Project 3A

Supervised and Unsupervised Similarity Methods in Deep Learning

CHADUTEAU Théa - FICM Mines Nancy GIMA ID Big Data

Febuary, 12, 2025

Table of contents

1 Introduction

1.1 Context

The ability to estimate similarity between two objects is a cornerstone of numerous data analysis algorithms, underpinning applications such as facial recognition, customer classification, and recommendation systems. In simpler cases, similarity can often be measured with predefined metrics, such as cosine similarity or Euclidean distance. However, when dealing with complex objects – such as multimedia data (e.g., images, videos, or audio) – defining a meaningful similarity measure becomes significantly more challenging.

Similarity learning, a subfield of machine learning, partially addresses this issue by leveraging models that automatically learn to evaluate similarity based on the data itself. In recent years, neural networks have emerged as powerful tools for this purpose. Neural architectures such as Siamese networks have demonstrated their effectiveness in learning embeddings that can represent and compare complex objects efficiently. These learned representations allow similarity to be modeled in a way that is adaptable to diverse data types, from 1D sequential data to high-dimensional 3D data structures. But what about unsupervised methods? The task is much trickier and harder to model to have interesting results as we are going to see later in this study.

The potential applications of similarity learning are vast, ranging from identifying duplicate images in a database to comparing protein structures in bioinformatics. Implementing such models involves addressing key challenges, including:

- Handling high-dimensional and multimodal data;
- Designing scalable solutions for large datasets;
- Mitigating the impact of noisy or incomplete data;
- Overcoming the limited availability of labeled pairs or triplets for training.

1.2 Goal definition of the project

The primary aim of this project is to explore and implement neural network models for learning similarity between complex objects. The approach seeks to go beyond traditional distance metrics by employing neural networks that can adaptively learn similarity scores tailored to the objects being compared. These scores may resemble measures like cosine similarity or Euclidean distance but are not constrained by the limitations of fixed metrics.

This project emphasizes experimentation and exploration, with key objectives including:

- Investigating state-of-the-art similarity learning methods, such as embeddings and clustering-based approaches;
- Establishing robust definitions of similarity that can generalize to real-world applications, such as classification or recommendation systems;
- Testing and developing neural architectures capable of handling 1D, 2D, and 3D data modalities;
- Developing unsupervised models from scratch capable of handling 1D and 2D data modalities.

1.3 Timeline of the project

The project is structured into several phases to ensure systematic progress:

Phase 1: Literature Review and Exploration (Month 1): Conduct an in-depth review of existing methods for similarity learning, including neural network-based approaches, and evaluate their strengths and limitations.

Phase 2: Prototyping and Initial Testing (Month 2): Develop and test initial models on a diverse set of datasets, exploring different neural architectures and similarity definitions.

Phase 3: Advanced Implementation and Analysis (Months 3): Refine the models based on initial findings and evaluate their performance on benchmark datasets. Implement clustering techniques to assess the quality of learned similarity measures.

Phase 4: Final Evaluation and Reporting (Month 4): Perform rigorous testing, consolidate results, and prepare a comprehensive report on the findings and insights gained throughout the project.

2 Litterature review

Similarity learning, a supervised machine learning approach, focuses on measuring the similarity or dissimilarity between objects instead of directly predicting labels. Various methods and frameworks have emerged to address the challenges posed by high-dimensional data, scalability, and noise. This section highlights the different types of similarity learning and prominent existing techniques in similarity learning.

2.1 How do we define similarity?

There are many definitions of what similarity in data science is. The choice of definition depends on the specific application, the type of data and the goals of the analysis. We tried below to synthesize the most common ways data scientists define similarity for their research.

2.1.1 Feature-based similarity

Similarity can be defined based on the shared features or attributes of the objects (1). In this context, two objects are considered similar if they share a high degree of overlap in their features, regardless of how those features are measured. This is common in data science and machine learning applications, where similarity is typically defined in terms of feature vectors.

2.1.2 Transformation-based similarity

One other way to define similarity is through the lens of transformational effort. This approach assesses similarity based on the minimal number of steps, operations, or transformations needed to convert one object into another (2). For example:

- In text processing, similarity can be defined by the minimum number of character insertions, deletions, or substitutions required to transform one string into another (e.g., Levenshtein distance or edit distance see p. section 2.2.3.2)
- In computational complexity, the Kolmogorov complexity measures the shortest algorithm or number of operations required to generate one object from another, which can also be viewed as a proxy for similarity.

This perspective is particularly useful when the focus is on the structural or procedural relationship between objects.

2.1.3 Semantic similarity

In domains like natural language processing (NLP) or knowledge graphs, similarity can be defined based on the meaning or context of the objects (3). For example, two words (e.g., "car" and "automobile") are similar if they share the same meaning, even if their literal representations (character sequences) differ. This is often formalized using techniques like WordNet similarity or contextual embeddings.

2.1.4 Relational similarity

Objects can also be defined as similar based on their relationships to other objects in a dataset (4). For instance, in graph-based methods, two nodes may be similar if they share common neighbors, belong to the same subgraph, or have similar roles within a network.

2.2 Measures of similarity

2.2.1 Distance metrics

One way to measure similarity is to use distance metrics, which calculates how far apart two points are in a given space.

2.2.1.1 Cosine similarity (5)

It measures the cosine of the angle of two non-zero vectors and therefore the similarity of them. The formula to calculate it is the following:

$$Cosine\ Similarity\ (A, B) = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sum_{i=1}^{n} (B_i)^2}$$

where $A_i$and B_i are the components of vectors A and B, and n is the dimension of the vectors. The resulting cosine similarity is a value between -1 and 1 where the value 1 indicates that the vectors are identical, the value 0 means that the vectors are orthogonal, and therefore there is no similarity, and the value -1 means they are diametrically opposed. Cosine similarity is widely used in various applications, such as information retrieval and recommendation systems, where the goal is to measure the similarity between **textual data. It is particularly suitable and useful for high-dimensional spaces such as text analysis.**

2.2.1.2 Euclidian distance (6)

It is often used for similarity learning and the idea behind this method is to measure the similarity between two vectors by calculating the straight-line distance (Euclidean distance) between their points in the vector space. Similarly to the previous method, smaller distances indicate greater similarity, while larger distances indicate higher dissimilarity. The Euclidean distance between two vectors A and B in an n-dimensional space is calculated using the following formula:

$$d(A,B) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}$$

where A_i and B_i are the components of the vectors in the i-th dimension. It is widely used in image recognition where data are represented in 2D and 3D spaces, which are intuitively understandable. Conversely, in high-dimensional spaces, the concept of "distance" is not straightforward.

2.2.1.3 Manhattan distance (7)

Manhattan distance, also known as the taxicab or city block distance, measures the similarity between two vectors by summing the absolute differences of their corresponding components. The formula to calculate Manhattan distance between two vectors

$$d(A,B) = \sum_{i=0}^{n}|A_i - B_i|$$

Where A_i and B_i are the components of the vectors A and B in the i-th dimension. Unlike Euclidean distance, which considers the shortest straight-line path, Manhattan distance is based on the path along the axes, making it more suitable for grid-based systems, such as navigating a city grid. Smaller distances indicate greater similarity, while larger distances represent increased dissimilarity. This method is particularly useful in high-dimensional spaces where axis-aligned movements are more natural or when the data contains significant outliers, as the Manhattan distance is less sensitive to extreme values compared to Euclidean distance. It is frequently applied in fields such as computer vision, clustering, and recommendation systems.

2.2.1.4 Jaccard similarity (8)

Jaccard similarity measures the degree of similarity between two sets by comparing the size of their intersection to the size of their union. It is a widely used metric for binary or categorical data. The formula to calculate Jaccard similarity between two sets A and B is as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

where $|A \cap B|$ represents the number of elements common to both sets, and $|A \cup B|$ represents the total number of distinct elements in the two sets. Jaccard similarity ranges from 0 to 1, with a value of 1 indicating that the two sets are identical and a value of 0 indicating no overlap. This measure is particularly effective for sparse datasets, such as text data represented as sets of words or terms. It is commonly used in applications like text similarity analysis, recommendation systems, and clustering, where determining overlap between sets of features, items, or users is critical. Its straightforward interpretation makes it a popular choice for comparing categorical and binary data.

2.2.1.5 Minkowski distance (9)

Minkowski distance is a generalized metric that encompasses several other distance measures, such as Euclidean and Manhattan distances, depending on the value of its parameter p. It measures the distance between two vectors A and B in an n-dimensional space using the following formula:

$$d(A,B) = \left(\sum_{i=0}^{n} |A_i - B_i|^p \right)^{\frac{1}{p}}$$

Where A_i and B_i are the components of the vectors A and B in the i-th dimension and p positive real number that determines the type of distance. For p=1, the Minkowski distance becomes equivalent to Manhattan distance, while for p=2, it reduces to Euclidean distance. By adjusting the value of p, Minkowski distance can adapt to various contexts, making it highly flexible.
Minkowski distance is especially useful in machine learning and data analysis when different metrics need to be explored or combined to account for the specific characteristics of the data. It is frequently applied in clustering and similarity-based learning tasks, particularly in high-dimensional spaces, where the choice of p can significantly affect the sensitivity of the distance measure to variations in the data. This adaptability makes it a powerful tool for analyzing complex datasets.

2.2.1.6 Chebyshev distance (10)

The Chebyshev distance, also known as the L∞ metric or maximum metric, is a measure of distance in a metric space. It calculates the greatest of the absolute differences between the coordinates of a pair of points. Formally, for two points $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$ in n-dimensional space, the Chebyshev distance is defined as:

$$d(x, y) = \max_{i=1,2\ldots,n} |x_i - y_i|$$

This metric is particularly useful in scenarios where the movement or transformation is restricted to grid-based or orthogonal paths, such as in chess where the King's moves are governed by this distance. It is also employed in various computational and optimization tasks, especially those involving grid-based systems, due to its ability to effectively measure the maximum single-coordinate deviation. Compared to other metrics like Euclidean or Manhattan distances, Chebyshev distance provides a distinct perspective by emphasizing the most significant deviation between the points.

2.2.2 Statistics-based measure

Another way to measure similarity is through statistics-based measure.

2.2.2.1 Pearson correlation (11)

The Pearson correlation coefficient, often denoted as r, is a statistical measure that quantifies the linear relationship between two continuous variables. It ranges from -1 to 1, where r=1 indicates a perfect positive linear correlation, r=−1 indicates a perfect negative linear correlation, and r=0 suggests no linear relationship between the variables. Mathematically, the Pearson correlation is defined as the covariance of the two variables divided by the product of their standard deviations, making it a normalized measure of covariance. This measure assumes that the relationship between the variables is linear and that both datasets are normally distributed. However, it is sensitive to outliers and does not capture non-linear dependencies, so it should be used in conjunction with other analyses when assessing complex relationships.

2.2.2.2 Spearmans' rank correlation coefficient (12)

Spearman's rank correlation coefficient, often referred to as Spearman's ρ is a non-parametric measure of the strength and direction of the association between two ranked variables. Unlike Pearson's correlation, which assesses linear relationships, Spearman's ρ evaluates how well the relationship between two variables can be described using a monotonic function. This measure is particularly useful when the data does not meet the assumptions of linearity or normal distribution. It is calculated by first ranking the data points for each variable and then computing the Pearson correlation on these ranks. Spearman's ρ is scaled between -1 and +1, where +1 indicates a perfect positive monotonic relationship, -1 a perfect negative monotonic

relationship, and 0 implies no association. It is widely used in various fields, due to its robustness to outliers and flexibility in handling ordinal data.

2.2.3 Edit-based distance measures

2.2.3.1 Hamming distance (13)

Hamming distance is a metric used to measure the difference between two strings of equal length by counting the number of positions at which their corresponding symbols differ. It is widely employed in computer science, telecommunications, and bioinformatics for error detection, error correction, and data comparison tasks. For example, in coding theory, Hamming distance helps determine how many bits need to be flipped to convert one binary string into another, providing a way to evaluate the robustness of error-correcting codes. In genomics, it is used to compare DNA or protein sequences to measure genetic variation. The Hamming distance is applicable only when the strings have equal length, and its simplicity makes it a foundational tool in various computational and mathematical applications.

2.2.3.2 Levenshtein distance (14)

Levenshtein distance, also known as edit distance, is a metric for quantifying the difference between two strings by calculating the minimum number of single-character edits required to transform one string into the other. These edits can include insertions, deletions, or substitutions. The Levenshtein distance is widely used in natural language processing, spell checking, DNA sequence analysis, and other applications where string similarity is essential. It is particularly useful in scenarios where small modifications in data can lead to significant changes in meaning or classification. This metric is flexible, allowing a weighted cost for different types of edits, making it adaptable to various practical applications.

2.2.3.3 Damerau-Levenshtein

Damerau-Levenshtein distance extends the concept of Levenshtein distance by accounting for an additional operation: transposition of two adjacent characters (15). It calculates the minimum number of operations—insertions, deletions, substitutions, or transpositions—required to transform one string into another. This metric is especially useful in applications such as spell checking and text similarity analysis, where common typographical errors, like swapped adjacent letters, need to be identified and corrected. By incorporating transpositions, it provides a more robust measure of similarity, particularly in contexts where such errors frequently occur.

2.2.3.4 Jaro-Winkler distance

Jaro-Winkler distance is a string similarity metric that measures how closely two strings match, with a focus on rewarding strings that share a common prefix (16). It builds upon the Jaro distance by assigning higher similarity scores to strings that match more characters in order and have fewer transpositions. The metric is particularly effective in applications like record linkage, where small variations or typos in names or identifiers need to be accounted for. Its emphasis on prefixes makes it well-suited for comparing short strings or those with similar beginnings, such as first and last names.

2.3 Types of similarity learning (17)

2.3.1 Regression similarity learning

In this setup, pairs of objects are given (x_i^1, x_i^2) together with a measure of their similarity $y_i \in \mathcal{R}$. The goal is to learn a function that approximates $f(x_i^1, x_i^2) \sim y_i$ for every new labeled triplet example (x_i^1, x_i^2, y_i) . This is typically achieved by minimizing a regularized loss $\min_w \sum \text{loss}(w; x_i^1, x_i^2, y_i) + \text{reg}(w)$

2.3.2 Classification similarity learning

Given are pairs of similar objects (x_i, x_i^+) and non-similar objects (x_i, x_i^-) . An equivalent formulation is that every pair (x_i, x_i^+) is given together with a binary label $y_i \in \{0; 1\}$ that determines if the two objects are similar or not. The goal is again to learn a classifier that can decide if a new pair of objects is similar or not.

2.3.3 Ranking similarity learning

Given are triplets of objects (x_i , x_i^+, x_i^-) whose relative similarity obey a predefined order: x_i is known to be more similar to x_i^+ than to x_i^-. The goal is to learn a function f such that for any new triplet of objects, (x_i , x_i^+, x_i^-), it obeys $f(x_i, x_i^+) > f(x_i, x_i^-)$ (**contrastive learning**). This setup assumes a weaker form of supervision than in regression, because instead of providing an exact **measure of similarity**, one only must provide the relative order of similarity. For this reason, ranking-based similarity learning is easier to apply in real large-scale applications.

2.4 Some existing methods

A common approach for learning similarity is to model the similarity function as a bilinear form. For example, in the case of ranking similarity learning, one aims to learn a matrix W that parametrizes the similarity function $f_W(x, z) = x^T Wz$. When data is abundant, a common approach is to learn a **siamese network** (see section 2.4.1) – a deep network model with parameter sharing.

2.4.1 Siamese network [1]

Siamese networks are one of the foundational architectures in similarity learning. These networks consist of two identical sub-networks that share parameters and are trained to compute embeddings for the inputs. The primary objective is to minimize the distance between embeddings of similar pairs while maximizing the distance for dissimilar pairs. A notable application is face verification (18), where the network learns whether two images belong to the same person.

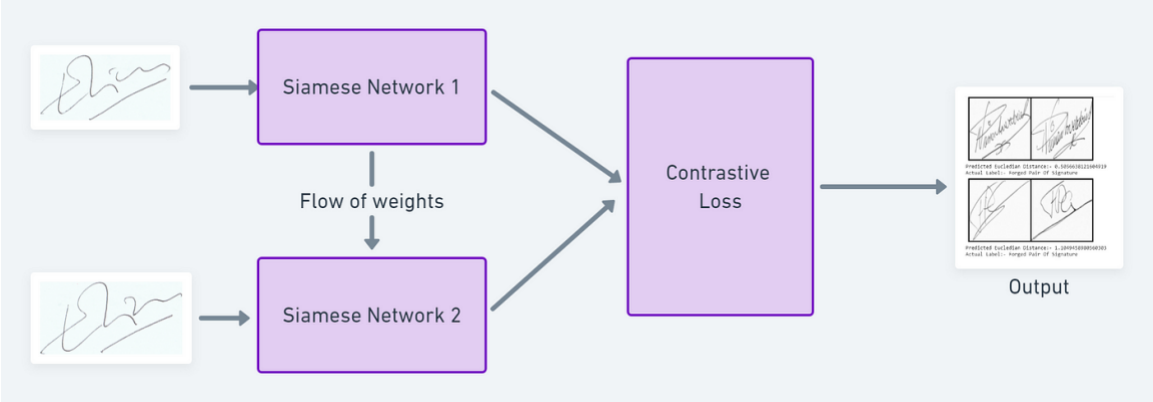


Figure 1: Siamese Network basic architecture [2]

The key features of a traditional Siamese networks include:

- **Parameter Sharing**: Both branches of the network have identical weights, ensuring that the embedding space is consistent across inputs.
- **Contrastive Loss**: This loss function penalizes large distances for similar pairs and small distances for dissimilar pairs.
- **Strengths**: These networks excel in tasks with limited labeled data by focusing on pairwise relationships rather than absolute labels.

