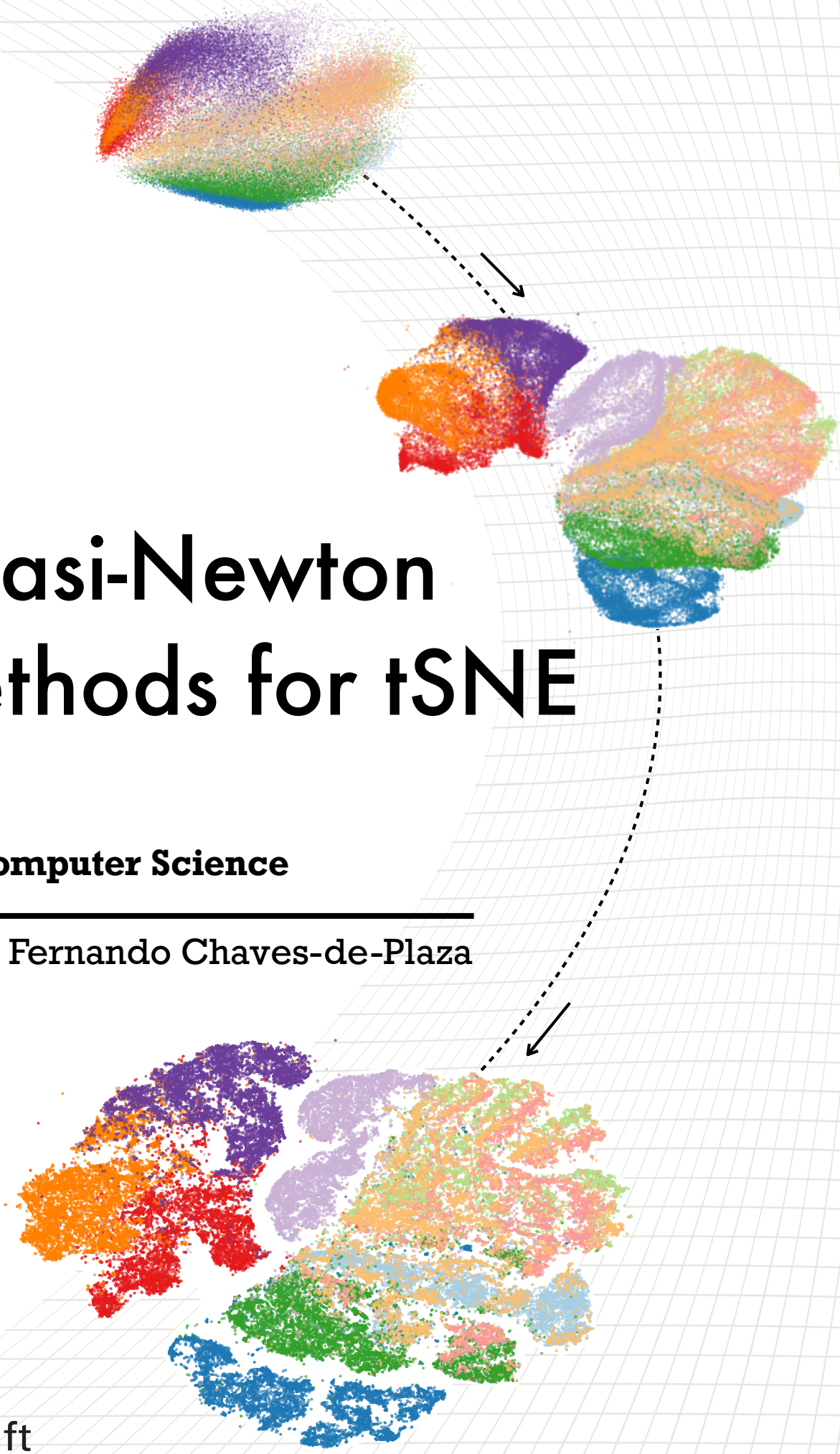

Quasi-Newton Methods for tSNE

MSc Computer Science

Nicolas Fernando Chaves-de-Plaza

TU Delft

Quasi-Newton Methods for tSNE

by

N.F. Chaves-de-Plaza

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday August 31, 2020.

Student number:	4795512	
Thesis committee:	Prof. Dr. A. Vilanova,	TU Delft
	Dr. K. Hildebrandt,	TU Delft, supervisor
	Dr. J. Wu,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Acknowledgments

This work would have not been possible without the support circle that I am fortunate to have. I am truly grateful to Dr. Klaus Hildebrandt for his mentoring throughout the project. Not only did he teach me a great deal about numerical optimization, but also about how to carry out research independently and how to communicate my ideas to an audience. I would also like to thank Prof. Dr. Anna Vilanova for her unconditional support and timely feedback, which helped me to significantly improve this work. On numerous occasions, I felt confused or pessimistic about my project. I owe a lot to Mario Coutiño, who devoted much time to discuss and clarify my ideas. This discussion proved fundamental to be able to shape my thesis. Finally, I am forever grateful to Lina for motivating me during the toughest phases of this project and for the countless hours that she spent helping me proofread and polish previous drafts of this document.

Above all, I would like to thank my parents, without whom I would not have been able to come to the Netherlands to pursue my passion for technology and to continue developing my education. I am forever indebted to them.

Contents

1 Introduction **1**

2 Related Work **5**
2.1 Dimensionality Reduction for Visualizing High-Dimensional Data . . . 5
2.2 Graph-based Nonlinear Embedding Techniques . . . 6
2.2.1 Modeling of the Problem . . . 7
2.2.2 Approximate uNN Graph Construction . . . 8
2.2.3 Iterative Optimization . . . 8
2.2.4 Approximation of the Gradient . . . 10
2.2.5 Hardware Accelerations . . . 11
2.3 Tools and Systems . . . 12

3 Background **13**
3.1 Quasi-Newton Optimization Frameworks . . . 13
3.1.1 Line Search . . . 14
3.1.2 Trust Region . . . 15
3.2 The Kullback-Leibler Divergence and its Derivatives . . . 16
3.2.1 The Affinity Matrices . . . 16
3.2.2 The Graph Laplacian . . . 17
3.2.3 The Gradient . . . 17
3.2.4 The Hessian . . . 19

4 Methodology and Implementation **21**
4.1 Experimental Pipeline . . . 21
4.1.1 Initialization of the Embedding . . . 21
4.1.2 Computation of the High-Dimensional Matrix of Affinities . . . 22
4.1.3 Computation of the Gradient and Cost Function Value . . . 22
4.1.4 Computation and Updating of the Hessian Approximation . . . 22
4.1.5 Finding the Next Iterate . . . 23
4.1.6 Optimization Loop Stopping Criteria and Logging . . . 24
4.2 Datasets . . . 25

5 Comparative Study of Techniques for Optimizing tSNE Cost Function **27**
5.1 Parameter Choice in Quasi-Newton Methods . . . 27
5.2 The Best Hessian Approximation . . . 29
5.2.1 Adding Low-Dimensional Information to the Spectral Direction . . . 30
5.2.2 Comparison of Hessian Approximations for tSNE . . . 31
5.3 Comparative Study of Optimization Techniques . . . 32
5.3.1 The Contenders . . . 32
5.3.2 The Comparative . . . 33

6 Fast Quasi-Newton Methods for tSNE **37**
6.1 Fast Quasi-Newton Methods for tSNE . . . 37
6.1.1 Scaling of the Cholesky Factorization . . . 38
6.1.2 Approximating the Gradient and the Cost Function Value . . . 39
6.1.3 The Effect of the Number of CG Iterations . . . 40
6.2 Quasi-Newton tSNE vs BH-SNE . . . 42
6.2.1 Performance Evaluation . . . 43
6.2.2 Different Initialization and Local Optima . . . 45

7 Discussion and Conclusions **51**
7.1 Discussion . . . 51
7.2 Limitations and Future Work . . . 52

A The tSNE Gradient **55**

B The tSNE Hessian **57**
B.1 Useful Variables and Derivatives . . . 57
B.2 Case 1: Same Point, Same Coordinate . . . 58
B.3 Case 2: Same Point, Different Coordinates . . . 59
B.4 Case 3: Different Points, Same Coordinate . . . 60
B.5 Case 4: Different Points, Different Coordinate . . . 60
B.6 A Laplacian-Based Expression for the Hessian . . . 60

Bibliography **61**

1 Introduction

The advent of more sophisticated data collection methods and technologies in different disciplines has marked the start of an era in which large datasets are the norm. A clear example of this can be found in the biological sciences, where different approaches to gather cellular-level data have been devised. For instance, mass cytometry [38] enables obtaining information of single cells with great detail; which has been used to understand how complex systems, such as the immune system, work in humans [18]. Due to the sheer level of detail that tools like this capture, resulting datasets can have millions of rows, each corresponding to a cell, and hundreds of columns with the characteristics that were measured. Accordingly, different tools and pipelines have been devised to analyze these large amounts of high-dimensional data.

Many of the available systems offer the possibility to visually explore the data to develop an intuition about its structure. Despite its apparent simplicity, this is no easy task, mainly because, as humans, we cannot picture more than three dimensions. Among the different ways that have been proposed to circumvent this limitation, one of the most widely used consists in using a dimensionality reduction (DR) technique to obtain a low-dimensional representation of the data that can be then visualized in a scatter plot. Figure 1.1 exemplifies this process. Interestingly, even though DR as a subfield of machine learning has been an active research topic for the past decades, it was not until the advent of graph-based nonlinear methods (NLE) such as t-Distributed Stochastic Neighbor Embedding (tSNE) that its popularity as a data visualization tool exploded. To understand the reason behind this, it is useful to define more precisely the DR problem and take a look at the landscape of available techniques.

