.

[16] Z. Wang, Eero Simoncelli, and Alan Bovik, *Multiscale structural similarity for image quality assessment*, vol. 2, December 2003, pp. 1398 – 1402 Vol.2, ISBN 0-7803-8104-1.

[17] Ming Xu, Matias Quiroz, Robert Kohn, and Scott Anthony Sisson, *Variance reduction properties of the reparameterization trick*, International Conference on Artificial Intelligence and Statistics, 2018.

Sažetak

U ovom radu izložena je teorijska pozadina i opisana konkretna implementacija varijacijskog autoenkodera, jednog od primjera generativnih modela koji su kao važan dio umjetne inteligencije značajno utjecali na današnje trendove.

Prvim poglavljem detaljno razrađujemo algoritam maksimizacije očekivanja koji predstavlja polazišnu točku za motivaciju funkcije gubitka našeg modela gdje enkoder nastoji što bolje aproksimirati pravu posteriori distribuciju, a dekoder smanjiti rekonstrukcijski gubitak u svrhu što boljih generativnih sposobnosti. Drugo poglavlje uvodi neuronsku mrežu koja upravo predstavlja tip modela enkodera i dekodera, a zbog čije fleksibilnosti varijacijski autoenkoder dobiva mogućnost modelirati vrlo kompleksne distribucije. Točnije, promatramo poseban tip konvolucijske neuronske mreže pogodnom za probleme obrade, odnosno kreiranja slika. Treće poglavlje definira oba modela koji međusobnim funkcioniranjem čine varijacijski autoenkoder, zajedno s funkcijom gubitka (eng. *evidence lower bound*) i reparametrizacijskim trikom koji, naspram tradicionalnog REINFORCE aproksimatora, smanjuje varijancu gradijenata tijekom provođenja algoritma propagacije unatrag. U četvrtom poglavlju prikazujemo dvije implemetacije varijacijskog autoenkodera koje se razlikuju po svojim rekonstrukcijskim funkcijama gubitka. Prva varijanta implementira prosječnu kvadratnu pogrešku, dok druga ipak uzima u obzir prirodu podataka (slike) te koristi MSSIM metriku za mjeru različitosti između ulaznog podatka i onog dobivenog rekonstrukcijom iz dekodera.

Rezultati ovog rada nastojali su približiti teoriju, razumijevanje i intuiciju iza načina kako funkcioniraju generativni modeli, a koji su danas sve češće viđeni, pogotovo u području obrade i kreiranja slika.

Summary

This thesis presents the theoretical background and describes the concrete implementation of the variational autoencoder (VAE), a key example of generative models that, as an integral part of artificial intelligence, have significantly influenced current trends.

The first chapter thoroughly explores the expectation-maximization (EM) algorithm, which serves as the foundational motivation for the loss function of our model. Here, the encoder aims to approximate the true posterior distribution as closely as possible, while the decoder minimizes the reconstruction loss to enhance the generative capabilities. The second chapter introduces the neural network, which forms the type of model for both the encoder and the decoder. Due to its flexibility, the VAE is capable of modeling highly complex distributions. Specifically, we focus on a special type of convolutional neural network, which is well-suited for image processing and generation tasks. The third chapter defines the encoder and decoder models, which together make up the VAE, along with the evidence lower bound loss function and the reparameterization trick, which reduces gradient variance during backpropagation, compared to the traditional REINFORCE estimator. The fourth chapter presents two VAE implementations, differing in their reconstruction loss functions. The first variant uses mean squared error, while the second takes the nature of the data (images) into account and uses the MSSIM metric to measure the difference between the input data and the reconstruction from the decoder.

The results of this work aim to bridge theory, understanding, and intuition behind how generative models function, especially in fields like image processing and generation, where such models are becoming increasingly prevalent.

Biography

I was born on February 17, 2001, in Virovitica and grew up in Grubišno Polje, where I completed elementary school and enrolled in the Bartol Kašić high school's gymnasium program. In the later years of elementary school, I developed an interest in mathematics, primarily because I found it easy to grasp without much effort. However, this advantage did not persist in my first year of high school, and I only regained my interest in mathematics in my second year. Over time, I discovered a genuine passion for the subject, which led me to attend the Center of Excellence for Mathematics in Bjelovar in my final year of high school.

This experience ultimately influenced my decision to enroll in the undergraduate Mathematics program at the Faculty of Science, University of Zagreb, in 2019. In 2022, I received an award for the most successful final-year undergraduate student. This period will also be remembered for the numerous lectures I held in linear algebra, mathematical analysis, and geometry, which captured significant interest from my colleagues. Throughout my undergraduate studies, I developed a strong interest in computer science, leading to the decision to pursue a master's degree in Computer Science and Mathematics in 2022. In 2024, I received another award for the most successful final-year graduate student. In February of the same year, I started an internship at Infobip, where I now work as a Data Scientist.