Siamese networks can effectively leverage unlabeled data through **semi-supervised learning techniques** (19). One approach involves training the network to learn a similarity function using a small set of labeled data. The network is then used to generate embeddings for the unlabeled data, which can be clustered based on similarity. Confident predictions from these clusters are treated as pseudo-labels and

[1] https://github.com/sohaib023/siamese-pytorch
[2] "A friendly introduction to Siamese Network" – Sean Benhur (Medium.com)

incorporated into the training process, iteratively refining the model. This method allows the network to utilize the structure of the unlabeled data to improve performance.

Another strategy employs **self-supervised learning** (20), where the network is trained on auxiliary tasks that do not require labels, such as ranking or reconstruction tasks. By learning to perform these tasks, the network captures meaningful representations of the data, which can then be fine-tuned with the available labeled data for the primary task.

Additionally, consistency-based training methods (21) can be applied, where the network is encouraged to produce consistent outputs for perturbed versions of the same input. This approach helps the model become more robust and better generalize from limited labeled data by effectively utilizing the information present in the unlabeled data.

Siamese Network are a great structure point to develop a model in similarity learning, however they face challenges in handling triplet relationships and scaling to large datasets due to the quadratic growth of pair combinations. We will see later on how these challenges can be faced and dealt with.

2.4.2 Locality sensitive hashing (Approximate-search algorithm) [3]

Locality-sensitive hashing maps objects into the same bucket with high probability if they are similar. **Hashes** input items so that similar items map to the same "buckets" in memory with high probability (the number of buckets being much smaller than the universe of possible input items). It is often applied in nearest neighbor search on large-scale high-dimensional date, e.g, image databases, document collections, time-series databases, and genome databases (22). LSH is commonly used in large-scale applications where exact similarity computation is computationally infeasible. It is highly efficient but may sacrifice precision depending on the hash function and data distribution.

2.4.3 Graph convolutional network (Graph-based algorithm)

Graph Convolutional Networks (GCNs) are a class of deep learning algorithms designed to process data represented as graphs. Unlike traditional convolutional neural networks that work on grid-like data (e.g., images), GCNs operate on nodes and edges of a graph, learning to propagate and aggregate information across the graph structure. At each layer, GCNs update node embeddings by combining information from a node's neighbors, allowing them to capture relational and structural information within the graph. This makes GCNs particularly effective for

[3] https://github.com/cchatzis/Nearest-Neighbour-LSH

similarity learning tasks in domains where objects have complex interrelationships, such as social networks, recommendation systems, and molecular chemistry (23). For instance, GCNs can learn representations of nodes (e.g., users, items, or molecules) and use these embeddings to compute similarities based on shared graph features or proximity.

More broadly, graph-based algorithms are highly relevant in similarity learning due to their ability to naturally model relationships and interactions between objects. Methods like spectral clustering and random walk-based similarity (24) leverage graph structures to identify communities or compute similarity scores by examining the connectivity patterns of nodes. Graph embeddings (25), another approach, transform nodes or subgraphs into vector representations, enabling the use of traditional similarity measures (e.g., cosine similarity) while preserving graph semantics. These algorithms are vital for tasks like semi-supervised learning, anomaly detection, and collaborative filtering, where the relationships between entities provide crucial context. Overall, graph-based approaches offer a powerful and flexible framework for learning and assessing similarity, especially in settings where data is inherently relational or topological.

2.4.4 Kernel methods

Kernel methods are a class of algorithms widely used in machine learning to handle non-linear data by transforming it into a higher-dimensional space where linear techniques can be effectively applied. The transformation is performed implicitly through the use of kernel functions (26), which compute the inner product of data points in this higher-dimensional space without explicitly mapping the data. Common kernel functions include the linear kernel, polynomial kernel, Gaussian (RBF) kernel, and sigmoid kernel, each suited to different types of data and tasks. These methods are particularly powerful in similarity learning, as they enable the comparison of data points in complex, non-linear feature spaces. Kernel methods are most famously utilized in Support Vector Machines (SVMs) and Principal Component Analysis (PCA) variants like Kernel PCA (27). In similarity learning, kernel-based approaches allow the development of flexible similarity measures that adapt to the underlying structure of the data, making them suitable for applications like image recognition, text categorization, and bioinformatics. Their ability to work with non-linearly separable data while retaining computational efficiency makes kernel methods an indispensable tool in machine learning.

2.4.5 Autoencoders [4]

Autoencoders are neural network architectures designed to learn efficient representations of data by compressing input into a latent space and reconstructing

[4] https://frcs.github.io/4C16-LectureNotes/autoencoders.html

it back to its original form. They operate in an unsupervised manner, as they do not require labeled data; the training objective is simply to minimize the reconstruction error between the input and output. Autoencoders consist of two main components: an encoder that maps the input data to a compressed latent representation and a decoder that reconstructs the original data from this representation. In similarity learning, the latent representations generated by the encoder capture the most salient features of the input data, enabling comparison of objects in this compact space. This makes autoencoders particularly useful for tasks like dimensionality reduction, clustering, and anomaly detection, where similarity can be assessed based on proximity in the latent space. For instance, similar data points will have closer representations in the latent space, providing a natural way to measure similarity without relying on explicit supervision. Their ability to extract meaningful features in an unsupervised manner makes autoencoders highly relevant for applications such as image matching, text similarity, and recommender systems.

2.4.6 Partially ordered sets

The concept of **Partially Ordered Sets (Posets)** has been effectively utilized in unsupervised similarity learning to enhance the learning of visual similarities without the need for labeled data. A notable approach involves initially grouping samples into compact surrogate classes based on local estimates of reliable similarities (28). These surrogate classes are then linked through local partial orders, establishing a structured relationship among them. This methodology formulates similarity learning as a partial ordering task, where soft correspondences of all samples to classes are employed. By adopting a self-supervised strategy, a Convolutional Neural Network (CNN) is trained to represent samples in a mutually consistent manner while concurrently updating the surrogate classes. This integrated model has demonstrated competitive performance in tasks such as detailed pose estimation and object classification, underscoring the efficacy of poset-based frameworks in capturing fine-grained visual similarities in an unsupervised setting.

2.5 Specific loss functions commonly used

2.5.1 Contrastive loss

Contrastive loss is one of the most used loss functions in Siamese networks, designed to differentiate between similar and dissimilar data points. It works by taking pairs of data points—an anchor and a comparison point—and encouraging similar pairs to be close in the embedding space while pushing dissimilar pairs apart. Contrastive loss is computationally efficient as it only requires handling pairs of data rather than larger sets, making it a popular choice for similarity learning tasks. Additionally, recent advancements in self-supervised learning have extended contrastive loss to

unsupervised scenarios, with methods such as SimCLR and BYOL (29) using augmented data pairs to learn meaningful representations.

2.5.2 Triplet loss

Building on contrastive loss, triplet loss introduces a more nuanced approach by incorporating three data points—an anchor, a positive example, and a negative example. The goal of triplet loss is to ensure that the anchor is closer to the positive example than to the negative one by a predefined margin, thereby improving the model's ability to capture relative similarity relationships. This method is particularly useful in applications like face recognition and person re-identification, where distinguishing between highly similar instances is crucial. However, triplet loss requires more training time compared to contrastive loss due to the need for careful triplet selection and the increased number of samples involved in the process. Despite its complexity, triplet loss provides greater flexibility in ranking-based similarity tasks, ensuring that the learned embeddings maintain the relative order of data points.

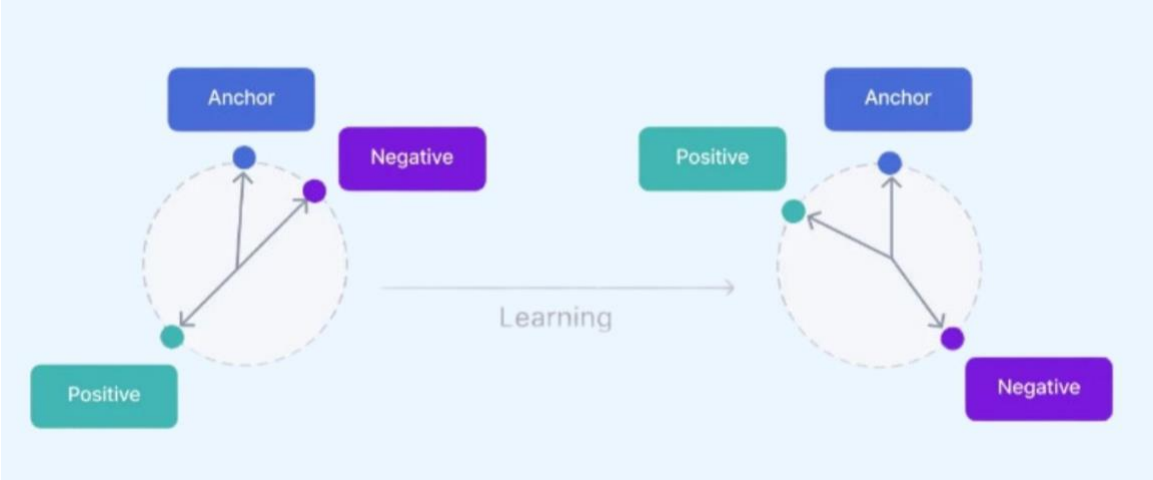


Figure 2: Triplet loss function [5]

2.6 Applications in Diverse Domains

2.6.1 Image similarity search (30)

Convolutional Neural Networks (CNNs) have been extensively utilized to extract features for image similarity tasks. A notable example is the Online Algorithm for Scalable Image Similarity (OASIS), which learns a bilinear similarity measure over

[5] from Big Blue Data Academy

sparse representations. OASIS employs an online dual approach using the passive-aggressive family of learning algorithms with a large margin criterion and an efficient hinge loss cost. Experiments have demonstrated that OASIS is both fast and accurate across various scales, achieving superior results compared to existing state-of-the-art methods while being significantly faster. For instance, on a dataset with thousands of images, OASIS outperformed other methods and, on a larger scale, was trained on over two million images within three days on a single CPU.

2.6.2 Code Similarity (31)

Deep learning techniques have been applied to assess similarities between source code components by leveraging representations such as Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs). For example, the Flow-Augmented Abstract Syntax Tree (FA-AST) enhances original ASTs with explicit control and data flow edges. Graph Neural Networks (GNNs) are then applied to these FA-ASTs to measure the similarity of code pairs. This approach has outperformed state-of-the-art methods in detecting code clones, which are semantically similar code fragments that may not share syntactic similarities.

2.6.3 Categorical Data

Meta-learning approaches have been proposed to automate the selection of similarity measures for partitioning categorical datasets, addressing the diversity of data types. These methods aim to learn how to learn, enabling models to adapt to new tasks or datasets efficiently by leveraging prior knowledge. This is particularly useful in scenarios where the optimal similarity measure is not apparent due to the heterogeneous nature of the data.

2.6.4 Remote Sensing (32)

In remote sensing, CNN-based feature maps have been employed for landscape similarity analysis by combining texture and spatial features to compare aerial imagery effectively. By extracting and integrating these features, models can assess the similarity between different regions, aiding in tasks such as land cover classification, change detection, and environmental monitoring. This approach enhances the ability to analyze complex patterns in large-scale geospatial data.

These applications demonstrate the versatility of similarity learning across various domains, highlighting the importance of tailored approaches to effectively capture and compare the intrinsic characteristics of diverse data types.

2.7 Challenges and Limitations

2.7.1 Common challenges and limitations

Existing similarity learning methods face the same limitations as other deep learning fields.

- **Scalability**: Techniques like Siamese and triplet networks struggle with large datasets due to the need for pairwise or triplet comparisons, which grow quadratically.
- **Data Quality**: Noise, outliers, and missing labels can significantly impact performance.
- **Generalization**: While methods like CNNs achieve high accuracy on specific tasks, they may overfit and fail to generalize to unseen data.

2.7.2 Challenges specific to similarity learning

There are many challenges that are specific to similarity learning.

- **Defining Similarity:** Defining what constitutes similarity is a fundamental challenge in similarity learning. Similarity is often subjective and highly dependent on the specific application, making it difficult to establish a universal measure that works across different domains and data types.
- **Unsupervised Methods:** Many similarity learning tasks lack labeled data, making it challenging to train models without supervision. Unsupervised approaches, such as clustering or contrastive learning, can struggle to capture meaningful relationships and often require careful tuning and validation against weak or proxy signals.
- **Scalability and Efficiency:** Similarity learning often involves comparing large numbers of data points, which can be computationally expensive and memory intensive. Efficient methods are needed to handle high-dimensional data and large-scale datasets while maintaining speed and accuracy.
- **Metric Selection:** Choosing the right similarity metric is crucial, as different metrics (e.g., Euclidean, cosine, learned distance functions) can yield vastly different results. The choice of metric depends on the data characteristics and application goals, and an inappropriate selection can negatively impact performance.
- **Negative Sampling:** In contrastive and triplet-based learning approaches, selecting appropriate negative samples is critical for effective model training. Poor sampling strategies can result in trivial or uninformative comparisons, leading to suboptimal learning and performance degradation.
- **Intra-class and Inter-class Challenges:** Similarity learning must cope with high intra-class variability, where items within the same category may look different, and inter-class similarities, where items from different categories

may appear similar. This makes it difficult to create clear decision boundaries and often results in misclassification.

- **Scalability of Similarity Search:** When deploying similarity models in real-world applications, efficient retrieval methods are required to search through massive databases. Approximate nearest neighbor search and indexing techniques must strike a balance between speed, memory usage, and accuracy.

3 Implementation of a Siamese Network

This section provides a detailed overview of the implemented Siamese Network for similarity learning, focusing on its architecture, training process, evaluation, and results. You can find the python code in the appendices.

3.1 Network Architecture

The Siamese Network implemented for this project is a neural architecture specifically designed to learn embeddings for input pairs and calculate their similarity. The network is built to process high-dimensional inputs (e.g., images) and generate embeddings that represent the inputs in a compact, meaningful feature space.

The embedding subnetwork is built as a fully connected neural network with multiple layers designed to learn meaningful embeddings for input data with an initial dimension of 784. The architecture consists of a series of fully connected layers that progressively reduce the dimensionality of the input while applying nonlinear transformations. The network begins with a 1024-dimensional layer, followed by subsequent layers with decreasing sizes of 512, 128, and finally an embedding dimension of 64. Each layer is followed by a ReLU activation function to introduce non-linearity, and LayerNorm to stabilize training by normalizing the activations. Dropout is applied at different rates (0.3 and 0.5) to prevent overfitting and enhance generalization. The final output of the network is a 64-dimensional embedding that captures the essential features of the input, which can then be compared using a distance metric to determine similarity.

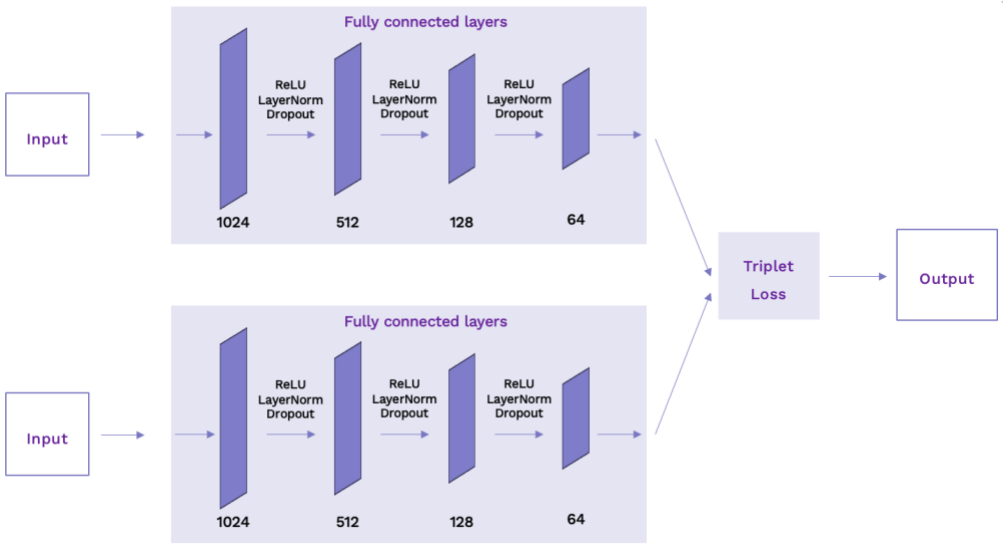


Figure 3: Siamese Network architecture

To enhance the network's capability, an *attention mechanism* is integrated into the architecture. This attention module assigns dynamic importance to different dimensions of the embeddings, effectively enabling the network to focus on the most relevant features for similarity computation. The attention mechanism applies a two-layer architecture: the first layer transforms the embedding using a Tanh activation, while the second layer applies a Softmax function to produce attention scores that sum to one. These scores are then used to weight the embeddings dynamically, ensuring that the network emphasizes the most critical features. To maintain stability, the attention weights are clamped between 0 and 1, and the weighted embeddings are normalized using L2 normalization.