The problem of DR can be cast as an optimization one: having two input datasets $X \in \mathbb{R}^{N \times D}$ and $Y \in \mathbb{R}^{N \times d}$ of mismatched dimensionalities (where $d << D$), the goal is to either maximize their similarity or minimize their difference, depending on a predefined cost function. Existing DR techniques can be either linear or nonlinear and convex or nonconvex [56]. The first of these categories concerns the processing of the input datasets X and Y. For instance, in Principal Components Analysis (PCA), a linear technique, the covariance matrix is used to construct a linear mapping from the high to the low-dimensional space. In contrast, Isomap, a nonlinear method, uses the matrix of pairwise geodesic distances between points. The second cat-

sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Label
5	3.4	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
6.3	2.5	4.9	1.5	versicolor
6	3	4.8	1.8	virginica
7.2	3.6	6.1	2.5	virginica

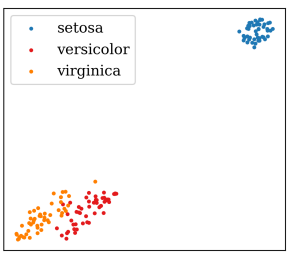

Figure 1.1: An example of using dimensionality reduction techniques for the visualization of high-dimensional data. In the table on the left, we present a sample of the Iris dataset, which contains 150 rows and 4 features, or dimensions. The last column of the table corresponds to the label of each instance, which is one of three possible types of flowers: Setosa, Versicolor, and Virginica. The image on the right corresponds to a scatter plot visualization of the two-dimensional embedding produced with tSNE. Note how, in the embedding, flowers with similar characteristics formed clusters of points.

egory, convexity, corresponds to a structural attribute of the cost function. Convex functions, like those used by PCA and Isomap, have a single optimum, which can be directly obtained by solving a generalized eigenproblem with standard linear algebra routines. On the other hand, nonconvex functions, like tSNE's, might have multiple local optima, forcing the usage of iterative optimization techniques.

Although convex techniques can perform well in synthetic datasets, they can fail to produce accurate embeddings in real-life ones, which often have a more complicated structure. The extra complexity of nonconvex methods, together with their nonlinear modeling of the problem, seems to make them more flexible and capable of embedding datasets that lie in nonlinear manifolds [56]. Of interest to us is tSNE, which has proven to be particularly good at preserving the local structure of the data. In Figure 1.2, we show the computational pipeline that tSNE follows to generate the embeddings. Besides differences in the modeling of the optimization problem, this pipeline is the same as in other NLE techniques such as LargeVis [51] and UMAP [31].

The NLE computational pipeline has two stages. In the initialization phase, the distances between the points in the high-dimensional space are transformed into affinities using a nonlinear kernel function. Having this, an optimal embedding Y^* is found with an iterative optimization algorithm. Naively implemented, this pipeline cannot scale to datasets of more than a few thousand points. On one hand, computing the high-dimensional matrix of affinities requires comparing all pairs of D-dimensional points, which takes $O(DN^2)$. On the other, the complexity of the second step not only depends on the time that it takes to complete each iteration but also on the overall number of iterations that are needed to converge. Assuming that we use gradient descent, as it is often the case, the former takes $O(N^2)$ time, because computing the gradient also requires all pairwise interactions, and the later is $O(1/k)$, where k is the iteration count.

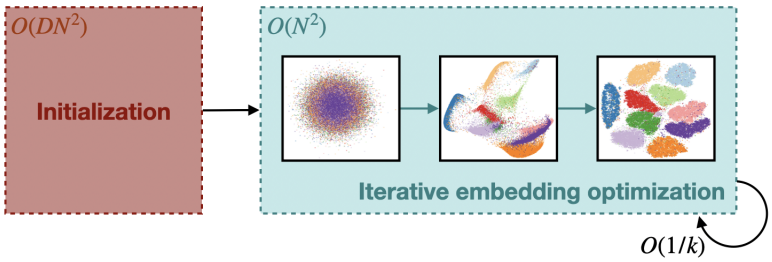

Figure 1.2: The computational pipeline of graph-based nonlinear embedding (NLE) techniques. The pictures inside the right component of the pipeline, iterative embedding optimization, present an example of how this iterative procedure progresses for tSNE with the MNIST dataset.

In the past decade, several advances on different parts of the pipeline have significantly reduced its complexity, enabling the analysis of larger datasets. Regarding the optimization phase, which is the focus of this work, the existing research is fragmented into two branches. While the first is concerned with reducing the complexity of the iteration, the second focuses on reducing the number of iterations that are needed to obtain a suitable embedding. This separation is problematic because the methods that can converge faster, usually do not take advantage of more efficient iterations, which hinders their practical utility. Taking this into account, this work aims to bridge this gap by proposing a set of methods that can both converge in fewer and faster iterations. To do this, we built on top of several recently proposed techniques that we introduce below.

Iterative optimization algorithms can achieve quadratic convergence ($O(1/k^2)$) by using the Hessian of the cost function to obtain a more effective sequence of iterates. However, in the NLE context, the full Hessian is complex to compute and can be hard to use given that, if indefinite, it would produce iterates that do not reduce the cost function value. Fortunately, it has been shown how, by using carefully crafted Hessian approximations, like the Spectral Direction (SD) presented in [58], it is still possible to obtain an improved convergence rate. Thus, the foundation of this work is SD, which is attractive for two reasons. First, it is sparse and constant, and therefore can be constructed efficiently. Second, it is positive definite, which makes it compatible with several linear solvers such as the Cholesky factorization and conjugate gradients (CG). To ensure that the iterative procedure converges, Hessian approximations need to be used with optimization frameworks such as Line Search (LS) and Trust Region (TR) [36]. Even though TR can have some advantages over LS, such as the production of better-scaled iterates, its performance with SD for the optimization of tSNE has not been studied. With this in mind, a common thread that runs throughout this work is the comparison of two quasi-Newton methods that we refer to as SD-LS and SD-TR.

The main goal of this work is to enable these quasi-Newton methods to scale to large datasets. For this purpose, we focused on the within iteration operations that are critical for their performance, identifying potential bottlenecks, and addressing several of them. First, we demonstrated experimentally that it is possible to use approximate gradients and cost function evaluations, which we computed with the Barnes-Hut (BH) algorithm [3], without affecting the robustness of the LS and TR frameworks. Then, we analyzed several linear solvers that are commonly used to obtain the next iterate and concluded that truncated CG is a better alternative. Finally, we explored the effect that different parameters such as the initial step size in LS and the initial trust-region radius in TR can have in the convergence rate of the algorithms. The resulting quasi-Newton methods, SD-LS-CG and SD-TR-CG, can embed large datasets faster than BH-SNE, one of the fastest and most popular Hessian-free tSNE implementation [55].

Besides the two fast quasi-Newton methods here proposed, this work makes several other contributions worth highlighting:

1. **Exploration of the Trust Region optimization framework.** Besides the work in [33], the effectiveness of TR has not been studied in the context of NLE. We evaluated its potential as an alternative of SD-LS for tSNE and show how the way it computes the next iterate can affect its convergence and the appearance of the resulting embedding.

2. **In-depth study of the usage of the SD approximation with tSNE.** In [58], the authors focused on a constant version of SD that is consistently good across different NLEs. Firstly, we showed how in the case of tSNE, SD's performance can be improved by using additional iteration-dependent information. Secondly, we replicated the results of the original work, which led us to the conclusion that SD, when paired with LS and TR, is better than other optimization methods such as nonlinear conjugate gradients and L-BFGS.

3. **Sensitivity analysis of different parameters of the proposed quasi-Newton methods.** Besides the approximation of the cost function and the gradient, we identified other parameters that can significantly affect the performance of these methods and found which values of these parameters consistently yield better results.

This work is going to be structured as follows. In Chapter 2 we will start by contextualizing NLEs, discussing the different ways in which they have been improved, and presenting different practical cases in which these tools have been used successfully for the visualization of high-dimensional data. In Chapter 3, we introduce the theoretical framework used for our method, which includes a primer in unconstrained optimization and a detailed description of the t-Distributed Neighbor Embedding building blocks. Then, in Chapter 4, we will describe the computational framework that supported our experiments and the datasets that were used to compare the different methods. After this, in the following two chapters, we present the results of our experiments. In Chapter 5, we present the outcomes of the replication and extension of [58]; and in Chapter 6 we explain the construction and evaluation of our fast quasi-Newton methods. Finally, in Chapter 7, we will discuss the results and the limitations, and present potential future work.

2

Related Work

In this chapter, we present a survey of the state of the art of the visualization of high-dimensional datasets through the usage of dimensionality reduction techniques. Concretely, we focus on tSNE and other related graph-based nonlinear embedding techniques (NLE). We start this chapter by motivating the usage of NLE techniques for the visualization of high-dimensional data in Section 2.1. Even though NLEs often have differing modeling choices, they share the same computational pipeline that was introduced in the previous chapter. In Section 2.2 we first explore the space of NLE methods and then go deeper into the different improvements of the pipeline that have enabled the processing of larger volumes of data. To motivate the reader and show the usefulness of NLEs at this task, we finalize this chapter in Section 2.3 presenting different successful use cases of these techniques.

2.1. Dimensionality Reduction for Visualizing High-Dimensional Data

Starting from a high-dimensional dataset $X \in R^{N \times D}$, the goal of dimensionality reduction (DR) techniques is to find a low-dimensional representation $Y \in R^{N \times D}$, where $d << D$, that better preserves the structure in the data. The latter can be divided into the global and local structure. While the former has to do with preserving the distances between all points, the latter focuses on preserving the relationships between those points that were close by in the original space. In the visualization of high-dimensional data, methods that preserve the local structure of the data have been favored. This includes techniques like t-Distributed Stochastic Neighbor Embedding (tSNE) [54], Uniform Manifold Approximation and Projection (UMAP) [31] and LargeVis [51], which popularity have risen in the past years. We can attribute their success to two "ingredients": their nonlinear transformation of the input data and their usage of nonconvex cost functions.

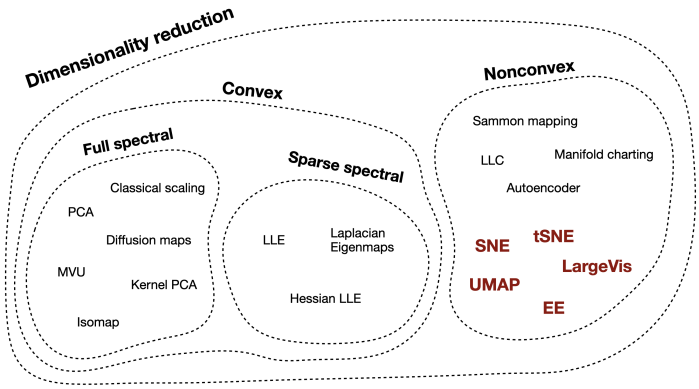

Figure 2.1: The landscape of dimensionality reduction techniques. We separate the different methods taking into account whether their cost function is convex or nonconvex. This work focuses on tSNE, which uses is a nonconvex cost function: the Kullback-Leibler divergence.

In general, DR techniques are cast as optimization problems, which means that a cost function C is used to measure the similarity, or dissimilarity, of X and Y. If C has a single optimum, we say the method is con-

vex; in contrast, if it might have several local optima, we say that it is nonconvex [56]. In Figure 2.1 we show the general landscape of DR techniques, taking this classification into account. Besides convex/nonconvex, we can also differentiate DR techniques depending on their processing of the input datasets. Most of these methods do not work directly with the raw datasets X and Y and instead transform them into square matrices of pairwise distances or similarities between points. A classical example of a linear technique is Principal Components Analysis (PCA) [19], which relates pairs of points using the squared Euclidean distance. In contrast, Isomap, a nonlinear method, uses geodesic distances (shortest paths between pairs of points on the u-Nearest Neighbor graph computed from X) when constructing the matrix of similarities. Experimentally, it has been shown how DR methods that use convex cost functions are not as effective at embedding real-life, highly nonlinear, data, regardless of how the input datasets are processed [56]. For this task, nonconvex and nonlinear techniques like autoencoders [17] and tSNE have performed better. This can be evidenced in Figure 2.2, where we compare the resulting embeddings of a convex linear technique (PCA), several convex nonlinear ones (Isomap [52], Laplacian Eigenmaps (LE) [4] and Locally Linear Embedding (LLE) [45]), and two NLEs (tSNE [54] and UMAP [31]).

There are two main reasons behind the success of NLEs at embedding more complex datasets. First, like Isomap, they model the relationships between points in a nonlinear way by using a u-NN graph. Importantly, instead of using distances, or similarities, between points, which grow as points get farther away from each other; NLEs transform the similarities into affinities, which have the opposite behavior. The second reason for the effectiveness of NLEs is that they use nonconvex cost functions which seem to grant them more flexibility, although they might be harder to optimize. As can be observed in Figure 2.2, the usage of affinities and nonconvex cost functions enables NLEs to produce embeddings that better represent the local structure in the data: clusters form when their corresponding points in the high-dimensional space are nearby. The visual outputs of NLEs depend to a large extent on the choice of cost function or, as we call it, the modeling of the problem. Besides this element, these methods are similar implementation-wise, sharing the computational pipeline presented in Figure 1.2, which has enabled a significant level of cross-pollination between the different techniques. In the next section, we delve deeper into these aspects of NLEs, presenting the different modeling alternatives and improvements of the computational pipeline that have been proposed.

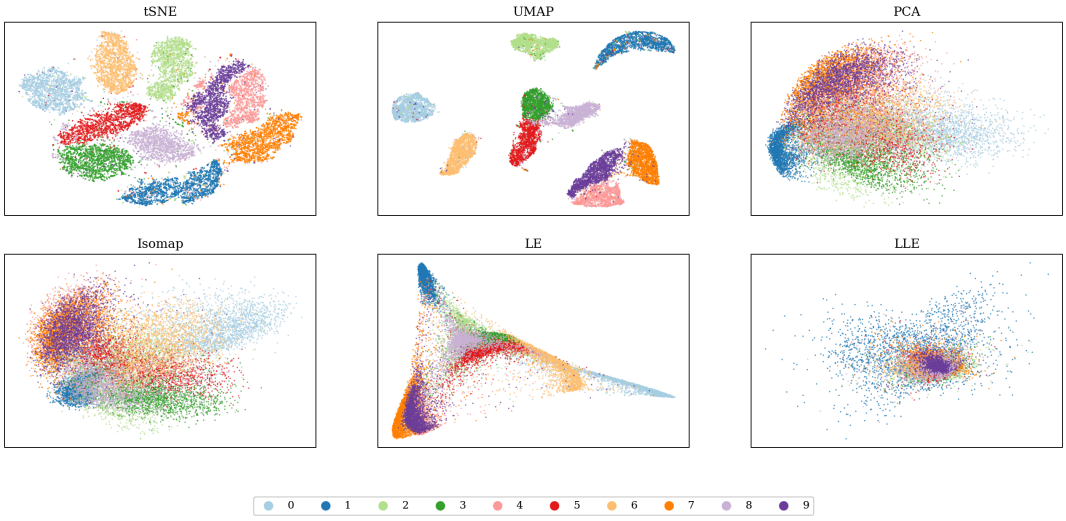

Figure 2.2: The outcome of embedding a sample of the MNIST dataset ($N = 20,000$ and $D = 784$) with different DR techniques: one convex linear (Principal Components Analysis (PCA)), tree convex nonlinear (Isomap, Laplacian Eigenmaps (LE) and Locally Linear Embedding (LLE)), and two nonconvex graph-based nonlinear embeddings (tSNE and UMAP). Note how the nonconvex nonlinear techniques, tSNE and UMAP, produced the best-looking embeddings with clearly defined clusters that correspond to similar points in the high-dimensional space.

2.2. Graph-based Nonlinear Embedding Techniques

In this work we are focused on improving the convergence speed of tSNE. Given that it is a graph-based nonlinear embedding technique (NLE), in this section we discuss what has been done to improve different parts of the NLE computational pipeline, which can also be used to speedup tSNE. In Subsection 2.2.1, we start by contrasting the modeling choices of different NLEs, which directly affects the quality of resulting embeddings.

Having this, in the rest of the section we discuss work in specific parts of the pipeline. In Subsection 2.2.2, we show how the matrix of high-dimensional affinities can be constructed in less time by using a sparse u-Nearest Neighbor graph instead. In Subsections 2.2.3 and 2.2.4, we investigate what has been done to accelerate the optimization step, which includes more efficient iterations and iterative algorithms that converge faster. Finally, in Subsection 2.2.5 we present improvements that cannot be put in any group because they affect the whole pipeline. Throughout this section we will emphasize tSNE-related concepts to prepare the reader for the next chapter, where we delve deeper into its inner workings.

2.2.1. Modeling of the Problem

In NLEs, the way the final embedding of a specific datasets looks depends on how the distances between points in the high and low-dimensional points are transformed into affinities. One of the first NLEs, that we cover here, is the Stochastic Neighbor Embedding (SNE) [16]. In SNE, distances are converted into probabilities, using the Gaussian kernel presented in Equation 2.1. This is done for both X and Y, which, after normalizing each row with its sum, yields the matrices of conditional probabilities P_{SNE} and Q_{SNE}, respectively. In the high-dimensional space, the bandwidth parameter σ_i is found for each point using a bisection search procedure. In the low-dimensional space, this parameter is assumed to be one. To obtain the optimal embedding Y, the Kullback-Leibler divergence, which is presented in Equation 2.2, is minimized gradient descent.

$$\text{gaussian}(d_{ij}) = \exp(-||d_{ij}||_2^2/2\sigma_i) \tag{2.1}$$

Although the embeddings produced by SNE were already significantly better than those of its predecessors, it had two main issues. First, optimizing the cost function could be tedious as a result of P_{SNE} being asymmetric due to the adaptive bandwidth. Second, it tended to produce embeddings that were crowded, meaning that they were too tight and the separations between clusters were not clear. The optimization can be simplified by symmetrizing P_{SNE}, which yields a matrix of joint probabilities that is easier to handle. For the crowding problem, Van der Maaten and Hinton proposed using a Student's t-distributed kernel with one degree of freedom ($\nu = 1$ in Equation 2.3) to more faithfully model the distances between the low-dimensional data points in [54]. This change of low-dimensional kernel, together with the symmetrizing and normalization of the affinity matrices resulted in the t-Distributed Stochastic Neighbor Embedding (tSNE). Although other ways to alleviate the crowding problem such as UNI-SNE [10] were proposed, they were trickier to optimize and did not yield embeddings of the same quality of those produced by tSNE.

$$KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{2.2}$$

Since the introduction of tSNE, there have been several works that aim at better understanding its theoretical underpinnings. For instance, in [24] it was shown how by using less than one degrees of freedom in the Student's t-distribution ($0 < \nu < 1$ in Equation 2.3), better embeddings can be obtained. In their experiments, it is possible to observe how heavier tailed distributions, that arise with the reduction of degrees of freedom, facilitate the formation of better-separated clusters in the embedding. Another aspect that has been explored is the computation of the adaptive bandwidths of the high-dimensional distribution. In [59], the authors characterize these high-dimensional affinities, which they call entropic affinities, and use root-finding methods together with a tree-based initialization scheme to compute them. Not only their approach produces good quality affinities but is also faster than the commonly used bisection method [16, 54].

$$\text{student}(d_{ij}) = \frac{1}{(1 - ||d_{ij}||_2^2/\nu)^\nu} \tag{2.3}$$

Two alternative ways to tSNE that have received a lot of attention recently are LargeVis and UMAP. These also encode the distances as affinities using Gaussian and generalized Student's t-distributed kernels, but they relate the resulting matrices using different cost functions. In LargeVis, the problem is cast as finding the low-dimensional affinities that maximize the likelihood of corresponding to the high-dimensional ones [51]. In UMAP, which is based on Riemannian geometry and fuzzy simplicial sets, the goal is to minimize the cross-entropy between the topological representations of the high and low-dimensional datasets [31]. Similarly to [24], UMAP optimizes several parameters of its low-dimensional kernel, obtaining a fatter tailed distribution, which in turn results in better-separated clusters. In contrast with tSNE, with a correct setting of

hyperparameters, embeddings produced by LargeVis and UMAP can often better capture the local structure in the data, resulting in embeddings with clearly separated clusters.

The last NLE that we will discuss is the Elastic Embedding (EE) [8]. Similar to SNE, it uses Gaussian affinities to encode both the high and low-dimensional points. But instead of comparing them using the Kullback-Leibler divergence, it defines a simpler cost function that has two terms: an attractive one, to keep similar points together, and a repulsive one to spread out all points. Even though visually the embeddings produced by EE are not better than those of tSNE, LargeVis, and UMAP; its simpler cost function has enabled a more in-depth study of NLEs. Relevant to this work is the unified framework of NLEs cost functions, their gradients, and Hessians presented in a subsequent paper [58]. The different quantities are expressed in terms of Laplacian matrices, which are useful when devising alternative optimization algorithms with better convergence properties as we will see below.