3.1.1 Comparison of distance metrics

In the evaluation of the Siamese network, three different distance-based similarity metrics were tested to determine their effectiveness in distinguishing between similar and dissimilar data points. These metrics included **Euclidean distance**, **cosine similarity**, and a **combination of multiple metrics** (cosine similarity, Euclidean distance, and dot product). The model was trained and tested under each configuration to observe differences in performance.

3.1.1.1 Euclidean distance

For the Euclidean similarity metric, the model achieved a relatively poor test accuracy of **0.0102**, indicating that Euclidean distance alone was not effective in capturing meaningful relationships between embeddings. The visualizations of the learned embeddings using LDA and t-SNE showed a high degree of overlap among different classes, further highlighting the metric's limitations in this context.

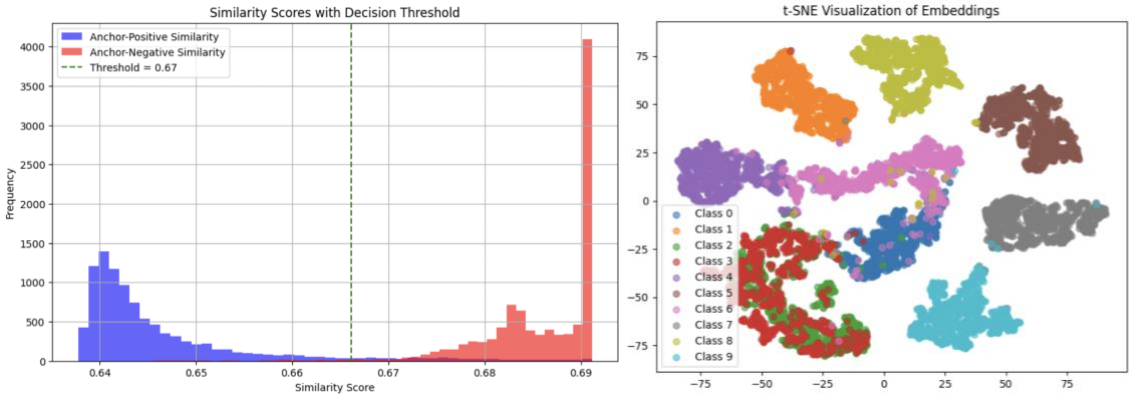


Figure 4: Similarity scores and t-SNE (Euclidean distance)

3.1.1.2 Cosine distance

On the other hand, the cosine similarity metric significantly improved the model's performance, yielding a test accuracy of **0.9827**. The embeddings were better clustered with clearer separation among classes, as shown in the LDA and t-SNE visualizations. The similarity score distribution also demonstrated a distinct separation between positive and negative pairs, suggesting that cosine similarity effectively captured the underlying structure of the data.

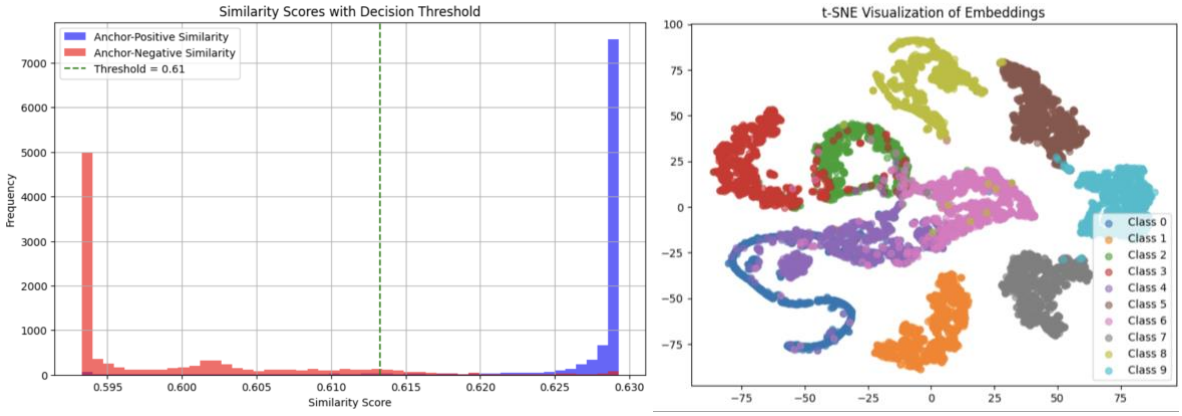


Figure 5: Similarity scores and t-SNE (Cosine distance)

3.1.1.3 Combination of distance metrics

Finally, a combined approach that leveraged multiple similarity measures (cosine similarity, Euclidean distance, and dot product) provided the best results, achieving a test accuracy of **0.9930**. This approach benefited from the strengths of each metric, leading to more discriminative embeddings and well-separated clusters in the visualization plots. The similarity score distribution further reinforced the model's robustness, with minimal overlap between anchor-positive and anchor-negative pairs.

We thus decided that the final model similarity computation module would use the *combination of distance metrics* to calculate how similar two embeddings are. The combination of these metrics is handled by a fusion module, which is a small fully connected layer followed by a Sigmoid activation. This module dynamically adjusts the weighting of the different metrics, ensuring the network can adapt to various types of similarity relationships. The resulting similarity score is scaled between 0 and 1, providing a clear and interpretable measure of similarity.
This architecture combines innovation and robustness, leveraging Layer Normalization, dropout regularization, dynamic attention mechanisms, and multi-metric similarity computation. By combining these elements, the network is well-suited for high-dimensional similarity learning tasks, demonstrating the flexibility to focus on critical features and adapt to diverse input relationships.

3.2 Loss Function

The implementation of the **Triplet Loss** in this project introduces a robust method for learning embeddings that preserve relative distances between anchor, positive, and negative examples. This loss function operates by computing two distances: the distance between the anchor and the positive example (representing similar pairs) and the distance between the anchor and the negative example (representing dissimilar pairs). Using these distances, the loss is calculated as the mean of the difference between the positive distance and the negative distance, augmented by a margin. This margin acts as a buffer, ensuring that the network not only learns to distinguish between similar and dissimilar pairs but also enforces a minimum separation between them in the embedding space. The use of torch.clamp ensures that only meaningful contributions are made to the loss when the positive distance exceeds the negative distance plus the margin, effectively ignoring cases where the triplet is already well-separated. This dynamic and intuitive loss function is critical for tasks that require fine-grained discrimination of relative similarities, such as ranking-based similarity learning or face verification.

3.3 Training of the model

The training process of the Siamese network is designed to maximize the effectiveness of the learned embeddings for similarity learning tasks. It focuses on the use of triplets of inputs, where each triplet consists of an anchor input, a positive input (similar to the anchor), and a negative input (dissimilar to the anchor). For each triplet, embeddings are generated by passing the inputs through the shared neural network. The Triplet Loss is then computed to encourage the anchor and positive embeddings to be closer together in the embedding space than the anchor and negative embeddings. The loss ensures that the difference between the distances (anchor-positive and anchor-negative) exceeds a predefined margin, pushing the network to learn representations that separate similar and dissimilar examples.
The training process was performed using GPU acceleration to manage the high computational demands of processing large datasets. The time complexity of different distance computations was also analyzed to ensure real-time applicability in production environments.

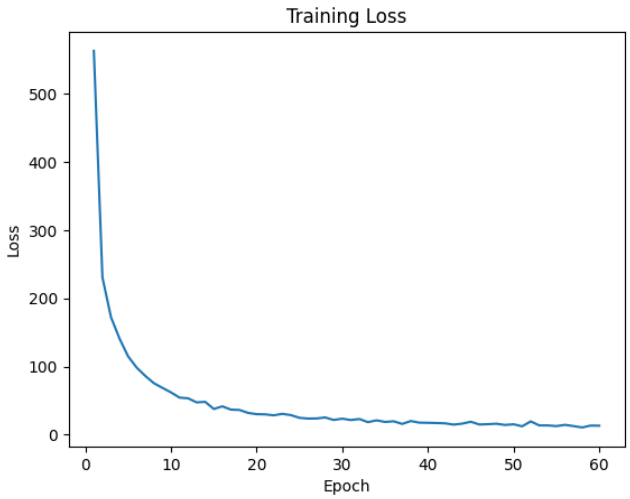


Figure 6 : Training Loss plot

3.3.1 Use of a dynamic margin

A critical aspect of this implementation is the use of a dynamic margin within the Triplet Loss function. The margin starts at a lower value and increases gradually over the course of training. This adjustment allows the network to focus initially on learning a basic distinction between positive and negative embeddings, without being constrained by overly strict separation requirements. As training progresses, the margin increases, forcing the model to refine the embedding space further and create more pronounced distinctions between classes. This gradual adjustment ensures that the model balances simplicity in the early stages of training with complexity as it becomes more refined.

3.3.2 Hard negative mining

Hard negative mining is another integral part of the training strategy. Hard negatives are negative examples that are particularly challenging because their embeddings are close to the anchor embedding, making them harder to distinguish from positives. During training, the model dynamically identifies these challenging examples by evaluating the distances between candidate negatives and the anchor embeddings. Those with the smallest distances—i.e., those closest to the anchor in the embedding space—are selected as hard negatives. This approach forces the model to focus on the most difficult cases, improving its ability to distinguish subtle differences and learn highly discriminative features. Without this step, the network could waste time on "easy negatives" that are already far from the anchor and provide little value for further optimization.

Regularization techniques such as weight decay are employed to improve the generalization of the network by penalizing large weights, thereby preventing

overfitting. To ensure computational efficiency, batches of triplets are generated dynamically during training. This approach reduces memory usage while allowing for the creation of diverse and challenging triplets for each training iteration.
Overall, the combination of dynamic margin adjustment and hard negative mining ensures the network learns embeddings that are both highly discriminative and robust. Dynamic margins enable the network to adapt its learning focus over time, while hard negative mining pushes the network to address its most challenging examples. Together, these strategies lead to an effective training pipeline for similarity learning tasks, preparing the network for tasks such as ranking, clustering, and retrieval with complex and nuanced data.

3.3.3 Hyperparameter tuning

Key hyperparameters such as learning rate, batch size, dropout rates, and weight initialization were tuned to optimize model performance. A combination of grid search and random search was employed to identify the optimal values. The impact of different hyperparameter choices was analyzed by evaluating the model's performance across multiple validation runs.

3.3.4 Datasets used for training

3.3.4.1 Images : Fashion-mnist

The **Fashion-MNIST** dataset, created by Zalando Research, is a widely used benchmark dataset designed as a drop-in replacement for the original MNIST dataset of handwritten digits. It contains **70,000 grayscale images** of size **28x28 pixels**, categorized into **10 fashion-related classes**, such as T-shirts, trousers, dresses, and sneakers. The dataset is divided into **60,000 training images** and **10,000 test images**, making it suitable for evaluating machine learning models in tasks related to image classification. Unlike the original MNIST dataset, which consists of simple handwritten digits, Fashion-MNIST offers more complex and visually diverse patterns, making it a more challenging and representative dataset for modern computer vision applications. Each class in the dataset is labeled from 0 to 9, representing different clothing categories, and the dataset is often used to test deep learning architectures such as convolutional neural networks (CNNs) and similarity learning models like Siamese networks. Due to its accessibility and lightweight nature, Fashion-MNIST is widely adopted in academic and research settings for developing and benchmarking machine learning algorithms in fashion-related classification tasks.

3.3.4.2 Numbers : mnist

The **MNIST (Modified National Institute of Standards and Technology) dataset** is a well-known benchmark dataset in the field of machine learning and computer vision, primarily used for handwritten digit classification. It consists of **70,000 grayscale images**, each of size **28x28 pixels**, representing the digits 0 through 9. The dataset is split into **60,000 training images** and **10,000 test images**, providing a standardized benchmark for evaluating classification algorithms. MNIST has been widely used in the development and testing of deep learning models, especially in early-stage research on convolutional neural networks (CNNs), autoencoders, and other machine learning techniques. Each image contains a centered, noise-free representation of a single handwritten digit, making it a relatively simple dataset that is ideal for learning fundamental machine learning concepts and debugging algorithms. Despite its simplicity, MNIST remains a popular choice for testing model architectures, optimization techniques, and performance evaluation due to its ease of use and accessibility.

3.4 Evaluation

The model evaluation is conducted using the test_model function, which assesses the performance of the Siamese network on a test dataset. This function switches the model to evaluation mode to disable gradient computations, ensuring a more efficient and memory-friendly inference process. The test process iterates through batches of anchor, positive, and negative samples, transferring them to the appropriate computing device (either CPU or GPU). The model generates embeddings for the anchor, positive, and negative inputs, and their pairwise similarities are computed. The accuracy of the model is determined by counting instances where the similarity score between the anchor and positive example is higher than that of the anchor and negative example, reflecting the model's ability to distinguish between similar and dissimilar pairs effectively. Additionally, all computed similarity scores are stored for further analysis, such as visualizing score distributions to gain deeper insights into the model's performance. Finally, the function computes and prints the overall test accuracy, providing a quantitative measure of the model's generalization capability on unseen data.

3.5 Results

The results of the model evaluation, as depicted in **Figure 7**, indicate that the similarity scores for the test pairs predominantly fall within a narrow range, specifically between 0.55 and 0.65. Despite this limited numerical range, the model effectively distinguishes between similar and dissimilar pairs. The similarity score distribution graph shows that dissimilar pairs tend to have scores clustered around 0.55, whereas similar pairs are closer to 0.65. This clear separation suggests that the model successfully captures meaningful relationships between the input pairs. The t-SNE visualization further reinforces this observation by displaying well-separated

clusters of embeddings corresponding to different classes, highlighting the model's ability to learn discriminative feature representations. The overall performance analysis, combined with the accuracy metric, confirms that the model generalizes well to unseen data and is effective in distinguishing between positive and negative pairs within the given threshold.

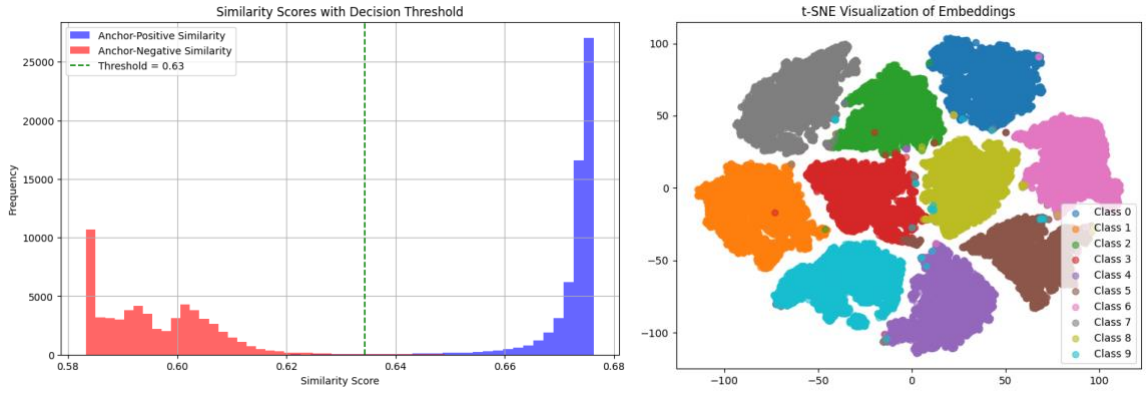


Figure 7: Results (mnist dataset)

4 Implementation of an Autoencoder

The implementation of an unsupervised method is much harder than implementing a supervised neural network. They are less existing methods and documentation on this topic. The first method we tried to implement was through an autoencoder.

4.1 Data preprocessing

The images are prepared using the preprocess_image function:

- **Transformation**: Images are resized to 128x128 pixels, and random augmentations (rotations, brightness changes, etc.) are applied.
- **Conversion**: Images are converted into normalized tensors, making the data compatible with a neural network model.

This preprocessing ensures that variations in real-world data do not negatively affect the model's performance.

To first test the model, we used 3 different types of images: Images of golden retriever dog, Images of german sheperd dogs and images of trees.

4.2 Architecture of the model

4.2.1 Enhanced convolutional encoder

The Enhanced Convolutional Autoencoder (CAE) is designed to learn compact representations of images while ensuring that these representations are meaningful and effective for downstream tasks such as clustering. Its architecture includes two main components: the encoder and the decoder. The encoder reduces the dimensionality of input images by extracting key features through convolutional layers, while the decoder reconstructs the original images from the compressed features to validate the quality of the encoding process. Training this model involves the use of two loss functions. The first, Mean Squared Error (MSE), ensures faithful reconstruction of the input images. The second, contrastive loss, enforces separation between dissimilar examples and pulls similar examples closer together in the learned feature space, improving the quality of the representations.