2.2.2. Approximate uNN Graph Construction

Regardless of their modeling choices, all NLEs construct a high-dimensional matrix of affinities that remains constant throughout the optimization. Given that interactions between all pairs of points are considered, a naive computation of this matrix takes $O(DN^2)$ time, where D is the dimensionality of the input data. Interestingly, when using the kernels introduced in the previous subsection to transform the distances between points, after a certain distance, affinities will be very close to zero. In consequence, it is possible to consider only the u-Nearest Neighbors of each point without a significant drop in performance [55]. Because of its applicability in other fields such as information retrieval, the problem of efficient approximate u-Nearest Neighbor (uNN) graph computation has been an active area of research for the past decades. The different existing approaches can be divided into two main groups that we describe below.

The first of these consists of the usage of hierarchical data structures such as kd-trees [6], vantage-point (VP) trees [63] and random projection trees [11]. These methods index the points in a hierarchical structure and then use it to be able to query approximate nearest neighbors more efficiently. The effectiveness and performance of these techniques depend on the algorithm that is used to partition the space. Although in [55] VP trees were used successfully, this approach can become inefficient with large dimensionalities, which are common. Alternatives include using a forest of randomized kd-trees [41] or the recently favored random projections. By randomly partitioning the space, the latter can reduce the time that it takes to retrieve the nearest neighbors. In LargeVis, the output of a random projection trees stage is further enhanced by using nearest neighbor exploration techniques [13, 51], which use the idea that "the neighbor of my neighbor might also be my neighbor" to iteratively refine the NN approximation.

The second approach for efficiently computing the uNN graph is based on Locally Sensitive Hashing (LSH). LSH techniques use a hash function to map similar points to the same "bucket". Then, when the u-Nearest Neighbors of a point are requested, the bucket where this point belongs, and also other buckets that are nearby, are retrieved and their points returned. LHS methods were initially designed for similarity search, where only the closest point is retrieved. For this reason, they might perform suboptimally for the task of retrieving the first u neighbors. Additionally, special hash functions must be designed for different distance metrics [1, 21, 37], which can be a complex endeavor. Despite these issues, LSH has been successfully applied in the context of NLE, greatly reducing the time that it takes to compute the high-dimensional matrix of affinities [9].

Among the two approaches, hierarchical structures are more widespread than LSH. This is likely because the former is simpler to implement and therefore there are more libraries available that can be easily integrated with any NLE. In the future, we expect that LSH methods will face greater adoption, especially as they become more versatile, easy to use, and more different distance functions are incorporated. With this, we now turn to the second stage of the pipeline with is the optimization of the embedding. We grouped the relevant work according to their main focus: iterative optimization algorithms with better convergence properties, or efficient computation of the gradient.

2.2.3. Iterative Optimization

There are two main components in the NLE computational pipeline. The first is the construction of the high-dimensional matrix of affinities, which we covered in the previous subsection. The second, which we detail in this subsection, corresponds to the optimization process that is used to find an embedding that optimizes the given cost function. As a consequence of NLEs' cost functions being nonconvex, iterative algorithms must be used. In the literature, we identified two main trends. On one hand, most of the current NLE implementations still use the highly tunned version of gradient descent that was first proposed in the original tSNE paper

[54]. With this as the starting point, recent work has focused on further tunning it and understanding its convergence properties. On the other hand, there has been some interest in exploring alternative optimization approaches that can exploit the structure of the cost function and use higher-order information such as the Hessian. In the following paragraphs, we further discuss these two independent developments.

Gradient descent, or the method of steepest descent as it is also known, has been the preferred way of optimizing NLE cost functions, mainly because of how easy it is to implement and its low per-iteration cost, which only requires the computation of the gradient. At each iteration, the next iterate Y_{k+1} is found using Equation 2.4. In its vanilla implementation, the learning rate or step size α is fixed at the beginning by the user. If it is set too high, the algorithm can end up oscillating around the optimum and if it is too low, it can take a lot of iterations to converge. This was greatly improved in the original tSNE [54] paper, where a momentum term and an adaptive learning rate scheme [20] were used. An important parameter of this enhanced gradient descent is the initial step size α_0. In [5], it was shown how the size of the embedding should be taken into consideration when setting this value. Additionally, the authors also show the positive effect of letting the optimization algorithm run for more than the usual 1000 iterations.

$$Y_{k+1} = Y_k + \alpha \nabla C(Y_k) \tag{2.4}$$

Besides the general optimization scheme, different tSNE-specific tricks have been devised to help the algorithm avoid local optima and produce embeddings with more space between clusters. The first of these, known as "early exaggeration" (ee) consists of scaling the input matrix of affinities by a constant term for some iterations at the beginning of the optimization [54]. This helps tSNE avoiding local optima in which, for example, a cluster of related points would be split by another cluster. Although in [55] and a lot of subsequent work 12 was blindly used as the ee factor, recently, different work has systematically analyzed the effect of this parameter, confirming the validity of this value and suggesting a relationship with the learning rate and the size of the embedding [5, 27]. A related similar technique known as "late exaggeration" (le) was briefly proposed in [28]. The authors found that by exaggerating the high-dimensional affinities towards the end of the optimization, resulting clusters can have more space in-between. Despite its apparent benefits, more studies should be conducted to verify that it is a reasonable thing to do. It must be noted that these add ons are, for the most part, a result of empirical observations by the authors and therefore are highly tunned for tSNE and, in most cases, lack theoretical support. Later on in this document, we show that we can obtain similar effects in the embedding without the usage of these extra parameters.

Moving away from gradient descent, a smaller subset of the research efforts have been devoted to devising alternative optimization approaches for NLEs. One way in which the convergence of iterative optimization algorithms can be accelerated is by using higher-order information such as the Hessian, which provides curvature information of the cost function [36]. A first attempt to use the Hessian was recorded in [33] where a trust region based Newton method was proposed for SNE. Although the results showed that using Hessian information could significantly reduce the number of iterations needed, this method was not further developed. Probably, because of how expensive each iteration was as a result of having to compute the complete Hessian. This pitfall was remedied in recent work, where approximations of the Hessian were used instead. The simplest way to do so is by using a diagonal scaling matrix, which can be obtained from the Laplacian of the graph of high-dimensional affinities (see Chapter 3 for an explanation of how the Laplacian is built). In [29] and [8], the authors tried this approach for the cost functions of sSNE, tSNE, and EE and found that it boosted the convergence of the algorithm without a significant computational overhead.

This idea of using approximations of the Hessian was further refined and validated in [58], which presents a systematic approach for using line search based quasi-Newton methods for optimizing different NLE cost functions. In the paper, Laplacian-based expressions of the gradient and Hessian of the different cost functions were derived. Then, these were used to construct positive-definite, fast to compute, and as close as possible approximations of the Hessian. In the results, the authors found that the Spectral Direction, which corresponds to the Laplacian matrix of the high-dimensional affinity graph, yielded the best results, outperforming the traditional gradient descent and other unconstrained optimization methods such as nonlinear conjugate gradients [44] and L-BFGS [64] in terms of the number of iterations needed for convergence. Given how promising were the results using the Spectral Direction, it occupies a central role in this work. Concretely, we are interested in better analyzing its behavior with tSNE and extending it to handle large scale datasets.

To finalize this subsection, we return to gradient descent, but in the stochastic setting. Two recent NLEs, LargeVis [51] and UMAP [31], have successfully used stochastic gradient descent to optimize their cost functions. This is possible because, in contrast to tSNE, these methods do not normalize the low-dimensional affinities. As a consequence, they can decouple the updating of the iterate in mini-batches of the affinities. To

perform the sampling, these techniques follow the negative sampling approach presented in [32]. In practice, this means that for each high-dimensional, or attractive edge, they randomly sample m low-dimensional, or repulsive, edges. Although using stochastic gradient descent does not necessarily results in a speedup of the optimization (number of iterations stays the same and can even increase) by running multiple iterations asynchronously, significant performance gains can be obtained.

2.2.4. Approximation of the Gradient

As was mentioned in the previous subsection, gradient descent is the preferred way to optimize of NLEs' cost functions. Given that the gradient of these functions depends on the low-dimensional matrix of affinities, which in turn requires the evaluation of a kernel for every pair of points in the embedding, each iteration of this algorithm has a time complexity of $O(N^2)$. In recent years, there has been a lot of effort put into accelerating this step by approximating the gradient. Different approximations have become available for the NLEs that we have discussed. We start with gradient approximations for tSNE, which rely on the fact that the gradient can be separated into a set of attractive and repulsive forces as can be observed in Figure 2.3. Of these, the former depends on the matrix of high-dimensional affinities and therefore they can be efficiently computed exploiting its sparsity. In contrast, the latter requires $O(N^2)$ comparisons. Fortunately, there are several ways in which this complexity can be reduced.

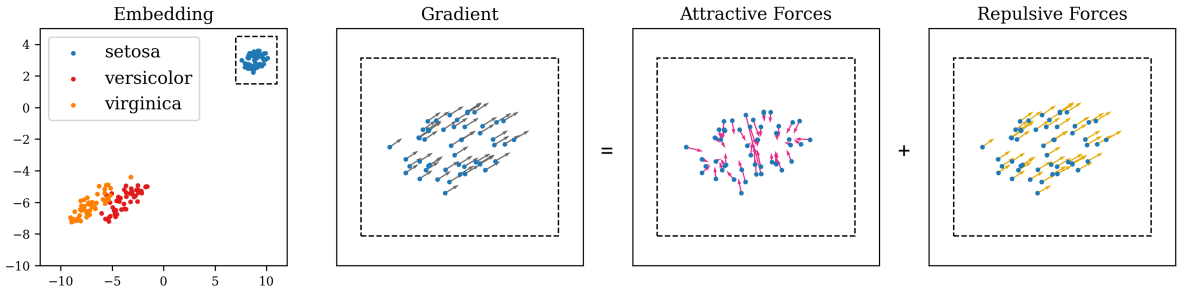

Figure 2.3: Visualization of the gradient of the Kullback-Leibler divergence for a two-dimensional embedding of the Iris dataset produced with tSNE, which is presented in the left-most scatter plot of the figure. The three charts on the right depict the gradient, and its decomposition into attractive and repulsive forces as a vector field plotted over one of the clusters of the embedding. It is possible to observe how attractive forces are drawing points towards the center of the cluster, while the repulsive ones are pulling away from this cluster from the other two. Note that the length of the vectors was automatically computed on a per plot basis to ease visualization and, because of this, vector lengths of different plots are not directly comparable. In the example that we considered for this figure, the attractive forces were small, and therefore their effect on the gradient is hard to appreciate. For this reason, the gradient looks similar to the repulsive forces.