4.2.1.1 Why is it better than a traditional Autoencoder?

The Enhanced CAE offers several advantages compared to a traditional autoencoder, particularly in its ability to process image data effectively.

The first advantage is its use of convolutional layers instead of fully connected layers. Convolutional layers are inherently better suited for image data because they preserve the spatial relationships between pixels, capturing patterns such as edges and textures. Additionally, convolutional layers are more parameter-efficient, which reduces the number of trainable parameters and, consequently, the risk of overfitting. This makes the model not only more effective but also computationally efficient.
The second improvement comes from the use of batch normalization, which is applied after the activation functions in the network. Batch normalization stabilizes and accelerates training by normalizing intermediate feature distributions, reducing sensitivity to learning rate changes, and preventing issues such as exploding or vanishing gradients.

The third advantage of the Enhanced CAE is its ability to extract hierarchical features through multiple convolutional layers in the encoder. Early layers in the network focus on detecting basic features such as edges or simple colors, while deeper layers capture more abstract and high-level patterns. This hierarchical feature extraction makes the learned representations richer and more meaningful for downstream tasks.

In the decoder, transposed convolutions are used to upsample the compact features and reconstruct the original image. This ensures that the reconstructions closely resemble the input images while preserving important features during the upsampling process. By maintaining the integrity of the reconstructed images, the decoder indirectly validates the encoder's ability to compress and retain key information.

Finally, the addition of contrastive loss further enhances the quality of the embeddings learned by the model. This loss function ensures that dissimilar samples are well-separated in the feature space, while similar samples are drawn closer together. This characteristic makes the learned embeddings more effective for clustering and other unsupervised learning tasks.

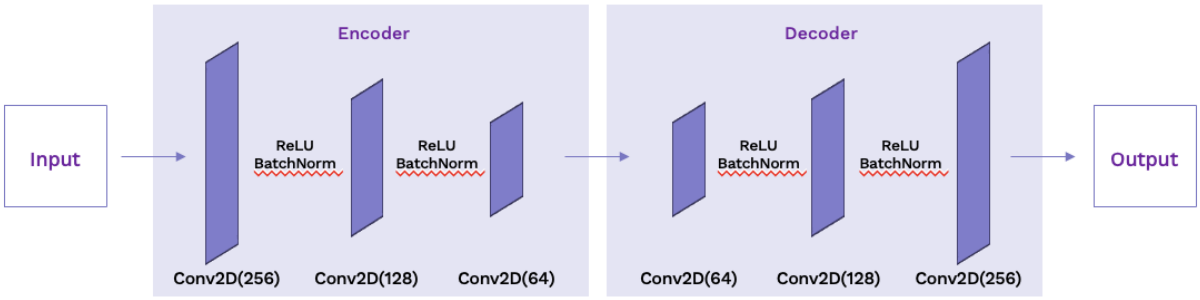


Figure 8: Autoencoder architecture

4.2.1.2 Why do we increase the data dimensions?

While the goal of an autoencoder is to learn compact representations, the decoder intentionally increases the data dimensions during reconstruction. This seemingly counterintuitive step has a specific purpose in the learning process.
The encoder reduces the input image to a compact representation by stripping away unnecessary information while retaining the essential features. This reduction serves as the primary form of dimensionality reduction in the autoencoder pipeline. However, the decoder's role is to reconstruct the image back to its original resolution, which necessitates an increase in dimensionality. This step is critical because it allows the model to verify whether the encoder's embeddings retain all the information required to accurately recreate the input. By demanding high-resolution reconstructions, the decoder ensures that the embeddings are robust and truly represent the original data.

Although this reconstruction increases the data size temporarily, the compact embeddings generated by the encoder remain the ultimate output. These embeddings are used for tasks such as clustering, visualization, or classification. Thus, the increase in data size during reconstruction is an intermediate step aimed at ensuring the quality of the learned representations, rather than a contradiction of the goal of dimensionality reduction.

Below are the comparison of the results between a Traditional Encoder that only uses Linear layers and our Enhanced version that uses Convolutional Layers and Batch Normalization.

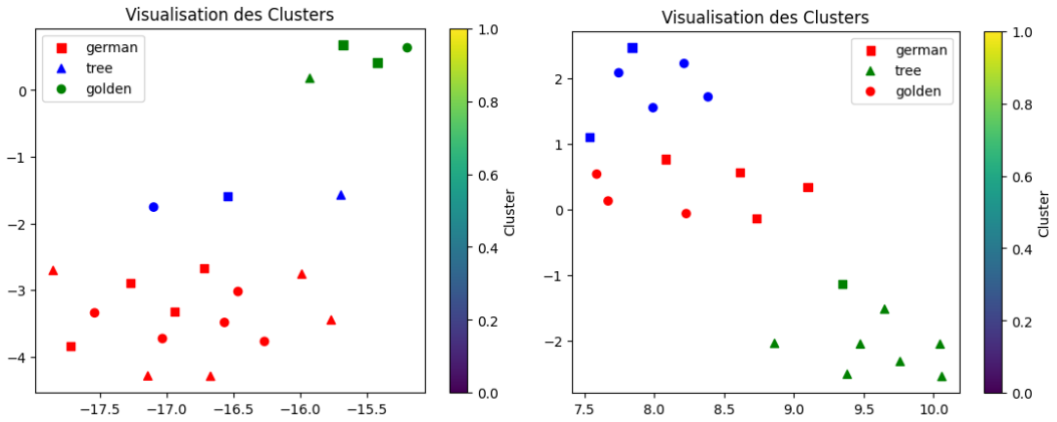


Figure 9: Cluster visualization (left: Traditional, right: Enhanced)

After the encoder reduces the dimensionality of the input data, further dimensionality reduction is applied using UMAP to bring the embeddings down to two dimensions. This step facilitates visualization and clustering. The reduced embeddings are then processed by the K-Means algorithm, which assigns cluster labels based on the learned representations.

The temporary increase in data size during the reconstruction step does not affect the final compact embeddings used for clustering. In essence, the dimensionality reduction occurs where it matters most—at the level of the embeddings. This ensures that the overall process remains efficient and focused on extracting meaningful and compact representations.

4.3 Training and dimensionality reduction

The model is trained on the dataset using the Adam optimizer. The weights are adjusted to minimize the combined loss, allowing the encoder to produce meaningful embeddings (vectors).
The extracted embeddings are normalized with StandardScaler and reduced to two dimensions using **UMAP (Uniform Manifold Approximation and Projection)**, a dimensionality reduction method that effectively preserves local relationships.

4.3.1 Comparison of different training loss functions

In our autoencoder training function, we are using two loss functions: **Mean Squared Error (MSE) loss** and **Contrastive loss**, each serving a distinct purpose in training.
The MSE loss is used for reconstruction, ensuring that the output of the autoencoder is as close as possible to the input images. It computes the average squared difference between the original and reconstructed images, penalizing large reconstruction errors more heavily. This loss helps the autoencoder learn to compress and reconstruct data effectively by minimizing the pixel-wise difference.

Contrastive loss is a distance-based loss function used to learn feature embeddings that preserve similarity relationships within the data. In your implementation, it operates on the embeddings extracted from the encoder and calculates pairwise Euclidean distances. The goal is to minimize the distance between similar samples (where the distance is below a threshold, e.g., 1.0) and maximize it for dissimilar samples. The loss encourages the model to cluster similar images together in the embedding space while pushing dissimilar ones apart. It consists of two terms:

- **Similar pairs loss**: Encourages embeddings of similar items to be closer by squaring small distances.
- **Dissimilar pairs loss**: Ensures embeddings of different items are sufficiently apart by penalizing them when they are too close.

In combination, these loss functions allow the autoencoder to not only reconstruct images accurately but also to learn a meaningful embedding space where similar images are closer and dissimilar ones are farther apart. This approach is particularly useful for tasks like anomaly detection, clustering, and similarity-based retrieval.
We also tested others loss functions: MSE Loss combined with Triplet Loss on the one hand, and Perceptual Loss and Triplet Loss on the other hand. Here are the results below of the embeddings with these pairs.

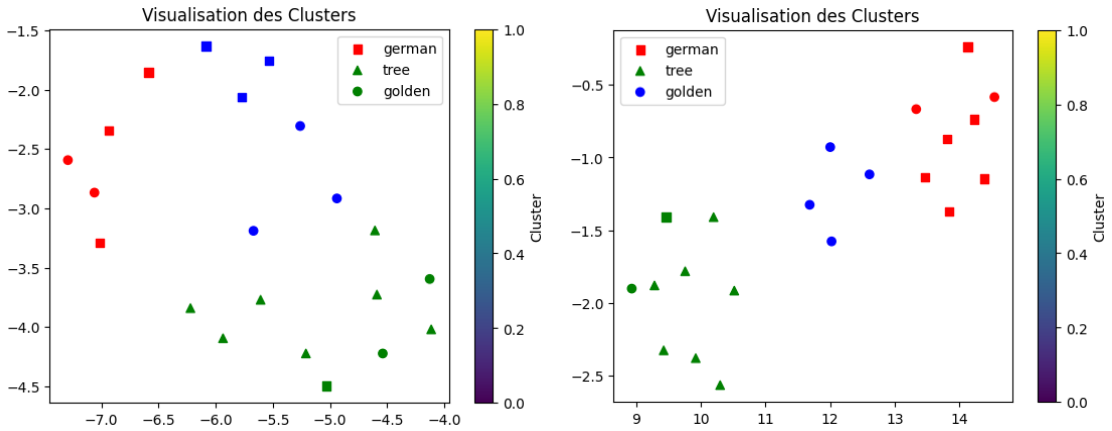


Figure 10: Model with Triplet Loss and MSE Loss (left) and Triplet Loss and perceptual loss (right)

The results with SME combined with Triplet Loss are far worse than the initial results (Fig.9). That could be explained by the fact that **Triplet Loss**, unlike Contrastive Loss, focuses on relative comparisons within triplets (anchor, positive, and negative samples), which may introduce challenges in efficiently learning embeddings when the triplet selection is not optimal. It requires careful mining of hard negative samples to achieve good performance, and poorly selected triplets can result in slow convergence or suboptimal embeddings. Contrastive Loss, on the other hand, provides a more direct optimization by considering pairwise relationships, potentially offering a more stable training process when the dataset contains noisy or imbalanced data. Also, when combining **MSE and Triplet Loss**, the MSE loss may not provide a strong enough supervisory signal for learning fine-grained representations in the latent space. MSE primarily focuses on pixel-level reconstruction and does not capture high-level perceptual similarities effectively, potentially conflicting with the embedding structure imposed by Triplet Loss, which operates in a different objective space.

The results combining Perceptual Loss and Triplet Loss are better but still not as good as our final model displayed Fig.9. Indeed, **Perceptual Loss**, which relies on deep feature representations extracted from pre-trained models, may not align well with Triplet Loss if the perceptual features emphasize content similarities differently from the relationships learned by the embedding network. This mismatch could lead to inconsistent gradients and difficulty in achieving a well-structured latent space that generalizes across similar and dissimilar samples.

4.4 Clustering and visualization

The reduced embeddings are clustered using the **K-Means algorithm**: Three clusters (n_clusters=3) were specified, representing the main groups in the data. Cluster labels are assigned to each data point.
The clusters are visualized in a plot where:

- Colored points represent the identified clusters.
- Point shapes indicate the image categories: "golden retriever" dogs (circle), "German shepherd" dogs (square), and trees (triangle).
- A legend and a color bar help interpret the results.

4.5 Results

We decided to test our model on the mnist dataset to be able to assess the model ability to separate well different classes with more data. Below the embeddings with the mnist dataset.

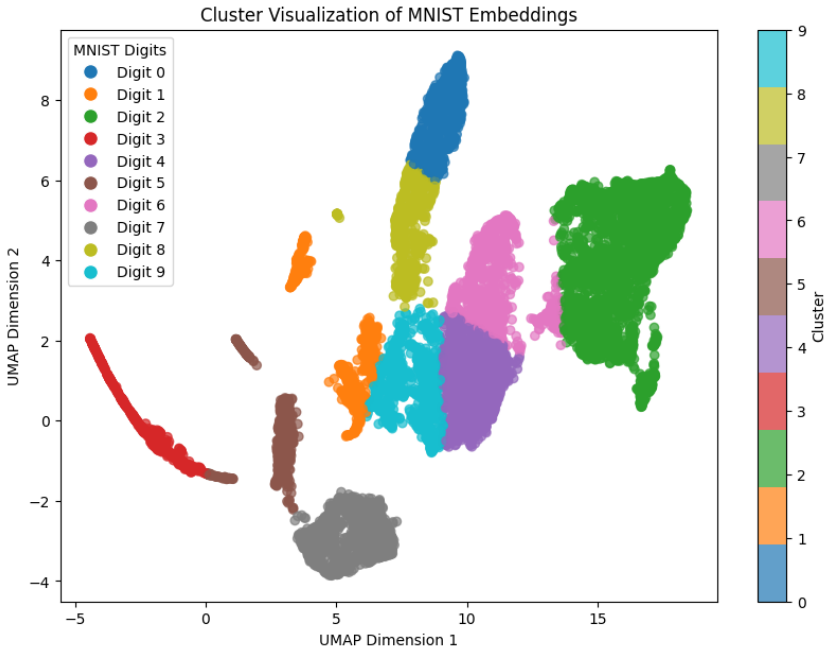


Figure 11: Clustering vizualization of MNIST embeddings (Autoencoder)

The plot shows well-separated clusters for different digit classes, which indicates that the autoencoder is learning meaningful features. Most of the clusters do not overlap significantly, suggesting that the latent space representations capture the inherent differences between digit classes. The digits with similar structures (e.g., 1 and 7) are relatively close, while more complex digits such as 8 and 3 have distinct areas. The UMAP dimensionality reduction appears to have preserved relationships between different digits well.

5 Levenshtein-based similarity approach

In the context of similarity learning, assessing the resemblance between sequences or feature representations is crucial for clustering, classification, and retrieval tasks. This section presents an implementation of a similarity learning approach based on the Levenshtein distance (mentioned in 2.2.3.2), which measures the minimum number of operations (insertion, deletion, substitution) required to transform one sequence into another. The method is applied to both textual sequences and feature representations extracted from images.

5.1 Implementation using a distance matrix

5.1.1 Data preparation

The dataset consists of **textual sequences**, representing biological or categorical data and **image feature representations**, extracted using the Histogram of Oriented Gradients (HOG) method from grayscale-converted images.
To integrate image-based data into the similarity computation, features are extracted using the HOG descriptor. The process involves:

1. Loading the image using skimage.io.
2. Converting the image to grayscale.
3. Extracting HOG features with a cell size of 16 x 16 pixels and a block size of 1 x 1

5.1.2 Custom implementation of the Levenshtein distance

A custom implementation of the **Levenshtein distance** is utilized to measure the dissimilarity between sequences. The function calculates an extended version of Levenshtein distance for numerical sequences by using Euclidean distance for numerical feature comparison and character-based operations for textual sequences.

5.1.3 Distance Matrix computation

To facilitate clustering, a pairwise distance matrix is computed for all combined data points (sequences and image features).
Calculating a full distance matrix for large datasets is computationally expensive and memory-intensive, posing significant challenges to scalability.
The computed distance matrix is used as input for the **DBSCAN** clustering algorithm, which identifies similar groups without requiring prior knowledge of the number of clusters.

5.1.4 Discussion and performance considerations

While the Levenshtein-based approach provides an accurate measure of similarity, its scalability is a key concern due to:

- **Quadratic time complexity:** Computing the distance matrix scales poorly with larger datasets.
- **Memory constraints:** Storing and processing large matrices can exhaust system resources.

Indeed, the first implementation we tested with simple arrays of characters [‘TGCB”, “TCBG”] and simple images did not work and took too much time to deliver the results. The model is not effective and need to be improved. We thought about modifying the model to replace the distance matrix with an approximate method such as Local Sensitive Hasting.

5.2 Proposed alternative using LSH

An alternative method to improve scalability involves using **Locality-Sensitive Hashing (LSH)**, which allows for approximate nearest neighbor searches by hashing similar data into the same "buckets" with high probability.

5.2.1 Concept of LSH

Locality-Sensitive Hashing (LSH) is an approximate nearest neighbor search technique designed to efficiently find similar items in high-dimensional spaces. Traditional methods for exact similarity search, such as brute-force comparisons, become computationally expensive as the dataset grows in size and dimensionality. LSH addresses this challenge by using hash functions that map similar data points to the same hash bucket with high probability while ensuring that dissimilar points are mapped to different buckets. Unlike traditional hashing methods that aim for uniform distribution, LSH is specifically designed to preserve the notion of similarity within the hash space. The algorithm works by creating multiple hash tables with different randomized hash functions to improve recall. When searching for similar items, instead of scanning the entire dataset, LSH significantly reduces the search space by focusing only on items in the same or neighboring hash buckets. This results in a trade-off between accuracy and efficiency, making LSH particularly useful in applications such as image retrieval, document clustering, and recommendation systems. By tuning the number of hash functions and tables, LSH can achieve a desirable balance between approximation accuracy and computational efficiency.