In [55], two approximation algorithms that rely on tree-based data structures were presented. The first approach, based on the Barnes-Hut algorithm [3], reduces the number of pairwise comparisons that need to be carried out to $O(N \log N)$ by using a tree to index the points and then using it to query clusters of points as opposed to single particles. For example, if a point i is too far from two other points j and k that are close by; instead of performing 2 comparisons ($i-j$ and $i-k$), the center of mass of the cell to which j and k belong is used instead. The tSNE method equipped with this algorithm is known in the literature as BH-SNE and was the first to enable the visualization of several hundreds of thousands of points, using an NLE. Given that in this work we use the BH approximation to accelerate quasi-Newton methods, we provide a more detailed description of this algorithm in Chapter 3

The Barnes-Hut algorithm compares a particle against cells of particles using the hierarchical structure. In theory, further speedups can be obtained if cells are directly compared against cells. In this case, two trees are built and, when two groups of points are sufficiently far from each other, their cells are compared as opposed to all their particles. This approach, known as the dual-tree (DT) approximation [15] was also tested in [55], but the authors found that the performance gains were not as significant as expected and therefore it has not been used with NLEs after that. It is important to note that there are no bounds on the approximation errors of BH and DT. Even though this has not been a problem when used in gradient descent, a central question in this work is whether the BH approximation can be "safely" used in quasi-Newton, which relies on line searches and trust regions to set their step size.

Another approximation approach that relies on the Fast Fourier Transform (FFT) was introduced in [28]. The resulting method denoted Flt-SNE, was able to reduce the computational complexity of the gradient computation to $O(pN)$, which is linear in the number of points. Flt-SNE starts with an interpolation step in

which the N points are interpolated "down" to p interpolation nodes that are equally spaced on a squared grid. Then, having this grid, FFT is used to approximate the pairwise interaction in $O(p \log p)$. Finally, the information at every p is interpolated back "up" to the N points of the embedding. Note that in this procedure, the most expensive part of corresponds to two interpolation steps which depend on N and p. In the paper, the authors found $p = 3$ to be optimal, making Flt-SNE computational complexity close to linear in the number of points. In practice, when compared to BH-SNE, Flt-SNE is slower for smaller datasets, but quickly outpaces BH-SNE as the number of points grows.

There are two ways to approximate the gradient that have been devised for tSNE and that assume that the space of the embedding is two dimensional. The first of these, pixelSNE [22], attempts accelerates BH-SNE by reusing the indexing structure across iterations and by computing a coarser approximation, using the pixel as the maximum level of granularity to be allowed. Although this approach can be fast for small screen resolutions, to obtain faithful representations of the data usually higher-resolution grids are needed which makes it harder for it to scale to larger datasets. The second method, known as GPGPU tSNE [43], uses a completely different approach that is based on the reinterpretation of the gradient so it can be computed using a scalar and a vector field. This is done so the parallel capabilities of GPUs can be used in the following way. First, the fields are computed in parallel using GPU shaders and using a hardware feature known as texture splatting to aggregate the contributions from different points. Then, these fields can be consulted in $O(1)$ to compute the gradient. In the paper, it is shown how this method is much faster than BH-SNE and comparable in performance to CUDA tSNE, another hardware-accelerated tSNE variant that we will discuss below. Before moving to other NLEs, we note that in our approach we will use the BH algorithm. Nevertheless, in future work, it would be interesting to explore the effect of the other approximations.

Tree-based methods have been used in other NLEs besides tSNE. In [60], a Fast Multipole Method (FMM) for approximating the gradient of EE in $O(N)$ was proposed. Similar to BH, FMMs consider scenarios in which individual interactions can be grouped into cell interactions. But it differs from BH in several important ways. First, it takes into consideration the kernel that is used to develop an expansion of the sum that can be computed in $O(N)$. Given that EE uses the Gaussian kernel for the low-dimensional affinities, in the paper the authors adopt an existing FMM known as the Fast Gauss Transform [49]. The problem with FMMs is that, although they have an edge in performance, their dependence on a specific kernel makes them hard to adapt to NLEs that use the Student's t-Distributed kernel, which is normally used. An important thing to consider about FMMs is that, in contrast with BH, they provide error bounds, which, in our case, could be useful for adding safeguards to the line search and trust region of our quasi-Newton methods.

We finalize this section with the alternative approach proposed by UMAP [31] and LargeVis [51]. As was mentioned before these methods abandon the normalization of the low-dimensional interactions, which in turn enables the usage of stochastic gradient descent (SGD) for the optimization. In this case, in contrast with previous methods that attempted to obtain an approximation as close as possible to the true gradient, noisy updates are used by sampling a limited number of repulsive edges using the negative sampling procedure proposed in [32]. For this, they iterate over the attractive forces, and, for each one of them, they sample m negative edges, with $m << N$. UMAP and LargeVis have shown that it is possible to optimize NLEs' cost functions using noisy iterates. It would be interesting to see whether this stochastic approximation of the gradient can be somehow be adapted to work with tSNE's cost function in the context of gradient-based and quasi-Newton optimization.

2.2.5. Hardware Accelerations

In the previous subsections, we focused on the different algorithmic improvements that have made it possible to scale NLEs to hundreds of thousands of points. Recently, different steps of the NLE pipeline have been extended to leverage specialized hardware, making the interactive exploration of datasets with millions of points possible. We can divide these extensions according to the hardware that they use. For instance, Multicore-TSNE accelerated the construction of the high-dimensional matrix of affinities and the gradient by distributing the uNN search for each point and the computation of each component of the gradient across processors [53]. With these improvements, it was able to obtain speedups of up to 5×, when using eight threads, with respect to BH-SNE. Similarly, the implementations of recent NLEs such as FltSNE, UMAP, and LargeVis have been done in a way that every step in the pipeline takes advantage of as many threads as possible.

For datasets in the order of millions of points, multicore implementations might not be enough. To deal with this case of use, several GPU-accelerated variants have been proposed. In this regard, the best performing method so far is tSNE-CUDA which relies on NVIDIA graphics cards to accelerate the computations [9].

Concretely, it uses FAISS [21], which is a parallel library for similarity search based on LSH, for the computation of the high-dimensional matrix of affinities and a parallel implementation of the Barnes-Hut algorithm for the computation of the gradient [7]. It achieved a 70× speedup with respect to Multicore-SNE, being able to produce a two-dimensional embedding of the GLOVE dataset, which has over two million points, in under ten minutes. Despite its success, the main drawback of tSNE-CUDA is that it requires an NVIDIA graphic card to work. The GPGPU approach proposed in [43] avoids this problem by using compute shaders, which are available regardless of the GPU, to parallelize the computations. This was made possible by their field-based interpretation of the gradient which can be easily implemented using texture splatting.

2.3. Tools and Systems

The improvements described in the previous section have made it possible for NLEs to handle larger amounts of high-dimensional data. In parallel to the upgrades that target specific parts of the NLE pipeline, modifications that add more functionality to it or that rethink the interaction between the end-user and the embedding algorithms have been proposed. In this section, we consider those that follow or could be integrated into the "progressive visual analytics" workflow [48], which argues that visual analytic tools should provide meaningful intermediate results. One of such approaches is Hierarchical-SNE (H-SNE) [40], which initially presents the user with an embedding of a set of global, coarse, landmarks; and then lets the user zoom into specific areas as desired. Embeddings for the landmarks at the different levels of granularity are precomputed, endowing this approach with two benefits: it reduces potential cluttering caused by millions of points visualized all at once, and gives the impression of a smoother, even though the time complexity of the algorithms is the same, experience.

Another recent progressive visual analytics approach A-SNE [41] focuses on the high-dimensional matrix of affinities. It starts with a rough approximation of this matrix and then lets the user refine specific areas as the optimization process advances. The advantage of doing this is that A-SNE avoids the bottleneck of computing a high-quality version of this matrix up front, letting the user immediately start interacting with the data once it has been loaded. An important element that H-SNE, A-SNE, and related tools do not address is the optimal selection of parameters. For this, different protocols and systems, which focus on issues such as the initialization of the data, setting the early exaggeration factor, the perplexity, and the number of iterations that are necessary to obtain reasonable embeddings, have emerged [5, 23]. Although these are based on tSNE, the most popular NLE thus far, we believe their findings should translate to some extent to other embedding techniques.

To conclude this chapter, we highlight the different use cases that these tools and systems, together with an improved NLEs pipeline, have enabled. Starting with biology, they have been used to visualize mass cytometry data to better understand complex parts of the human body such as the immune system [18, 57]. Another emerging area in which NLEs are being used is to enable the understanding of deep neural networks. Usually, this is done, by visualizing their hidden layers, which can contain important information about the features that the model has learned. This has also been done during the training process of the networks to help the user fine-tune them in real-time [42]. Finally, NLEs have also been in the art word to embed data thousands of pictures of paintings [1], which has resulted in new and unexpected experiences for the spectators.

[1] The experience can be accessed at https://experiments.withgoogle.com/t-sne-map

3

Background

In this chapter, we present the theoretical foundations of our work, which aims at accelerating the convergence of the t-Distributed Stochastic Neighbor Embedding (tSNE) by using quasi-Newton methods. To find an embedding $Y \in \mathbb{R}^{N \times d}$ that correctly captures the local structure of $X \in \mathbb{R}^{N \times D}$, where $d << D$, tSNE attempts to minimize the Kullback-Leibler divergence between the two datasets. Taking into account that this function is nonconvex, this is done using iterative optimization algorithms that, after a series of steps, converge to an optimum of the function. When only the gradient is used to generate these steps, the process can take several thousands of iterations to converge. Quasi-Newton methods improve this situation by using the second derivative of the function, the Hessian. To ensure their robustness, these are commonly used with the line search and trust region frameworks, both of which we explore in this work. Having said this, in Section 3.1 we present a general introduction to the problem of optimizing nonconvex functions and the usage of quasi-Newton optimization frameworks. Then, we make this more concrete for the case of tSNE by delving deeper into its cost function and its derivatives.