5.2.2 Implementation strategy

The clustering pipeline in our similarity learning approach is based on a hybrid methodology that integrates **Locality-Sensitive Hashing (LSH), Levenshtein**

distance, Histogram of Oriented Gradients (HOG) feature extraction, and clustering techniques such as DBSCAN and UMAP visualization. The objective is to efficiently cluster both DNA sequences and image data in an unsupervised manner while addressing scalability challenges.

5.2.2.1 Clustering DNA Sequences

The DNA clustering process begins with the generation of synthetic sequences representing biological data. To overcome the computational limitations of a full pairwise distance calculation, **MinHashLSH** is employed to approximate sequence similarity by hashing similar sequences into the same buckets with high probability. Each sequence undergoes MinHash signature computation, and an LSH index is built to efficiently retrieve potential matches.

To further refine the clustering, the Levenshtein distance is computed within the candidate groups identified by LSH. This is achieved using a parallelized implementation via multiprocessing, which significantly reduces the computational overhead. The resulting pairwise distance matrix is then normalized using MinMaxScaler to ensure non-negative values and appropriate scaling before applying the **DBSCAN clustering algorithm**, which identifies clusters based on the precomputed distance matrix without requiring prior knowledge of the number of clusters.

5.2.2.2 Clustering Image Data

For the image clustering task, feature representations are extracted from grayscale-converted images using the **Histogram of Oriented Gradients (HOG)** method. This feature extraction technique provides robust shape-based descriptors by analyzing gradient orientations within localized regions of the image. The feature extraction process involves:

1. Loading the image using skimage.io.
2. Converting the image to grayscale.
3. Extracting HOG features with a cell size of 16 x 16 pixels and a block size of 1 x 1.

Once the HOG features are extracted, they are normalized using MinMaxScaler to ensure consistency across feature vectors. The clustering is performed using **DBSCAN**, which groups similar images based on their feature representations. Subsequently, **UMAP (Uniform Manifold Approximation and Projection)** is applied to reduce the dimensionality of the high-dimensional feature space, enabling visualization of clusters in a 2D space.

5.2.3 Challenges and Optimization Considerations

Despite the successful integration of the aforementioned techniques, the clustering results exhibit suboptimal separation of data points, indicating areas for improvement. The current approach may require further tuning of critical parameters such as the eps (neighborhood radius) and min_samples (minimum cluster size) values in DBSCAN. Additionally, exploring alternative feature extraction techniques beyond HOG and experimenting with different similarity measures could enhance cluster quality.

Furthermore, applying advanced dimensionality reduction techniques, such as t-SNE or autoencoder-based embeddings, may provide better representations of the underlying data structure. Future work could also involve optimizing the LSH parameters to achieve a more balanced trade-off between efficiency and accuracy in similarity approximations.

In conclusion, while the implemented approach provides a foundational framework for similarity-based clustering, ongoing refinements and optimizations are necessary to improve the accuracy, scalability, and overall performance of the clustering pipeline.

For our test of the model, we used the same images as in the first test of the autoencoder (Golden retriever, German sherperds and tree images)

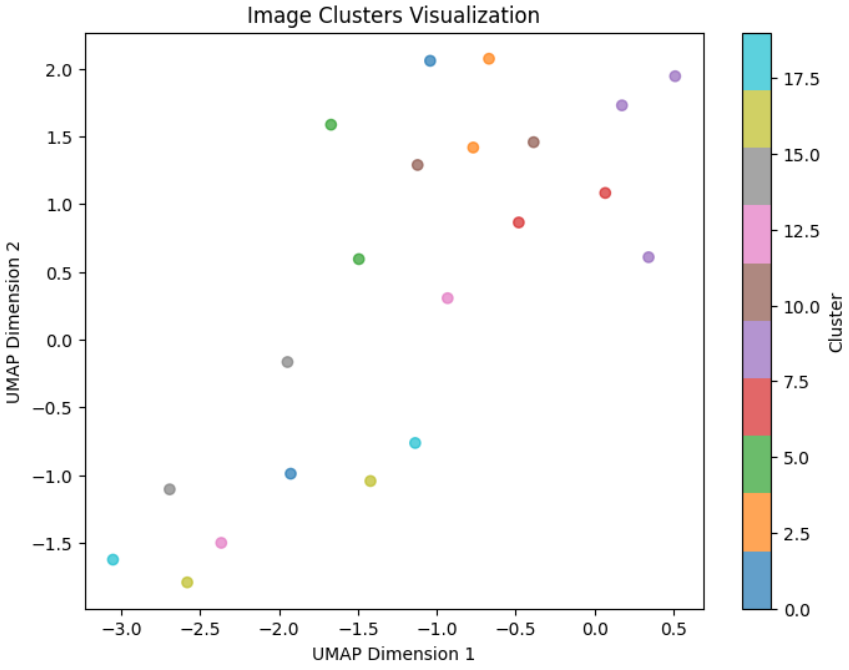


Figure 12: Levenshtein-based approach clustering

6 Conclusion

This research project has provided an in-depth exploration of similarity learning, encompassing both supervised and unsupervised approaches to measure and compare complex data objects effectively. We investigated various similarity definitions, ranging from feature-based and transformation-based approaches to semantic and relational perspectives, highlighting the critical role of appropriate similarity measures in different contexts. Our study demonstrated the effectiveness of supervised methods, particularly Siamese networks, in achieving high accuracy when sufficient labeled data is available. The combination of multiple distance metrics, such as cosine similarity and Euclidean distance, proved to be instrumental in enhancing performance, while techniques like dynamic margin adjustment and hard negative mining contributed to improved training efficiency and robustness.
However, one of the most significant challenges encountered in this research was the implementation of unsupervised similarity learning methods. Our attempts with autoencoders illustrated the complexities of extracting meaningful representations without explicit supervision. While our enhanced convolutional autoencoder yielded promising results in clustering and visualization, it required extensive fine-tuning and additional techniques, such as contrastive loss, to achieve satisfactory performance. The Levenshtein-based approach further underscored the inherent difficulties in handling large-scale datasets, where the computational cost of pairwise comparisons proved to be a limiting factor. Despite the integration of Locality-Sensitive Hashing (LSH) to mitigate scalability issues, the results highlighted the need for more efficient solutions capable of processing high-dimensional and diverse data efficiently.

These findings reinforce the critical need for further research into unsupervised methods that can scale effectively and adapt to the complexities of real-world applications. Future work should focus on optimizing clustering strategies, improving dimensionality reduction techniques, and developing hybrid models that integrate the strengths of both supervised and unsupervised learning. Additionally, exploring self-supervised and contrastive learning approaches could provide new avenues to leverage unlabeled data more effectively. Addressing these challenges will be crucial to unlocking the full potential of similarity learning, particularly in domains such as image retrieval, code similarity detection, and bioinformatics, where the demand for accurate and scalable similarity measures continues to grow.

7 References

1. *Tversky's feature-based similarity and beyond.* Likavec, Silvia, Lombardi, Ilaria et Cena, Federica. Torino : s.n., 2014.

2. *Similarity as transformation.* Hahn, Ulrike, Chater, Nick et Richardson, Lucy B. s.l. : Elsevier Science BV., 2003.

3. *Semantic Similarity- A Review of Approaches and Metrics.* D., Akila. s.l. : International Journal of Applied Engineering Research, 2018.

4. *A generalized model of relational similarity.* Kovacs, Balazs. s.l. : Social Networks, 2010, Vol. 32.

5. Han, Jiawei et Pei, Jian. Cosine Similarity. *Science Direct.* [En ligne] https://www.sciencedirect.com/topics/computer-science/cosine-similarity#:~:text=Cosine%20similarity%20measures%20the%20similarity,document%20similarity%20in%20text%20analysis..

6. Euclidean Distance. *Wikipedia.* [En ligne] https://en.wikipedia.org/wiki/Euclidean_distance.

7. Manhattan Distance. *Wikipedia.* [En ligne] https://fr.wikipedia.org/wiki/Distance_de_Manhattan.

8. *Jaccard Similarity Made Simple: A Beginner's Guide to Data Comparison.* Jadeja, Mayurdhvajsinh . s.l. : Medium.com, 2022.

9. Duarte, Felipe. Minkowski Distance. *ScienceDirect.* [En ligne] https://www.sciencedirect.com/topics/computer-science/minkowski-distance.

10. Yang, Shuyuan. Chebyshev Distance. *ScienceDirect.* [En ligne] https://www.sciencedirect.com/topics/computer-science/chebyshev-distance.

11. Leon, Andrew. Pearson Correlation Coefficient. *ScienceDirect.* [En ligne] https://www.sciencedirect.com/topics/economics-econometrics-and-finance/pearson-correlation-coefficient.

12. Hernandez, Mike. Spearman's Rank Correlation Coefficient. *ScienceDirect.* [En ligne] https://www.sciencedirect.com/topics/mathematics/spearmans-rank-correlation-coefficient.

13. Hamming Distance. *Wikipedia.* [En ligne] https://fr.wikipedia.org/wiki/Distance_de_Hamming.

14. *Understanding the Levenshtein Distance Equation for Beginners.* Nam, Ethan. s.l. : Medium.com, 2019.

15. *String correction using the Damerau-Levenshtein distance.* Zhao, Chunchun et Shani, Sartaj. s.l. : BMC Bioinformatics, 2019.

16. *Jaro winkler vs Levenshtein Distance.* Kulkarni, Srinivas. s.l. : Medium.com, 2021.

17. Similarity Learning. *Wikipedia.* [En ligne] https://en.wikipedia.org/wiki/Similarity_learning.

18. *Siamese Neural Networks for One-shot Image Recognition.* Koch, Gregory, Zemel, Richard et Salakhutdinov, Ruslan. s.l. : University of Toronto, 2015.

19. *Semi-Supervised Learning using Siamese Networks.* Sahito, Attaullah, Franck, Eibe et Pfahringer, Bernhard. s.l. : University of Waikato, 2021.

20. *Exploiting Unlabeled Data in CNNs by Self-Supervised Learning to Rank.* Liu, Xialei, van de Weijer, Joost et Bagdanov, Andrew. 8, s.l. : IEEE Xplore, 2019, Vol. 41.

21. *SiamALNet: A Semi-supervised Siamese Neural Network with Active Learning Approach for Auto-Labeling.* Supriyo, Roy, Aniket, Adsule et Kumar Sharma, Ashish. s.l. : Springer Nature Link, 2024.

22. *Similarity Search in High Dimensions via Hashing.* Gionis, Aristides, Indyk, Piotr et Motwani, Rajeev. s.l. : Stanford University, 1999.

23. *Deep graph similarity learning: a survey.* Willke, Theodore, et al. s.l. : Data mining and knowledge discovery, 2013, Vol. 27.

24. White, Scott et Smyth, Padhraic. *A Spectral Clustering Approach To Finding Communities in Graphs.* s.l. : Hillol Kargupta, University of Maryland Baltimore County, 2005. PR119.

25. *Understanding Graph Embedding Methods and Their Applications.* Xu, Mengjia. 4, s.l. : SIAM Review, 2021, Vol. 63.

26. *Kernel-driven similarity learning.* Kang, Zhao, Peng, Chong et Cheng, Qiang. s.l. : Neurocomputing, 2017, Vol. 267.

27. *Feature extraction using PCA and Kernel-PCA for face recognition.* Ebied, Hala M. s.l. : IEEE, 2012.

28. *Deep Unsupervised Similarity Learning Using Partially Ordered Sets.* Bautista, Miguel, Ommer, Bjorn et Sanakoyeu, Artsiom. s.l. : IEEE, 2017.

29. *G-SimCLR: Self-Supervised Contrastive Learning with Guided Projection via Pseudo Labelling.* Chakraborty, Souradip, Paul, Sayak et Gosthipaty, Aritra Roy. s.l. : IEEE, 2020.

30. *Large Scale Online Learning of Image Similarity Through Ranking.* Chechik, Gal, et al. s.l. : Journal of Machine Learning Research, 2010.

31. *Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree.* Wang, Wenhan, Li, Ge et Xin, Xia. s.l. : IEEE, 2020.

32. *Landscape Similarity Analysis Using Texture Encoded Deep-Learning Features on Unclassified Remote Sensing Imagery.* Robertson, Colin et Malik, Karim. s.l. : MDPI, 2021.

8 Appendices

1. Siamese Network code (trained and tested on Mnist dataset) (pages 44 to 54)
2. Autoencoder code (pages 55 to 60)
3. Levenshtein-based model code (pages 61 to 64)

Import libraries

```
import zipfile
import random
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import StandardScaler
from torch.utils.data import DataLoader, TensorDataset, Dataset
from sklearn.preprocessing import StandardScaler
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Read training data

```
def load_mnist(file_path):
    with zipfile.ZipFile(file_path, 'r') as z:
        z.extractall()
    # Charger les données
    data = pd.read_csv("mnist_train.csv")
    X = data.iloc[:, 1:].values  # Les pixels
    y = data.iloc[:, 0].values   # Les labels
    return X, y
```

Siamese Network

- Siamese Network: Encodes data in a reduced space to calculate similarity.
- Attention mechanism: Identifies relevant dimensions in embeddings to improve accuracy.
- Dynamic weighting: A linear layer learns to combine different types of distances. (Cosine, Euclidean and product)

```
# Define the Siamese Network
class SiameseNetwork(nn.Module):
    def __init__(self, input_dim=784, embedding_dim=64):
        super(SiameseNetwork, self).__init__()
        self.embedding = nn.Sequential(
            nn.Linear(input_dim, 1024),
            nn.ReLU(),
            nn.LayerNorm(1024),
            nn.Dropout(0.3),

            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.LayerNorm(512),
            nn.Dropout(0.3),

            nn.Linear(512, 128),
            nn.ReLU(),
            nn.LayerNorm(128),
            nn.Dropout(0.5),

            nn.Linear(128, embedding_dim),
            nn.ReLU()
        )


        # Attention mechanism
        self.attention = nn.Sequential(
            nn.Linear(embedding_dim, embedding_dim),
            nn.Tanh(),
            nn.Linear(embedding_dim, 1),
            nn.Softmax(dim=1)
        )
```

```
        # Fusion of distances with dynamic weights
        self.distance_fusion = nn.Sequential(
            nn.Linear(3, 1),  # Combines cosine, euclidean, and product
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.embedding(x)

    def similarity(self, embedding_a, embedding_b):
        # Appliquer l'attention
        # Add attention weight regularization
        attention_weights_a = torch.clamp(self.attention(embedding_a), min=0, max=1)
        attention_weights_b = torch.clamp(self.attention(embedding_b), min=0, max=1)


        # Pondérer les embeddings
        weighted_a = F.normalize(embedding_a * attention_weights_a, p=2, dim=1)
        weighted_b = F.normalize(embedding_b * attention_weights_b, p=2, dim=1)


        # Calculer les distances pondérées
        diff = torch.abs(weighted_a - weighted_b)
        cosine_sim = nn.functional.cosine_similarity(weighted_a, weighted_b)
        euclidean_dist = torch.norm(diff, dim=1)
        product = torch.sum(weighted_a * weighted_b, dim=1)

        # Combiner les distances et appliquer Sigmoid (plusieurs version, enlever commentaires de la version à tester)
        #version1: combinaison de cosine_sim, euclidean_dist et product
        distances = torch.stack([cosine_sim * 2, euclidean_dist, product], dim=1)
        #version2: cosine_sim uniquement
        #distances = torch.stack([cosine_sim, cosine_sim, cosine_sim], dim=1)
        #version3: euclidean_dist uniquement
        #distances = torch.stack([euclidean_dist, euclidean_dist, euclidean_dist], dim=1)
        fused_similarity = self.distance_fusion(distances).squeeze()
        return torch.sigmoid(fused_similarity)
```

Loss function

```
# Define Triplet Loss
class TripletLoss(nn.Module):
    def __init__(self):
        super(TripletLoss, self).__init__()

    def forward(self, anchor, positive, negative, margin):
        # Calculate positive and negative distances
        pos_dist = torch.norm(anchor - positive, dim=1)
        neg_dist = torch.norm(anchor - negative, dim=1)

        # Loss calculation with dynamic margin
        loss = torch.mean(torch.clamp(pos_dist - neg_dist + margin, min=0.0))
        return loss
```

Train and test functions for the Siamese Network

```
def train_siamese(model, train_loader, epochs=10, lr=0.001):
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-4)  # Weight decay for L2
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)  #Diminuer le taux d'apprentissage tous
    criterion = TripletLoss(margin=1.0)

    for epoch in range(epochs):
        model.train()
        epoch_loss = 0
        for pair_a, pair_b, labels in train_loader:
            pair_a, pair_b, labels = pair_a.float(), pair_b.float(), labels.float()

            optimizer.zero_grad()
            embed_a = model(pair_a)
            embed_b = model(pair_b)
            similarity_scores = model.similarity(embed_a, embed_b)
            loss = criterion(embed_anchor, embed_positive, embed_negative)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

        print(f"Epoch {epoch + 1}/{epochs}, Loss: {epoch_loss:.4f}")
```

```
def test_siamese(model, test_loader):
    model.eval()
    all_labels = []
    all_scores = []

    with torch.no_grad():
        for pair_a, pair_b, labels in test_loader:
            pair_a, pair_b, labels = pair_a.float(), pair_b.float(), labels.float()
            embed_a = model(pair_a)
            embed_b = model(pair_b)
            similarity_scores = model.similarity(embed_a, embed_b)
            all_labels.extend(labels.numpy())
            all_scores.extend(similarity_scores.numpy())

    auc = roc_auc_score(all_labels, all_scores)
    print(f"Test AUC: {auc:.4f}")
```

Prepare the data

```
def prepare_data(X, y):
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    pairs, labels = [], []
    for i in range(len(y)):
        for j in range(i+1, len(y)):
            pairs.append((X[i], X[j]))
            labels.append(1 if y[i] == y[j] else 0)
    pair_a = np.array([p[0] for p in pairs])
    pair_b = np.array([p[1] for p in pairs])
    return pair_a, pair_b, np.array(labels)
```

Training of the model

As the dataset is quite large, positive and negative pairs are dynamically generated during training, which keeps the memory under control.

```
def precompute_embeddings(dataset, model, device):
    model.eval()
    embeddings = []
    with torch.no_grad():
        for data in DataLoader(dataset, batch_size=128):  # Ajustez batch_size
            data = data.to(device).float()
            embeddings.append(model(data))
    return torch.cat(embeddings, dim=0)
```

```
# Dynamic generation of triplet during training

class TripletDataset(Dataset):
    def __init__(self, X, y, model, precomputed_embeddings=None, threshold=None):
        self.X = X
        self.y = y
        self.model = model
        self.precomputed_embeddings = precomputed_embeddings
        self.threshold = threshold  # Threshold pour le hard negative mining

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

    def __getitem__(self, idx):
        # Anchor
        anchor = torch.tensor(self.X[idx], dtype=torch.float32)

        # Positive samples
        positive_indices = np.where(self.y == self.y[idx])[0]
        if len(positive_indices) < 2:  # Ensure at least one valid positive
            raise IndexError("No valid positive samples found.")
        positive_idx = np.random.choice(positive_indices)
        positive = torch.tensor(self.X[positive_idx], dtype=torch.float32)

        # Negative candidates
        negative_candidates = np.where(self.y != self.y[idx])[0]
        if len(negative_candidates) < 1:  # Ensure at least one valid negative
            raise IndexError("No valid negative samples found.")
```

```
        # Hard Negative Mining (using precomputed embeddings)
        if self.precomputed_embeddings is not None:
            with torch.no_grad():
                anchor_embedding = self.precomputed_embeddings[idx]
                neg_embeddings = self.precomputed_embeddings[negative_candidates]
                distances = torch.norm(anchor_embedding - neg_embeddings, dim=1)

                # Filter negatives based on the threshold
                if self.threshold is not None:
                    hard_negatives = distances[distances < self.threshold + 0.2] #Increased treshold
                    if len(hard_negatives) == 0:
                        hard_negative_idx = negative_candidates[torch.argmin(distances)]
                    else:
                        hard_negative_idx = negative_candidates[torch.argmin(hard_negatives)]
                else:
                    hard_negative_idx = negative_candidates[torch.argmin(distances)]

                hard_negative = torch.tensor(self.X[hard_negative_idx], dtype=torch.float32)
        else:
            hard_negative_idx = np.random.choice(negative_candidates)
            hard_negative = torch.tensor(self.X[hard_negative_idx], dtype=torch.float32)

        return anchor, positive, hard_negative




def train_model(model, train_loader, epochs, lr):
    model.train()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = TripletLoss()  # No fixed margin here
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # Get the device

    # List to store loss for each epoch
    epoch_losses = []


    for epoch in range(epochs):
        epoch_loss = 0.0

        # Example: Adjust margin dynamically based on the epoch
        margin = 1.0 + (epoch / epochs) * 0.5  # Grow margin from 1.0 to 1.5

        for batch in train_loader:
            anchor, positive, negative = batch
            anchor = anchor.float().to(device) # Move to device
            positive = positive.float().to(device) # Move to device
            negative = negative.float().to(device) # Move to device

            optimizer.zero_grad()

            # Forward pass
            embed_anchor = model(anchor)
            embed_positive = model(positive)
            embed_negative = model(negative)

            # Compute triplet loss with dynamic margin
            loss = criterion(embed_anchor, embed_positive, embed_negative, margin)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        print(f"Epoch {epoch + 1}/{epochs}, Margin: {margin:.4f}, Loss: {epoch_loss:.4f}")

        # Append the epoch loss to the list
        epoch_losses.append(epoch_loss)

    # Plotting the loss
    plt.plot(range(1, epochs + 1), epoch_losses)
    plt.title('Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.show()
```

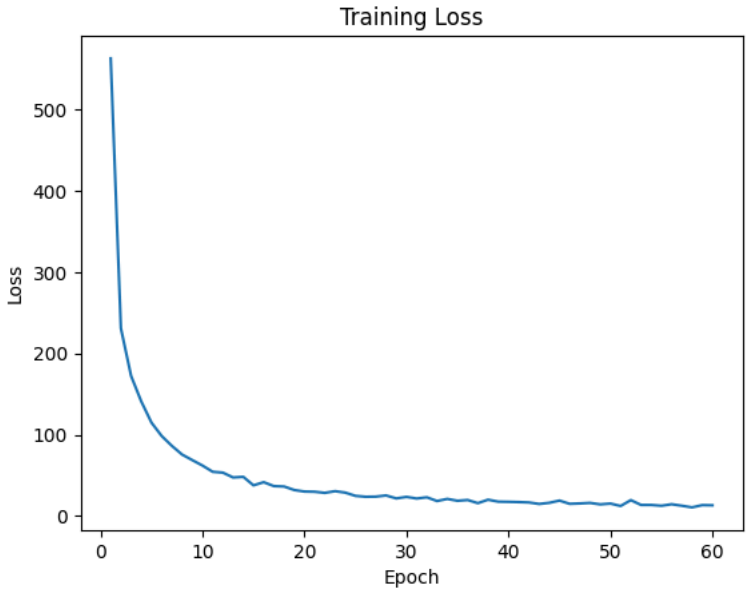
Test du modèle

```
def test_model(model, test_loader):
    model.eval()
```

```
    correct = 0
    total = 0

    all_scores_pos, all_scores_neg = [], []
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # Get device
    with torch.no_grad():
        for anchor, positive, negative in test_loader:
            anchor, positive, negative = anchor.float().to(device), positive.float().to(device), negative.float().to(device)

            # Compute embeddings
            embed_anchor = model(anchor)
            embed_positive = model(positive)
            embed_negative = model(negative)

            # Compute similarity scores
            pos_similarity = model.similarity(embed_anchor, embed_positive)
            neg_similarity = model.similarity(embed_anchor, embed_negative)

            # Accuracy computation
            correct += torch.sum(pos_similarity > neg_similarity).item()
            total += anchor.size(0)

            # Store scores for analysis
            all_scores_pos.extend(pos_similarity.cpu().numpy())
            all_scores_neg.extend(neg_similarity.cpu().numpy())

    # Compute accuracy
    accuracy = correct / total
    print(f"Test Accuracy: {accuracy:.4f}")

    # Optional: Further analysis (e.g., histograms of similarity scores)
    return all_scores_pos, all_scores_neg
```

```
# Main
if __name__ == "__main__":
    # Load and preprocess data
    file_path = "/content/drive/My Drive/Colab Notebooks/Projet 3A/4. New measure/mnist_train.zip"
    X, y = load_mnist(file_path)

    # Normalize and flatten data
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    X = X.reshape(X.shape[0], -1)  # Ensure shape is [batch_size, 784]

    # Initialize the Siamese Network
    embedding_net = SiameseNetwork(input_dim=784, embedding_dim=64)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    embedding_net.to(device)

    # Define a threshold for hard negative mining
    threshold = 0.5

    # Precompute embeddings for the dataset
    print("Precomputing embeddings for the dataset...")
    precomputed_embeddings = precompute_embeddings(X, embedding_net, device)

    # Save embeddings and pass them to the dataset
    print("Embedding computation done. Using precomputed embeddings.")

    # Prepare DataLoader with TripletDataset and precomputed embeddings
    train_dataset = TripletDataset(X, y, model=embedding_net, precomputed_embeddings=precomputed_embeddings, threshold=0.7)
    test_dataset = TripletDataset(X, y, model=embedding_net, precomputed_embeddings=precomputed_embeddings, threshold=0.5)

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, drop_last=True)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, drop_last=True)

    # Train the model
    train_model(embedding_net, train_loader, epochs=60, lr=0.0005)


    # Evaluate the model
    pos_scores, neg_scores = test_model(embedding_net, test_loader)
```

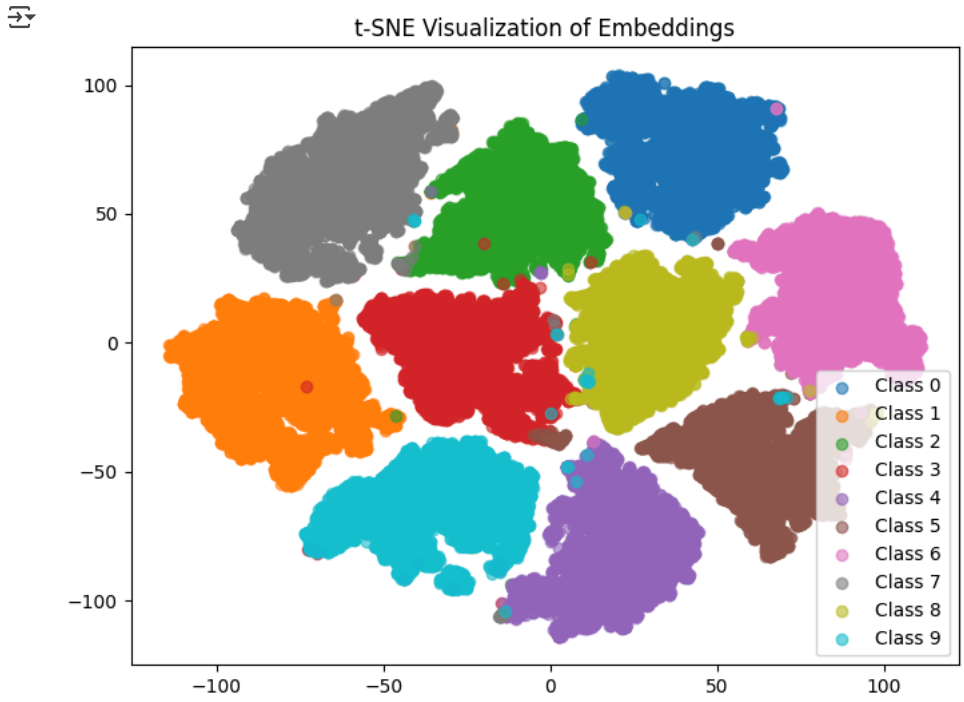
```
Precomputing embeddings for the dataset...
Embedding computation done. Using precomputed embeddings.
Epoch 1/60, Margin: 1.0000, Loss: 562.7404
Epoch 2/60, Margin: 1.0083, Loss: 230.8784
Epoch 3/60, Margin: 1.0167, Loss: 172.2546
Epoch 4/60, Margin: 1.0250, Loss: 140.8200
Epoch 5/60, Margin: 1.0333, Loss: 114.8203
Epoch 6/60, Margin: 1.0417, Loss: 98.3675
Epoch 7/60, Margin: 1.0500, Loss: 86.2447
Epoch 8/60, Margin: 1.0583, Loss: 75.5751
Epoch 9/60, Margin: 1.0667, Loss: 68.7388
Epoch 10/60, Margin: 1.0750, Loss: 62.0380
Epoch 11/60, Margin: 1.0833, Loss: 54.3832
Epoch 12/60, Margin: 1.0917, Loss: 53.3433
Epoch 13/60, Margin: 1.1000, Loss: 47.3926
Epoch 14/60, Margin: 1.1083, Loss: 48.1555
Epoch 15/60, Margin: 1.1167, Loss: 37.7000
Epoch 16/60, Margin: 1.1250, Loss: 41.5978
Epoch 17/60, Margin: 1.1333, Loss: 36.7690
Epoch 18/60, Margin: 1.1417, Loss: 36.2867
Epoch 19/60, Margin: 1.1500, Loss: 31.9516
Epoch 20/60, Margin: 1.1583, Loss: 30.0931
Epoch 21/60, Margin: 1.1667, Loss: 29.8250
Epoch 22/60, Margin: 1.1750, Loss: 28.4572
Epoch 23/60, Margin: 1.1833, Loss: 30.5304
Epoch 24/60, Margin: 1.1917, Loss: 28.7581
Epoch 25/60, Margin: 1.2000, Loss: 24.7677
Epoch 26/60, Margin: 1.2083, Loss: 23.6640
Epoch 27/60, Margin: 1.2167, Loss: 23.8295
Epoch 28/60, Margin: 1.2250, Loss: 25.2780
Epoch 29/60, Margin: 1.2333, Loss: 21.6277
Epoch 30/60, Margin: 1.2417, Loss: 23.5017
Epoch 31/60, Margin: 1.2500, Loss: 21.5087
Epoch 32/60, Margin: 1.2583, Loss: 22.9808
Epoch 33/60, Margin: 1.2667, Loss: 18.4103
Epoch 34/60, Margin: 1.2750, Loss: 21.0190
Epoch 35/60, Margin: 1.2833, Loss: 18.7391
Epoch 36/60, Margin: 1.2917, Loss: 19.6671
Epoch 37/60, Margin: 1.3000, Loss: 15.7961
Epoch 38/60, Margin: 1.3083, Loss: 20.0500
Epoch 39/60, Margin: 1.3167, Loss: 17.6480
Epoch 40/60, Margin: 1.3250, Loss: 17.4337
Epoch 41/60, Margin: 1.3333, Loss: 17.1242
Epoch 42/60, Margin: 1.3417, Loss: 16.6751
Epoch 43/60, Margin: 1.3500, Loss: 14.8316
Epoch 44/60, Margin: 1.3583, Loss: 16.2116
Epoch 45/60, Margin: 1.3667, Loss: 18.9877
Epoch 46/60, Margin: 1.3750, Loss: 14.9850
Epoch 47/60, Margin: 1.3833, Loss: 15.4408
Epoch 48/60, Margin: 1.3917, Loss: 16.1087
Epoch 49/60, Margin: 1.4000, Loss: 14.3140
Epoch 50/60, Margin: 1.4083, Loss: 15.2849
Epoch 51/60, Margin: 1.4167, Loss: 12.3291
Epoch 52/60, Margin: 1.4250, Loss: 19.5187
Epoch 53/60, Margin: 1.4333, Loss: 13.5983
Epoch 54/60, Margin: 1.4417, Loss: 13.5842
Epoch 55/60, Margin: 1.4500, Loss: 12.6074
Epoch 56/60, Margin: 1.4583, Loss: 14.4651
Epoch 57/60, Margin: 1.4667, Loss: 12.6285
Epoch 58/60, Margin: 1.4750, Loss: 10.6363
Epoch 59/60, Margin: 1.4833, Loss: 13.3846
Epoch 60/60, Margin: 1.4917, Loss: 13.1751
```



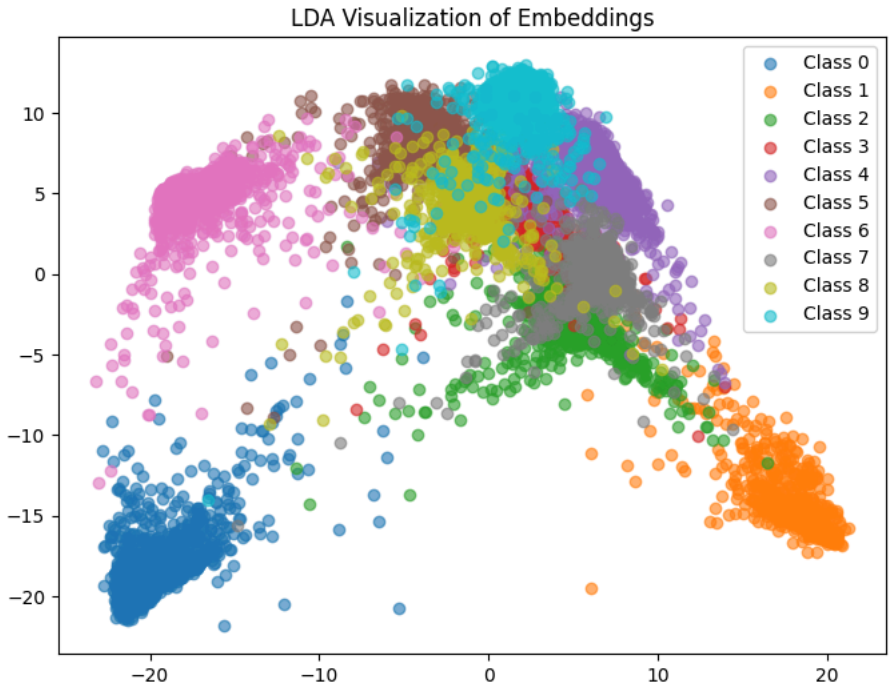