- $X \in \mathbb{R}^{N \times D}$ is the matrix of high-dimensional points.
- $Y \in \mathbb{R}^{Nd}$, $C(Y) \in \mathbb{R}$, $\nabla C(Y) \in \mathbb{R}^{Nd}$, $\nabla^2 C(Y) \in \mathbb{R}^{Nd \times Nd}$, and $\boldsymbol{p} \in \mathbb{R}^{Nd}$ correspond to the embedding, tSNE's cost function, its gradient, its Hessian and the search direction that is used in iterative optimization methods to move between consecutive iterations, respectively.
- Note how we treat Y, $\nabla C(Y)$ and $\boldsymbol{p}$ as vectors. This is to be consistent with the notation that we use in the equations in the Line Search and Trust Region frameworks. Given a vector V, v_i means that we get the vector of d coordinates of the i^{th} point and $v_i^{(c)}$ that we get the value for coordinate c of the i^{th} point.
- We use $B \approx \nabla^2 C(Y)$ to denote that an approximation of the Hessian, as opposed to the full Hessian, is used.
- We use the subscript k to refer to the value of any quantity at iteration k. For example $\nabla C(Y_k)$ is the value of the gradient at iteration k. To avoid cluttering the explanations, we sometimes omit the Y (i.e. C_k or ∇C_k).
- The Kronecker product $\otimes$ is an operator used to build block matrices. For instance, having a matrix $A \in \mathbb{R}^{N \times N}$, the result of $A^{\otimes} = A \otimes I_d$, where I_d is a $d \times d$ identity matrix, will be a $Nd \times Nd$ matrix. An entry a_{ij} of A, will be transformed into the $d \times d$ block $a_{ij} \times I_d$ in $A^{\otimes}$.

3.1. Quasi-Newton Optimization Frameworks

Nonconvex cost functions such as tSNE's are characterized by the presence of multiple local optima. Because of this, a closed form solution cannot be obtained and an iterative algorithm must be used instead. Starting from an arbitrary embedding $Y_0 \in R^{N \times d}$, iterative methods produce a series of iterates $Y_0, Y_1, ..., Y_k, ..., Y^*$ that eventually reach an optimizer Y^* of the cost function C. For this to work, at every iterate Y_k the neighborhood surrounding it must be smooth. Having this, second-order Taylor expansion,

$$C(Y+\boldsymbol{p}) \approx C(Y) + \nabla C(Y)^T \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T \nabla^2 C(Y+t\boldsymbol{p})\boldsymbol{p} \text{ with } t \in \{0,1\} \tag{3.1}$$

can be used to find a suitable step $\boldsymbol{p}$ to the next iterate Y_{k+1}.

In this section, we cover two frameworks that leverage this approximation of C at the current iterate. In the first, denominated Line Search, we find a direction in which to move, and then decide how much to advance in that direction. In the second approach, known as Trust Region, both of these steps are done at once by finding a direction that minimizes a surrogate of C within a predefined boundary. Figure 3.1 presents an overview of the functioning of these methods on an example two-dimensional function. In the next paragraphs, we give a more detailed description of these frameworks.

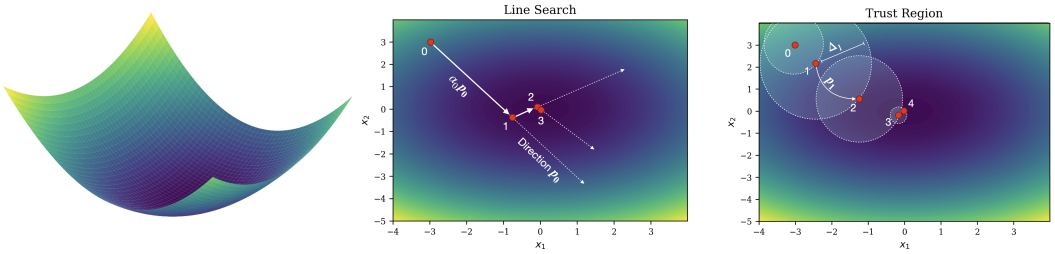

Figure 3.1: Minimization of a two-dimensional quadratic function using the Line Search (LS) and Trust Region (TR) frameworks. The left-most panel presents a surface plot of the function. The center panel illustrates the process of optimizing it using the LS framework. As can be observed, at each iteration, first we find the direction and then the step size. In the panel on the right, we present the optimization process using the TR framework. In this case, we find a direction within the radius, which, in this case, forms a spherical area due to the usage of the 2-norm. At each iteration, the size of the radius depends on the quality of the direction at the previous iteration.

3.1.1. Line Search

In the Line Search (LS) framework, the next iterate is found by moving from the current iterate in a given direction. This can be written as,

$$Y_{k+1} = Y_k + \alpha_k \boldsymbol{p}_k \tag{3.2}$$

where $\boldsymbol{p}_k$ is the direction and α_k is the step size. For the rest of this subsection, we avoid the iteration subscripts to avoid cluttering the explanation.

The first step in LS consists of finding a suitable direction, which can be done by zeroing the derivative of the Taylor expansion with respect to $\boldsymbol{p}$ and solving the resulting linear system,

$$B\boldsymbol{p} = -\nabla C(Y) \tag{3.3}$$

for $\boldsymbol{p}$. A thing to note in this expression is that, often, the Hessian $\nabla^2 C(Y)$ is too expensive to compute, and therefore an approximation B is used instead. Importantly, the latter must be positive definite to ensure that the direction is a decent one.

In practice, we can solve Equation 3.3 directly or iteratively. In the case of the former, we consider Cholesky factorization. The idea is to decompose the Hessian matrix in a way that $B = LL^T$. Then, the factor L can be used to find $\boldsymbol{p}$ by first solving $Lx = -\nabla C(Y)$ and then $L^T\boldsymbol{p} = x$. This process is known as forward-backward substitution and is more efficient than alternatives such as Gaussian elimination. Iterative methods are an alternative to direct solvers and are preferred when the Hessian matrix changes regularly or it is too big to fit in memory. In this work, we consider the conjugate gradients (CG) method, which iteratively builds an approximation of the solution, stopping when the desired precision is reached. It is important to stress the fact that, although methods for indefinite or semidefinite matrices exist, in this work we assume that the matrix is positive definite, which is a requirement of both Cholesky factorization and CG. This is an important motivator for the Hessian approximations that we present towards the end of this chapter.

Having a suitable descent direction, the next step in LS consists of finding an optimal step size. This is done by solving the one-dimensional optimization problem,

$$\underset{\alpha}{\operatorname{argmin}}\, \phi(\alpha) = \underset{\alpha}{\operatorname{argmin}}\, C(Y+\alpha\boldsymbol{p}) \text{ with } \alpha > 0 \tag{3.4}$$

where α corresponds to the desired step size. The reason this value cannot be set manually is that the algorithm could end up oscillating around the optimum, if α is too big, or taking more iterations to converge if it is too small. Taking into account the nonconvex and, in the case of tSNE's Kullback-Leibler divergence, highly nonlinear nature of C, the minimizer of ϕ cannot be found exactly and therefore an iterative procedure which is known as line search is required, hence the framework's name. Although there are many sophisticated algorithms available for this purpose, these are often too expensive since they require multiple evaluations of the gradient and the cost function. Therefore, we focus on a simpler and more efficient alternative that has been shown to perform well in practice [58], which is the backtracking line search.

The backtracking line search procedure starts with an initial estimate of the step size α and tests whether the Armijo condition,

$$C(Y + \alpha \boldsymbol{p}) < C(Y) + c\alpha \boldsymbol{p}^T \nabla C(Y) \tag{3.5}$$

is satisfied. This condition ensures that a sufficient decrease in the cost function value is achieved. If the inequality does not hold, the backtracking line search will update the estimate of α, shrinking it by a factor of τ, and try again. The values of c, in the Armijo condition, and τ must satisfy $c < \tau$ and $c \in (0,1)$.

A note on the optimization algorithm for tSNE. Currently, the most popular method for optimizing tSNE is Gradient Descent (GD), which is an instance of the LS framework with some adjustments. First, it does not use Hessian information at all, which means that $B = I$ in Equation 3.3 and implies that it follows the direction opposite to the gradient ($\boldsymbol{p} = -I\nabla C$). Second, it avoids using a line search to find the optimal α, and instead a user-defined step size is dynamically updated throughout the optimization using momentum and an adaptive scheme [20]. Each iteration of this algorithm is significantly faster than a quasi-Newton iteration, mainly because there is no need to solve a linear system. Nevertheless, because it does not use Hessian information, it can take GD up to an order of magnitude more iterations to converge.

3.1.2. Trust Region

Instead of computing the direction and the step size separately, the Trust Region (TR) framework attempts to directly find a direction $\boldsymbol{p}$ that is properly scaled. To do this, a surrogate function m, which is simpler than the original cost function, is minimized within an area which is referred to as the trust region, hence the framework's name. Concretely, in TR the goal at each iteration is to solve the trust region subproblem, which is a constrained optimization problem of the form,

$$\underset{\boldsymbol{p}}{\operatorname{argmin}}\, m(\boldsymbol{p}) \text{ s.t. } ||\boldsymbol{p}||_2 \leq \Delta = \underset{\boldsymbol{p}}{\operatorname{argmin}}\, C(Y) + \nabla C(Y)^T \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T B \boldsymbol{p} \text{ s.t. } ||\boldsymbol{p}||_2 \leq \Delta \tag{3.6}$$

where m is the surrogate function, which is based on the Taylor expansion that was presented before; Δ corresponds to the trust region radius; and, as was the case with LS, B is an approximation of the true Hessian $\nabla^2 C(Y)$, which is often too expensive to compute. An interesting aspect of the TR framework is that we can use different norms such as the 1-norm and ∞-norm in the constraint. In this work, we focus on the widely used 2-norm, which defines a ball around the iterate as can be observed in Figure 3.1.

There are different ways to solve the optimization problem in Equation 3.6. Most of them are based on the Cauchy point, which we can obtain by following the direction opposite to the gradient until we reach the border of the TR. This point is important because it is a lower bound on the amount of cost function reduction that the algorithm can obtain at a given iteration. In general, available methods use the Cauchy point as a fallback if they cannot find a better alternative within the TR. The conjugate gradients method that is used to solve LS's linear system can be adapted to find a suitable $\boldsymbol{p}$. In addition to the checks that ensure that the Hessian matrix is positive definite and that the tolerance has not been reached, we define a set of conditions that guarantee that the resulting direction's length $||\boldsymbol{p}||_2$ is less than the radius Δ. This is precisely what was done in [47] and what we use for our experiments. We refer the reader to this work for an in-depth explanation of how this modified CG algorithm works and its theoretical guarantees.

Having a direction $\boldsymbol{p}$, we can measure its quality ρ with the ratio,

$$\rho = \frac{C(Y) - C(Y + \boldsymbol{p})}{m(0) - m(\boldsymbol{p})} \tag{3.7}$$

where $m(0) = C(Y)$. Given that m is an approximation of the true function, ρ indicates how close the predicted decrease in the cost function is to the observed one. This ratio is used to decide whether the direction

should be accepted or rejected and whether the radius should be updated. For the former, we accept $\boldsymbol{p}$ if $\rho < \eta_1$. For the latter, we use the conditions presented in Equation 3.8 to determine the radius of the next iteration Δ_{k+1}. Note that $0 \leq \eta_1 \leq \eta_2$ and $0 \leq \eta_3 \leq 1$ and that a maximum radius $\Delta_{\max}$ must be set.

$$\Delta_{k+1} \begin{cases} \eta_2 \Delta_k \text{ if } \rho < \eta_2 \\ \min(2\Delta_k, \Delta_{\max}) \text{ if } \rho > \eta_3 \text{ and } ||\boldsymbol{p}||_2 = \Delta \\ \Delta_k \text{ if in any other case} \end{cases} \tag{3.8}$$

To conclude, although both LS and TR depart from the same theoretical base, the Taylor expansion of C, their algorithmic structure is quite different. TR has been regarded as a more robust alternative given that it produces properly scaled directions, nevertheless, the subproblem that arises can be hard to solve. In contrast, LS is simpler to implement as exact of the shelf solvers can be used and good enough step sizes can be found with simple iterative procedures. This shows that, in general, there is no better method for optimizing nonconvex functions. For this reason, in this work we focus on both approaches, measuring their performance at optimizing tSNE.

3.2. The Kullback-Leibler Divergence and its Derivatives

To find a suitable embedding Y^* for the high-dimensional data, tSNE encodes the pairwise distances between high and low-dimensional points as affinity matrices P and Q and then compares them using the Kullback-Leibler (KL) divergence. The latter, also known as relative cross-entropy, measures the difference between two probability distributions using Equation 3.9. In the previous section, we showed how, besides the cost function C, quasi-Newton methods need to have access to its gradient $\nabla C(Y)$ and Hessian $\nabla^2 C(Y)$. We now go over these in more detail, describing their properties. We also present important building blocks such as the computation of the matrices of affinities P and Q and the graph Laplacian L.

$$KL(P||Q) = C(Y) = \sum_i \sum_j p_{ij} \log(\frac{p_{ij}}{q_{ij}}) \tag{3.9}$$

3.2.1. The Affinity Matrices

As can be observed in Equation 3.9, KL compares the discrete probability distributions P and Q, which are computed from the high and low-dimensional points X and Y, respectively. The starting point in both cases is the matrix of Euclidean pairwise distances, which for two points z_i and z_j can be computed using,

$$d(z_i, z_j) = ||z_i - z_j|| = (\sum_n (z_i^n - z_j^n)^2)^{1/2} \tag{3.10}$$

where n corresponds to a specific coordinate. To compute the high-dimensional affinity matrix P, the first step consists of transforming the distances $d(x_i, x_j)$ into conditional affinities, using the Gaussian kernel,

$$\text{gaussian}(d(x_i, x_j)) = g_{ij} = \exp(-d(x_i, x_j)^2 / 2\sigma_i^2) \tag{3.11}$$

where x_i and x_j are points in the high-dimensional space. The extent of the kernel is determined by the bandwidth σ_i, which is set individually for each point, taking into account the user-defined perplexity value. The latter can be regarded as the effective number of neighbors that are taken into consideration. The bandwidth can be found using a bisection search procedure [54] or root-finding algorithms [59] and we direct the reader to these references for more details. The next step consists in normalizing each row of G with,

$$p_{j|i} = \frac{g_{ij}}{\sum_{k \neq i} g_{ik}} \tag{3.12}$$

which results in a conditional matrix of affinities. Note that because of the adaptive σ_i, this matrix is asymmetric, which can make the optimization process difficult. Therefore, the last step in tSNE is to symmetrize and further normalize these conditional affinities using,

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N} \tag{3.13}$$

which yields the matrix of joint probabilities, or of affinities, P.

In a similar fashion, in the low-dimensional space the input is the matrix of pairwise euclidean distances $d(y_i, y_j)$ for $\{y_i, y_j\} \in Y$. The main difference in this case is the choice of kernel used to compute the affinities.

To deal with the crowding problem [54], we use the Student's t-Distributed Kernel with one degree of freedom which is defined as,

$$\text{t-Student}(d(y_i, y_j)) = t_{ij} = (1 + d(y_i, y_j)^2)^{-1} \tag{3.14}$$

In contrast with P, in this case the resulting matrix is already symmetric and therefore to obtain Q we only need to symmetrize it using,

$$q_{ij} = \frac{t_{ij}}{Z} \tag{3.15}$$

where Z is the normalization term corresponding to the sum all of the affinities,

$$Z = \sum_{k \neq l} t_{kl} = \sum_{k \neq l} (1 + d(y_k - y_l)^2)^{-1} \tag{3.16}$$

Affinity matrices as graphs. We have shown how P and Q correspond to matrices of affinities between each pair of high and low-dimensional points, respectively. It is useful to look at these matrices as adjacency matrices of a graph in which each point is a vertex, and the affinity value between two of these is an edge. Although in theory, these are fully connected graphs, because for every edge we assign an affinity, as we will see in the next chapter, in P most of the entries are close to zero and therefore it can be seen as the u-Nearest Neighbor graph, where u is set taking into account the perplexity.

3.2.2. The Graph Laplacian

The Graph Laplacian L is a diagonally dominant square matrix that results from subtracting a weights matrix W from a matrix of degrees D. D is diagonal and can be computed by adding the elements of the rows of W, which for an element d_{ii}, corresponds to,

$$d_{ii} = \sum_j w_{ij} \tag{3.17}$$

With this, the Laplacian matrix follows,

$$L = D - W \tag{3.18}$$

An important property of L is that, when the elements of D are positive, the matrix is guaranteed to be positive semidefinite. As we will see later, the Hessian of tSNE's Kullback-Leibler divergence can be written as a sum of Laplacians. This will facilitate the construction of Hessian approximations that work well with Cholesky and CG solvers used in the LS and TR frameworks.

3.2.3. The Gradient

The Kullback-Leibler divergence, presented in Equation 3.9 only depends on the embedding Y, considering that the high-dimensional matrix of affinities P is computed at the beginning and never changes. Therefore, the i^{th} entry of the gradient $\nabla C(Y) \in \mathbb{R}^{N \times d}$ of KL can be computed by taking the partial derivative of KL with respect to y_i which results in,

$$\frac{\partial C(Y)}{\partial y_i^{(c)}} = 4 \sum_{j \neq i} (p_{ij} - q_{ij}) q_{ij} Z (y_i^{(c)} - y_j^{(c)}) \tag{3.19}$$

where Z is the normalization term presented in Equation 3.16. For the curious reader, an in-depth derivation of the gradient is presented in Appendix A.

Given that for each entry of the gradient N operations need to be carried out, its computation does not scale well with the size of the data. In this work, we use the Barnes–Hut algorithm to approximate it, reducing its complexity. To understand how BH operates, it is useful to look at the gradient as a set of attractive and repulsive forces. By slightly rearranging the terms of Equation 3.19, we obtain,

$$\frac{\partial C(Y)}{\partial y_i^{(c)}} = F_{attr} + F_{rep} = 4 \sum_{j \neq i} p_{ij} q_{ij} Z (y_i^{(c)} - y_j^{(c)}) - 4 \sum_{j \neq i} q_{ij}^2 Z (y_i^{(c)} - y_j^{(c)}) \tag{3.20}$$

where the gradient has been split into two terms. The first corresponds to the set of attractive forces that ensures that similar points in the high-dimensional space are close by in the embedding. And the second corresponds to the set of repulsive forces that involve all points and ensure that clusters in the embedding have white space between them. A visual representation of this decomposition was presented in the previous chapter in Figure 2.3.

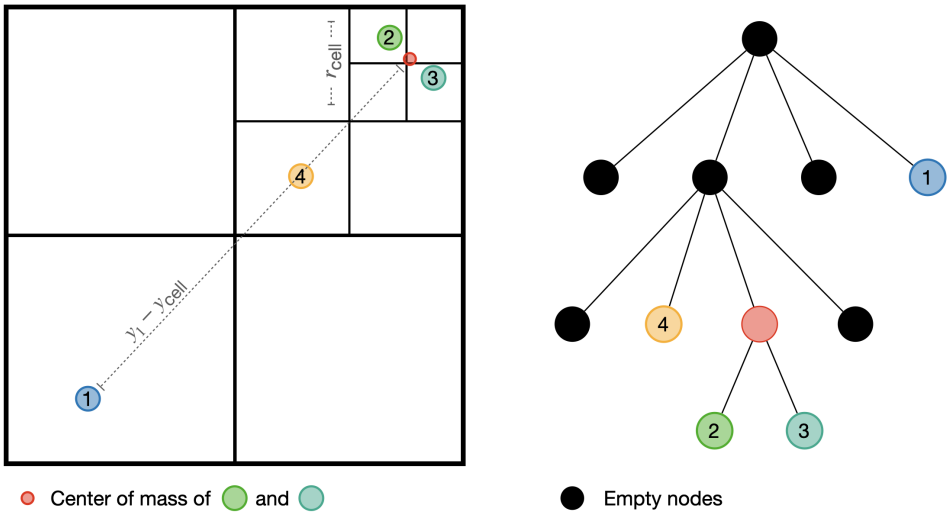

Figure 3.2: Using a quadtree to reduce the number of pairwise comparisons that are needed to compute the gradient of the Kullback-Leibler divergence. The left panel shows how the quadtree recursively partitions the embedding space until there is only one point per cell. The result of this procedure is the tree that the panel in the right of the figure presents. This tree can be used to speed up the computation of the gradient by avoiding a subset of the $O(N^2)$ comparisons. In the figure, we use a cell, as opposed to the individual points that it contains, if the ratio between its width and the distance of its center of mass to the target point (y_1 in this case), is below a threshold θ.