```
Test Accuracy: 0.9996
```

Visualization of Embeddings

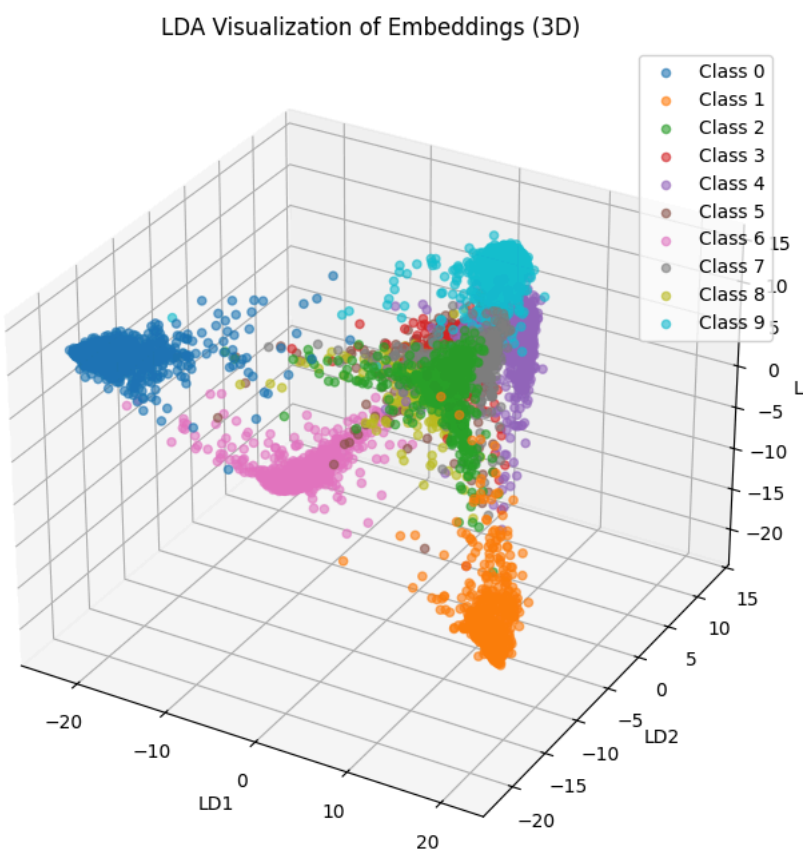
```
from sklearn.manifold import TSNE

#t-SNE visualization of embeddings

def plot_embeddings(model, X, y):
    model.eval()
    with torch.no_grad():
        embeddings = model(torch.tensor(X, dtype=torch.float32, device = "cuda"))
    embeddings_2d = TSNE(n_components=2).fit_transform(embeddings.cpu().numpy())
    #embeddings_2d = embeddings.cpu().numpy()

    plt.figure(figsize=(8, 6))
    for label in np.unique(y):
        indices = y == label
        plt.scatter(embeddings_2d[indices, 0], embeddings_2d[indices, 1], label=f"Class {label}", alpha=0.6)
    plt.title("t-SNE Visualization of Embeddings")
    plt.legend()
    plt.show()

# Call this function:
plot_embeddings(embedding_net, X, y)
```




```
#LDA visualization of embeddings

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

def plot_embeddings_lda(model, X, y):
    model.eval()
    with torch.no_grad():
        embeddings = model(torch.tensor(X, dtype=torch.float32, device = "cuda"))
    embeddings_2d = LinearDiscriminantAnalysis(n_components=2).fit_transform(embeddings.cpu().numpy(), y)

    plt.figure(figsize=(8, 6))
    for label in np.unique(y):
        indices = y == label
        plt.scatter(embeddings_2d[indices, 0], embeddings_2d[indices, 1], label=f"Class {label}", alpha=0.6)
    plt.title("LDA Visualization of Embeddings")
    plt.legend()
    plt.show()

# Call this function:
plot_embeddings_lda(embedding_net, X, y)
```

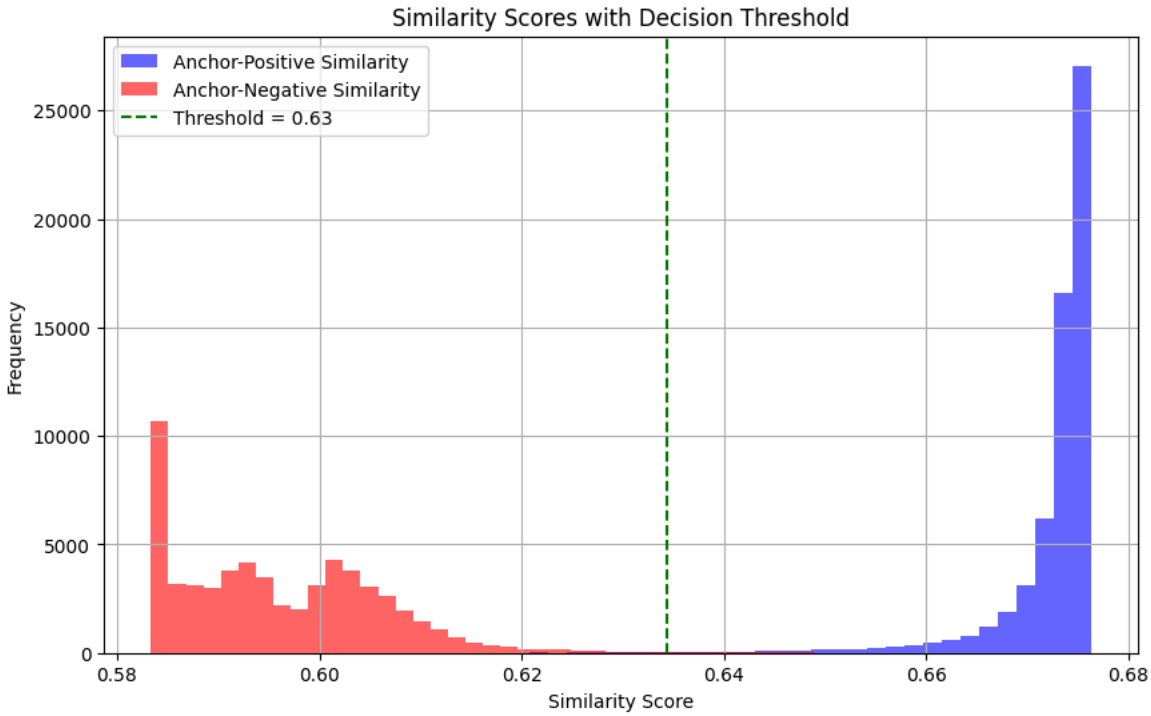



```
# LDA 3 dimensions

def plot_embeddings_lda3D(model, X, y):
    model.eval()
    with torch.no_grad():
        embeddings = model(torch.tensor(X, dtype=torch.float32, device="cuda"))
    embeddings_2d = LinearDiscriminantAnalysis(n_components=3).fit_transform(embeddings.cpu().numpy(), y)

    # 3D plotting
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    for label in np.unique(y):
        indices = y == label
        ax.scatter(embeddings_2d[indices, 0], embeddings_2d[indices, 1], embeddings_2d[indices, 2], label=f"Class {label}",

    ax.set_title("LDA Visualization of Embeddings (3D)")
    ax.set_xlabel("LD1")
    ax.set_ylabel("LD2")
    ax.set_zlabel("LD3")
    plt.legend()
    plt.show()

# Call this function:
plot_embeddings_lda3D(embedding_net, X, y)
```



Histogram Visualization with threshold

```
def analyze_threshold(pos_scores, neg_scores):
    all_scores = np.concatenate([pos_scores, neg_scores])
    threshold = np.mean(all_scores)  # Example threshold (mean of all scores)

    plt.figure(figsize=(10, 6))
    plt.hist(pos_scores, bins=50, alpha=0.6, label='Anchor-Positive Similarity', color='blue')
    plt.hist(neg_scores, bins=50, alpha=0.6, label='Anchor-Negative Similarity', color='red')

    plt.axvline(x=threshold, color='green', linestyle='--', label=f'Threshold = {threshold:.2f}')
    plt.title('Similarity Scores with Decision Threshold')
    plt.xlabel('Similarity Score')
    plt.ylabel('Frequency')
    plt.legend()
    plt.grid(True)
    plt.show()

# Call the function
analyze_threshold(pos_scores, neg_scores)
```



Density Plot

```
def plot_density(pos_scores, neg_scores):
    plt.figure(figsize=(10, 6))
    sns.kdeplot(pos_scores, label='Anchor-Positive Similarity', shade=True)
    sns.kdeplot(neg_scores, label='Anchor-Negative Similarity', shade=True)
    plt.title('Density Plot of Similarity Scores')
    plt.xlabel('Similarity Score')
    plt.ylabel('Density')
    plt.legend()
    plt.show()

# Call the function
plot_density(pos_scores, neg_scores)
```

```
<ipython-input-41-59a125c1ac9a>:3: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(pos_scores, label='Anchor-Positive Similarity', shade=True)
```

Test of the model on random images in mnist_test.csv

```
#Take two random images from mnist_test.csv to test the trained model
X_test = pd.read_csv('/content/drive/My Drive/Colab Notebooks/Projet 3A/4. New measure/mnist_test.csv')
X_test = X_test.values

# Get two random indices
random_indices = random.sample(range(len(X_test)), 2)

# Get the images at those indices, EXCLUDING the first column (likely the label)
image1 = X_test[random_indices[0], 1:]  # Select columns from index 1 onwards
image2 = X_test[random_indices[1], 1:]  # Select columns from index 1 onwards

# Reshape images to 28x28 for display
image1_display = image1.reshape(28, 28)
image2_display = image2.reshape(28, 28)

# Display images using matplotlib
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.imshow(image1_display, cmap='gray')
plt.title(f"Random Image 1 (Index: {random_indices[0]})")

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

Enhanced autoencoder model for similarity

```
!pip install umap-learn
from IPython.display import display
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms
from sklearn.mixture import GaussianMixture
from torch.utils.data import DataLoader, Dataset
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from umap import UMAP
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Montage du drive
from google.colab import drive
drive.mount('/content/drive')
```

Afficher la sortie masquée

+ Code + Texte

1. Image preprocessing

```
# Prétraitement des images
def preprocess_image(image_path):
    transform = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
    ])
    image = Image.open(image_path).convert('RGB')
    return transform(image)

# Dataset non supervisé
class UnsupervisedDataset(Dataset):
    def __init__(self, data):
        self.data = data

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

    def __getitem__(self, idx):
        item = self.data[idx]
        image = preprocess_image(item['image_path'])
        return image
```

2. Enhanced autoencoder definition

```
# Autoencodeur amélioré
class EnhancedConvAutoencoder(nn.Module):
    def __init__(self):
        super(EnhancedConvAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 256, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(256),
            nn.Conv2d(256, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.Conv2d(128, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.Flatten()
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 128, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
```

```
            nn.BatchNorm2d(128),
            nn.ConvTranspose2d(128, 256, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(256),
            nn.ConvTranspose2d(256, 3, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        batch_size = x.size(0)
        # The output of the encoder has 64 channels, not 256
        spatial_dim = int(np.sqrt(x.numel() / (batch_size * 64)))
        # Reshape with 64 channels
        x = x.view(batch_size, 64, spatial_dim, spatial_dim)
        x = self.decoder(x)
        return x
```

3. Training loss function

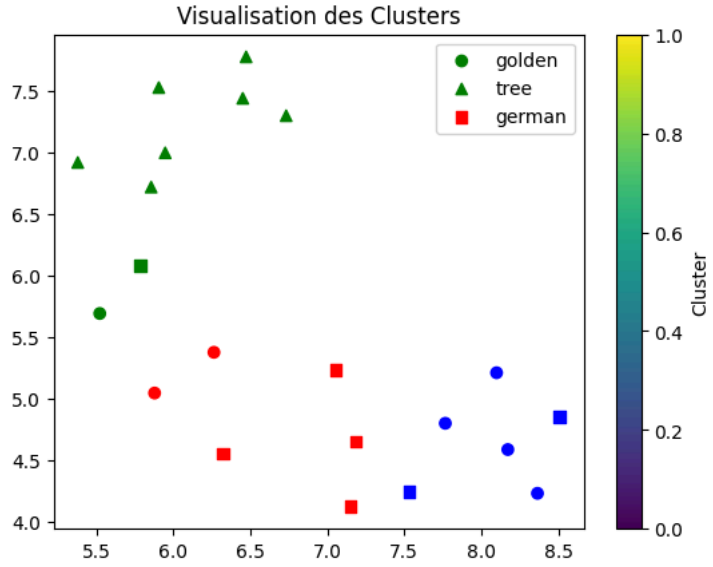
```
# Fonction d'entraînement avec perte contrastive
def train_autoencoder_with_contrastive_loss(model, dataloader, optimizer, device, epochs=50):
    model.to(device)
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for images in dataloader:
            images = images.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            embeddings = model.encoder(images).view(images.size(0), -1)  # Embeddings
            # Perte de reconstruction
            mse_loss = nn.MSELoss()(outputs, images)
            # Perte contrastive
            distances = torch.cdist(embeddings, embeddings, p=2)
            similar_loss = torch.mean(distances[distances < 1.0] ** 2)
            dissimilar_loss = torch.mean((1.0 - distances[distances >= 1.0]).clamp(min=0) ** 2)
            contrastive_loss = similar_loss + dissimilar_loss
            loss = mse_loss + contrastive_loss
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(dataloader):.4f}")
```

4. Embeddings

```
# Récupération des embeddings
def get_embeddings(data_loader, model, device):
    embeddings = []
    model.to(device)
    model.eval()
    with torch.no_grad():
        for images in data_loader:
            images = images.to(device)
            embedding = model.encoder(images).view(images.size(0), -1).cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Visualisation des embeddings
def visualize_embeddings(embeddings, cluster_labels, image_paths):
    # Define markers for different image groups
    group_markers = {
        "golden": "o",  # Circle for golden retrievers
        "german": "s",  # Square for german sheperds
        "tree": "^"  # Triangle for trees
    }
    # Define colors for different clusters
    cluster_colors = ['r', 'g', 'b']  # Red, Green, Blue for clusters 0, 1, 2

    # Extract group labels from image paths
    group_labels = [
        "golden" if "golden" in path else "german" if "german" in path else "tree"
        for path in image_paths
    ]

    # Create a scatter plot with colors and markers
    fig, ax = plt.subplots()
```

```
    # Create separate scatter plots for each group with labels
    for group in set(group_labels):
        indices = [i for i, label in enumerate(group_labels) if label == group]
        scatter = ax.scatter(
            embeddings[indices, 0],
            embeddings[indices, 1],
            c=[cluster_colors[cluster_labels[i]] for i in indices],
            marker=group_markers[group],
            label=group
        )

    # Add legend for image groups
    ax.legend()
    plt.colorbar(scatter, label="Cluster")  # Pass the scatter plot object to colorbar
    plt.title("Visualisation des Clusters")
    plt.show()

# Données
example_data = [
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden2.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden3.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden4.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden5.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden6.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden7.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german1.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german2.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german3.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german4.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german5.png"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german6.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german7.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree2.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree3.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree4.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree5.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree6.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree7.webp"}
]

unsupervised_dataset = UnsupervisedDataset(example_data)
dataloader = DataLoader(unsupervised_dataset, batch_size=4, shuffle=True)

# Configuration du modèle
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
autoencoder = EnhancedConvAutoencoder()
optimizer = optim.Adam(autoencoder.parameters(), lr=1e-4)

# Entraînement
train_autoencoder_with_contrastive_loss(autoencoder, dataloader, optimizer, device, epochs=25)

# Extraction des embeddings
data_loader = DataLoader(unsupervised_dataset, batch_size=1, shuffle=False)
embeddings = get_embeddings(data_loader, autoencoder, device)

# Réduction avec UMAP
scaler = StandardScaler()
embeddings = scaler.fit_transform(embeddings)
umap = UMAP(n_components=2, n_neighbors=10, min_dist=0.0, metric='cosine', random_state=42)
reduced_embeddings = umap.fit_transform(embeddings)

# Clustering avec KMeans
n_clusters = 3
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(reduced_embeddings)

# Visualisation des clusters KMeans
visualize_embeddings(reduced_embeddings, cluster_labels, [d['image_path'] for d in example_data])
```

```
/usr/local/lib/python3.11/dist-packages/PIL/Image.py:1045: UserWarning: Palette images with Transparency expressed in by
  warnings.warn(
Epoch [1/25], Loss: nan
Epoch [2/25], Loss: nan
Epoch [3/25], Loss: nan
Epoch [4/25], Loss: nan
Epoch [5/25], Loss: nan
Epoch [6/25], Loss: nan
Epoch [7/25], Loss: nan
Epoch [8/25], Loss: nan
Epoch [9/25], Loss: nan
Epoch [10/25], Loss: nan
Epoch [11/25], Loss: nan
Epoch [12/25], Loss: nan
Epoch [13/25], Loss: nan
Epoch [14/25], Loss: nan
Epoch [15/25], Loss: nan
Epoch [16/25], Loss: nan
Epoch [17/25], Loss: nan
Epoch [18/25], Loss: nan
Epoch [19/25], Loss: nan
Epoch [20/25], Loss: nan
Epoch [21/25], Loss: nan
Epoch [22/25], Loss: nan
Epoch [23/25], Loss: nan
Epoch [24/25], Loss: nan
Epoch [25/25], Loss: nan
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was renamed
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/umap/umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting rando
  warn(
```



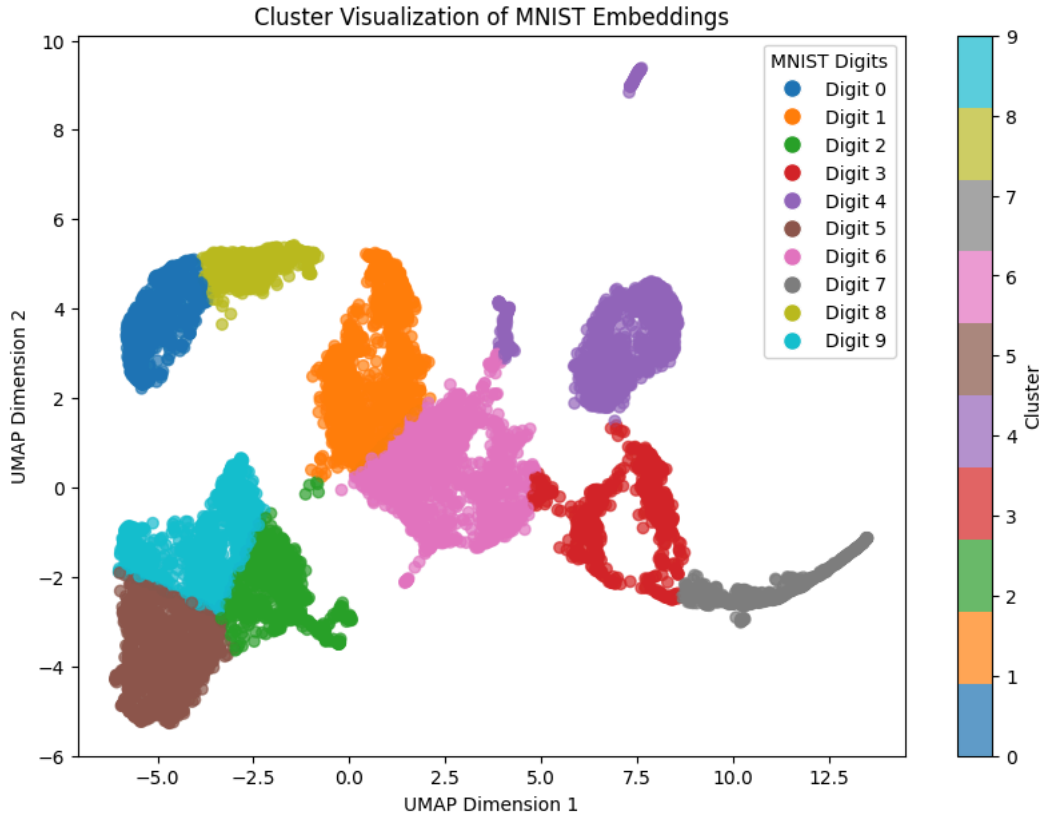
Test of the model on the mnist dataset

```
# Load MNIST dataset (without using labels in training)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

mnist_train = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
mnist_test = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

# Create DataLoader without using labels
train_loader = DataLoader(mnist_train, batch_size=64, shuffle=True)
test_loader = DataLoader(mnist_test, batch_size=64, shuffle=False)

# Autoencoder for unsupervised learning
class MNISTAutoencoder(nn.Module):
    def __init__(self):
        super(MNISTAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Flatten()
        )
        self.decoder = nn.Sequential(
```

```
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        batch_size = x.size(0)
        encoded = encoded.view(batch_size, 64, 7, 7)  # Reshape to match decoder input
        decoded = self.decoder(encoded)
        return decoded

# Training function (unsupervised, no labels used)
def train_autoencoder(model, dataloader, optimizer, device, epochs=4):
    model.to(device)
    criterion = nn.MSELoss()
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for images, _ in dataloader:  # Ignore labels
            images = images.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, images)  # Unsupervised: compare reconstruction only
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        print(f"Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(dataloader):.4f}")

# Function to extract embeddings
def get_embeddings(data_loader, model, device):
    embeddings, labels = [], []
    model.to(device)
    model.eval()
    with torch.no_grad():
        for images, targets in data_loader:  # Targets not used in unsupervised learning
            images = images.to(device)
            encoded = model.encoder(images).view(images.size(0), -1).cpu().numpy()
            embeddings.append(encoded)
            labels.extend(targets.numpy())  # Collect labels only for visualization
    return np.vstack(embeddings), np.array(labels)

# Visualization function for clusters
def visualize_clusters(embeddings, cluster_labels, labels):
    plt.figure(figsize=(10, 7))
    scatter = plt.scatter(embeddings[:, 0], embeddings[:, 1], c=cluster_labels, cmap='tab10', alpha=0.7)
    plt.colorbar(scatter, label="Cluster")
    plt.title("Cluster Visualization of MNIST Embeddings")
    plt.xlabel("UMAP Dimension 1")
    plt.ylabel("UMAP Dimension 2")

    # Add legend to indicate the actual digit classes (for visualization only)
    legend_labels = {i: f"Digit {i}" for i in range(10)}
    handles = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor=plt.cm.tab10(i/10), markersize=10) for i in range(
    plt.legend(handles, legend_labels.values(), title="MNIST Digits")
    plt.show()

# Train the autoencoder (unsupervised)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
autoencoder = MNISTAutoencoder()
optimizer = optim.Adam(autoencoder.parameters(), lr=0.001)

print("Training the Autoencoder in an Unsupervised Manner...")
train_autoencoder(autoencoder, train_loader, optimizer, device)

# Extract embeddings
print("Extracting Embeddings...")
embeddings, labels = get_embeddings(test_loader, autoencoder, device)

# Dimensionality reduction using UMAP
scaler = StandardScaler()
embeddings_scaled = scaler.fit_transform(embeddings)
umap = UMAP(n_components=2, n_neighbors=15, min_dist=0.1, metric='euclidean', random_state=42)
reduced_embeddings = umap.fit_transform(embeddings_scaled)

# Perform clustering with KMeans (10 clusters for digits 0-9)
n_clusters = 10
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(reduced_embeddings)
```

```
# Visualize clusters
print("Visualizing Clusters...")
visualize_clusters(reduced_embeddings, cluster_labels, labels)
```

```
Training the Autoencoder in an Unsupervised Manner...
Epoch [1/4], Loss: 0.8748
Epoch [2/4], Loss: 0.8454
Epoch [3/4], Loss: 0.8449
Epoch [4/4], Loss: 0.8447
Extracting Embeddings...
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was renamed
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/umap/umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting rando
  warn(
Visualizing Clusters...
```



1. First version with a combined distance matrix

Model not effective enough, isn't able to find a solution because it costs too much

```
!pip install python-Levenshtein
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.metrics import pairwise_distances
import Levenshtein
from skimage.feature import hog
from skimage.color import rgb2gray
from skimage import io

# Montage du drive
from google.colab import drive
drive.mount('/content/drive')

# Exemple de données (chaînes + images)
sequences = ["ACTG", "ACGT", "TGCA", "AGTG", "ACGG"]
image_paths = [
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden2.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden3.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden4.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden5.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden6.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden7.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german1.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german2.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german3.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german4.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german5.png"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german6.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german7.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree2.jpeg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree3.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree4.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree5.jpg"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree6.webp"},
    {"image_path": "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree7.webp"}
]

# Fonction pour extraire descripteurs HOG d'une image
def image_to_sequence(image_path):
    image = io.imread(image_path)
    gray_image = rgb2gray(image)
    features, _ = hog(
        gray_image, pixels_per_cell=(16, 16), cells_per_block=(1, 1), visualize=True
    )
    return features  # Retourne une séquence numérique

# Créer une représentation mixte (séquences + images)
image_sequences = [image_to_sequence(path['image_path']) for path in image_paths]

# Calcul de la distance de Levenshtein étendue
def extended_levenshtein(seq1, seq2):
    n, m = len(seq1), len(seq2)
    dp = np.zeros((n + 1, m + 1))

    for i in range(n + 1):
        for j in range(m + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            else:
                cost = (
                    np.linalg.norm(seq1[i - 1] - seq2[j - 1])
                    if isinstance(seq1[0], np.ndarray)
                    else int(seq1[i - 1] != seq2[j - 1])
                )
                dp[i][j] = min(
                    dp[i - 1][j] + 1,     # Suppression
                    dp[i][j - 1] + 1,     # Insertion
                    dp[i - 1][j - 1] + cost,  # Substitution
                )
    return dp[n][m]
```

```
# Créer une matrice de distance combinée
#Calculer une matrice de distance pour de grands ensembles peut être coûteux.
#Solution : Approximation avec des méthodes comme locality-sensitive hashing (LSH)

all_data = sequences + image_sequences
distance_matrix = np.zeros((len(all_data), len(all_data)))

for i in range(len(all_data)):
    for j in range(i + 1, len(all_data)):
        dist = extended_levenshtein(all_data[i], all_data[j])
        distance_matrix[i][j] = dist
        distance_matrix[j][i] = dist

# Clustering avec DBSCAN
clustering = DBSCAN(eps=5, min_samples=2, metric="precomputed")
clusters = clustering.fit_predict(distance_matrix)

# Résultats
print("Matrice de distances :\n", distance_matrix)
print("Clusters attribués :", clusters)
```

Afficher la sortie masquée

2. Second version with LSH (approximation)

```
!pip install python-Levenshtein datasketch scikit-image umap-learn

import numpy as np
import random
import Levenshtein
from datasketch import MinHash, MinHashLSH
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import MinMaxScaler
from multiprocessing import Pool
from skimage.feature import hog
from skimage.color import rgb2gray
from skimage.transform import resize
from skimage import io
import matplotlib.pyplot as plt
from umap import UMAP

# ------------------------- DNA SEQUENCE PROCESSING -------------------------

# Generate synthetic DNA sequences
def generate_dna_sequences(num_sequences, length=100):
    bases = ['A', 'C', 'G', 'T']
    return [''.join(random.choices(bases, k=length)) for _ in range(num_sequences)]

# Use a smaller subset for testing
dna_sequences = generate_dna_sequences(100, length=100)

# Function to create MinHash signature for LSH
def get_minhash(sequence, num_perm=128):
    minhash = MinHash(num_perm=num_perm)
    for char in sequence:
        minhash.update(char.encode('utf8'))
    return minhash

# Create an LSH index for DNA sequences
lsh_dna = MinHashLSH(threshold=0.8, num_perm=128)
minhashes_dna = {}

for idx, seq in enumerate(dna_sequences):
    minhash = get_minhash(seq)
    lsh_dna.insert(f"dna_{idx}", minhash)
    minhashes_dna[f"dna_{idx}"] = minhash

# Find LSH candidate matches
candidates = {}
for idx in range(len(dna_sequences)):
    key = f"dna_{idx}"
    matches = lsh_dna.query(minhashes_dna[key])
    candidates[key] = matches

# Parallelized Levenshtein Distance Calculation
def compute_distance(pair):
    i, j = pair
    return i, j, Levenshtein.distance(dna_sequences[i], dna_sequences[j])
```

```
pairs = []
for i in range(len(dna_sequences)):
    key_i = f"dna_{i}"
    for key_j in candidates[key_i]:
        j = int(key_j.split('_')[1])
        if i != j:
            pairs.append((i, j))

# Initialize distance matrix with zeros
distance_matrix = np.zeros((len(dna_sequences), len(dna_sequences)))

# Use multiprocessing to compute distances in parallel
with Pool(processes=4) as pool:  # Adjust for system cores
    results = pool.map(compute_distance, pairs)

# Fill the distance matrix
for i, j, dist in results:
    distance_matrix[i][j] = dist
    distance_matrix[j][i] = dist

# Normalize the distance matrix to ensure non-negative values for DBSCAN
distance_matrix = np.clip(distance_matrix, 0, None)
scaler = MinMaxScaler()
distance_matrix = scaler.fit_transform(distance_matrix)

# Run DBSCAN clustering with adjusted parameters
clustering_dna = DBSCAN(eps=0.7, min_samples=1, metric="precomputed")
dna_clusters = clustering_dna.fit_predict(distance_matrix)

print("DNA Clusters:", np.unique(dna_clusters))

# ------------------------- IMAGE PROCESSING -------------------------

image_paths = [
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden.jpeg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden2.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden3.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden4.jpeg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden5.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden6.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/golden7.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german1.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german2.jpeg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german3.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german4.jpeg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german5.png",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german6.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/german7.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree2.jpeg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree3.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree4.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree5.jpg",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree6.webp",
    "/content/drive/My Drive/Colab Notebooks/Projet 3A/6. Unsupervised method/image/tree7.webp"
]

# Function to extract HOG descriptors from images with resizing
def extract_hog_features(image_path, target_size=(128, 128)):
    image = io.imread(image_path)
    image_resized = resize(image, target_size, anti_aliasing=True)
    gray_image = rgb2gray(image_resized)
    features, _ = hog(gray_image, pixels_per_cell=(16, 16), cells_per_block=(1, 1), visualize=True)
    return features

# Extract and normalize image features
image_features = np.array([extract_hog_features(img_path) for img_path in image_paths])
scaled_features = np.array(image_features)

# Normalize image features
scaler = MinMaxScaler()
scaled_features = scaler.fit_transform(scaled_features)

# Clustering image features with adjusted parameters
clustering_images = DBSCAN(eps=0.7, min_samples=1)
image_clusters = clustering_images.fit_predict(scaled_features)

print("Image Clusters:", np.unique(image_clusters))

# Adjust neighbors based on available data
n_neighbors = max(2, min(3, len(scaled_features) - 1))
```

```
# Ensure data is dense before passing to UMAP
scaled_features = np.asarray(scaled_features)

# UMAP Dimensionality Reduction with adjusted parameters
umap = UMAP(n_components=2, n_neighbors=n_neighbors, init="random", random_state=42)
reduced_features = umap.fit_transform(scaled_features)

# Plotting the results
plt.figure(figsize=(8, 6))
scatter = plt.scatter(reduced_features[:, 0], reduced_features[:, 1], c=image_clusters, cmap='tab10', alpha=0.7)
plt.colorbar(label="Cluster")
plt.title("Image Clusters Visualization")
plt.xlabel("UMAP Dimension 1")
plt.ylabel("UMAP Dimension 2")
plt.show()
```

```
Requirement already satisfied: python-Levenshtein in /usr/local/lib/python3.11/dist-packages (0.26.1)
Requirement already satisfied: datasketch in /usr/local/lib/python3.11/dist-packages (1.6.5)
Requirement already satisfied: scikit-image in /usr/local/lib/python3.11/dist-packages (0.25.0)
Requirement already satisfied: umap-learn in /usr/local/lib/python3.11/dist-packages (0.5.7)
Requirement already satisfied: Levenshtein==0.26.1 in /usr/local/lib/python3.11/dist-packages (from python-Levenshtein)
Requirement already satisfied: rapidfuzz<4.0.0,>=3.9.0 in /usr/local/lib/python3.11/dist-packages (from Levenshtein==0.2
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.11/dist-packages (from datasketch) (1.26.4)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from datasketch) (1.13.1)
Requirement already satisfied: networkx>=3.0 in /usr/local/lib/python3.11/dist-packages (from scikit-image) (3.4.2)
Requirement already satisfied: pillow>=10.1 in /usr/local/lib/python3.11/dist-packages (from scikit-image) (11.1.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in /usr/local/lib/python3.11/dist-packages (from scikit-image) (2.
Requirement already satisfied: tifffile>=2022.8.12 in /usr/local/lib/python3.11/dist-packages (from scikit-image) (2025.
Requirement already satisfied: packaging>=21 in /usr/local/lib/python3.11/dist-packages (from scikit-image) (24.2)
Requirement already satisfied: lazy-loader>=0.4 in /usr/local/lib/python3.11/dist-packages (from scikit-image) (0.4)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (1.6.1)
Requirement already satisfied: numba>=0.51.2 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (0.60.0)
Requirement already satisfied: pynndescent>=0.5 in /usr/local/lib/python3.11/dist-packages (from umap-learn) (0.5.13)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from umap-learn) (4.67.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.11/dist-packages (from numba>=0.51.2
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.11/dist-packages (from pynndescent>=0.5->umap-lear
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.22-
DNA Clusters: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
/usr/local/lib/python3.11/dist-packages/PIL/Image.py:1045: UserWarning: Palette images with Transparency expressed in by
  warnings.warn(
Image Clusters: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was renamed
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/umap/umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting rando
  warn(
```