As was mentioned before, P can be approximated by a uNN graph, and therefore the attractive term can be efficiently computed, taking advantage of the sparsity of this graph. In contrast, the repulsive term requires the comparison of all pairs of points in Y, which scales quadratically with N. BH reduces the number of comparisons that are needed by indexing the low-dimensional points in a quadtree (or octree if in 3D) as presented in Figure 3.2. In this hierarchical structure, the points are located at the leaves and intermediate nodes contain aggregate information of their children such as the center of mass. In the example in the figure, suppose that we are interested in finding the first entry of the gradient. In this case, we would need to compute the distances between y_1 and y_2, and y_1 and y_3. Notice that y_2 and y_3 are close to each other and far away from y_1. While traversing the tree, BH will compare the width r_{cell} of the node that contains y_2 and y_3, with the distance of their center of mass y_{cell} to y_1, using Equation 3.21. If the value of the ratio is smaller than a user-defined approximation threshold θ, the cell will be used as a proxy for the points that it contains, saving, in this case, the need for an extra computation. A larger value of θ implies a coarser and faster approximation, while $\theta = 0$ corresponds to the exact gradient computation. In practice, it has been shown how BH is robust to changes in the value of this parameter, which is often set to $\theta = 0.5$.

$$\frac{r_{cell}}{||y_i - y_{cell}||^2} < \theta \tag{3.21}$$

Approximating the Kullback-Leibler divergence with BH. In the LS and TR frameworks, we need to evaluate the cost function value at least once per iteration. Given that its complexity is $O(N^2)$, this makes these methods impractical for larger datasets. Fortunately, we can also use BH to approximate it. Concretely, the bottleneck in the computation of the Kullback-Leibler divergence is the normalization term Z that is obtained by rearranging its terms following,

$$KL(P||Q) = C(Y) = \sum_i \sum_j p_{ij} \log(\frac{p_{ij}}{q_{ij}}) = \sum_i \sum_j p_{ij} (\log(p_{ij}) - \log(t_{ij}) + \log(Z)) \tag{3.22}$$

where t_{it} corresponds to the Student's t-distributed kernel. Besides this, all operations depend on the number

of non-zeros of P and therefore are efficient. To fix this, we just need to use the approximated normalization term Z_{approx} that is obtained during the computation of the gradient using BH.

3.2.4. The Hessian

The LS and TR frameworks presented at the beginning of this chapter use Hessian information to obtain better descent directions. In this subsection, we present the Hessian of tSNE and several of the approximations that have performed well in practice. Previously, we showed how the gradient of KL is a vector of Nd entries that collects the partial derivatives of the cost function with respect to each coordinate of every point. To compute the Hessian, we compute the derivative of each entry of the gradient with respect to every other entry, resulting in a square symmetric matrix of shape $Nd \times Nd$. Because of the number of computations involved, we do not present the full derivation of the Hessian here and instead direct the reader to Appendix B, where this is done. Furthermore, in this work, instead of using the algebraic derivation for each entry of the matrix, we will use the equivalent Laplacian-based expression of the whole matrix [58],

$$\nabla^2 C = 4L \otimes I_d + 8L^{yy} - 16((L^q \otimes I_d)Y)^T((L^q \otimes I_d)Y) \tag{3.23}$$

with weights,

$$\begin{aligned} w_{ij} &= (p_{ij} - q_{ij})t_{ij} \\ w^q_{ij} &= -q_{ij}t^2_{ij} \\ w^{yy}_{ni,mj} &= -(p_{ij} - 2q_{ij})(y^{(n)}_i - y^{(n)}_j)(y^{(m)}_i - y^{(m)}_j)t^2_{ij} \end{aligned} \tag{3.24}$$

As was already mentioned, this way of representing the Hessian is particularly useful for finding ways of including second-order information in the optimization procedures. The "approximate" Hessian that we are interested in must be simple and efficient to compute, as close as possible to the true Hessian $\nabla^2 C(Y)$, and compatible with linear solvers such as Cholesky decomposition and conjugate gradients (CG), which require positive definite matrices. In this work, we focus on the approximations proposed in [58], which are: the fixed-point iteration (FP), the Spectral Direction (SD), and the spectral direction with extra low-dimensional information (SDLD). Starting with the closely related FP and SD, the latter corresponds to the positive Laplacian SD= L^+, which has weights,

$$w^+_{ij} = p_{ij}t_{ij} \tag{3.25}$$

where p_{ij} is a constant term and t_{ij} varies at each iteration. For computational efficiency, the latter is often assumed to be equal to one [58]. Nevertheless, in this work, we show that not only this term can be added without significant overhead, but also that it can improve the performance of SD. Note that this matrix can be positive semidefinite. To be able to use it with the Cholesky factorization and CG, which require positive definiteness, we can add a multiple of the identity matrix βI, where $\beta = 1^{-10}\min(D^+)$. As for FP, it can be derived from a fixed point iteration as shown in [8], and ends up being equivalent to using the diagonal of SD, D^+.

The final approximation, SDLD, adds more positive semidefinite terms of the Hessian. Concretely, it corresponds to the Laplacian,

$$\text{SDLD} = 4L^+ \otimes I_d + 8L^{yy}_{n=m} \tag{3.26}$$

where only the terms of L^{yy} that share coordinate ($n = m$) are used. Because the weight matrix of L^{yy} has a term that does not depend on P, it is not sparse, and therefore the resulting Laplacian is dense. Additionally, its positive semidefiniteness is not as easy to fix as SD's, and therefore to ensure that it is stable, we use it only with iterative linear solvers such as CG, which include safeguards for these cases.

In this chapter, we presented the LS and TR frameworks and gave a detailed description of the different components of tSNE such as its cost function and its derivatives. In the next chapter, we make this theory concrete by introducing an experimental framework that implements tSNE and different optimization methods. There, we will discuss in-depth technical aspects such as the representation of the sparse matrices and the computational complexity of the different algorithms.

4

Methodology and Implementation

In the following two chapters we present the results of the different experiments that were conducted. In these, we compared versions of tSNE that used different solvers. To make the results directly comparable we build a complete experimental pipeline in Python with several useful components such as data loaders, efficient implementations of tSNE-related routines such as the computation of the cost function and its derivatives, and a generic optimization loop that enabled different schemes and had logging capabilities of all the relevant metrics. In this chapter, we present the most relevant details of this system. We start in Section 4.1, describing the different components of the experimental pipeline and then in Section 4.2 introduce the datasets that were considered in the experiments.

4.1. Experimental Pipeline

For our experiments, we implemented a version of the NLE pipeline tailored specifically for tSNE and for being able to handle multiple optimization schemes. Figure 4.1 presents a schematic of its computational flow. The pipeline has two main components which are the initialization and the iterative optimization of the embedding. In the former, we initialize the embedding and compute the matrix of high-dimensional affinities P. In the latter, the optimization loop takes place, which can be divided into two general steps. First, we compute the within iteration quantities, which depending on the optimization algorithm include the Hessian approximation, the gradient, and the next iterate. Then, we verify whether the algorithm has converged to decide if the loop must be stopped. In the following subsections, we will delve deeper into the individual components to clarify important algorithmic and implementation details.

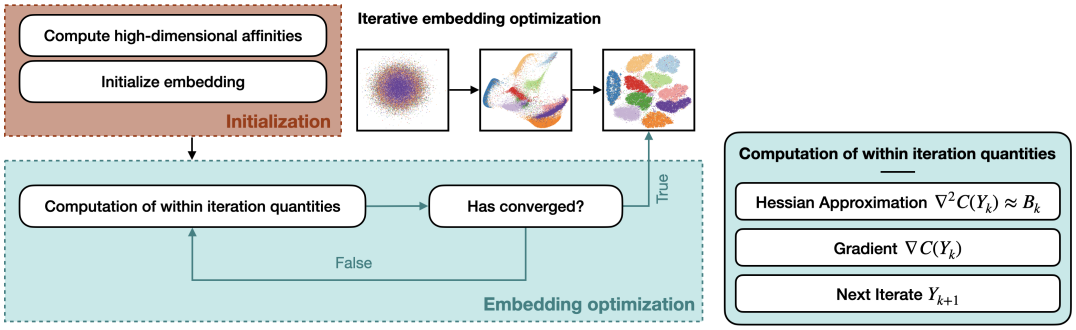

Figure 4.1: The overall structure of the computational pipeline that we employed in our experiments (left) and an in-depth view of the "computation of within iteration quantities" box (right). Note the algorithm will iterate until a termination condition is satisfied, producing a sequence of iterates like the one that can be observed in the top right section of the left plot.

4.1.1. Initialization of the Embedding

In iterative optimization, the starting point Y_0 must be chosen with care. In tSNE, there are two popular alternatives that we support. The first one is random initialization, in which we sample the value of every

parameter of the embedding using a Gaussian $\mathcal{N}(\mu, \sigma^2)$, where $\mu = 0$ and $\sigma = 1 \times 10^{-4}$. To enable the reproducibility of our work, we allow the user to set a random seed which guarantees that the same sequence of random numbers will be drawn. The second option that we support is PCA initialization, for which we compute the PCA decomposition of the input data X, and pick first d columns to form the initial embedding. Given that this PCA embedding can be arbitrarily scaled, which can cause problems during the optimization, we rescale the points to fit in a centered ball of radius 1×10^{-4}. Taking into account that this initialization is deterministic, no seed is needed.

4.1.2. Computation of the High-Dimensional Matrix of Affinities

There are multiple ways to compute P which can be either exact or sparse, using the u-Nearest Neighbor (uNN) graph. In this work, we used scikit-learn's kd-tree-based nearest neighbor search to compute the approximate matrix of Euclidean distances in $O(DN\log N)$. Given that this algorithm scales poorly with the dimensionality of the input data D, if the latter is higher than 100, we used PCA to reduce its number of dimensions to this value. We determined the number of neighbors u to be considered taking into account the user defined perplexity $perp$ with $u = 3perp + 1$. And used the bisection based approach described in [54] to compute the bandwidths for each point. Finally, we symmetrize and normalize our matrix to obtain P. These operations can be performed efficiently due to the sparsity structure of P, which we visualize in Figure 4.2 for the Iris dataset.

4.1.3. Computation of the Gradient and Cost Function Value

In this work we consider both exact ($\theta = 0$) and approximate ($\theta > 0$) cost function and gradient evaluations. For the exact versions, keeping in mind that their complexity is $O(N^2)$, we tried distributing the computations across multiple threads using Numba [25] to reduce computation time. For the approximate versions, we used the Barnes-Hut algorithm which reduces the complexity to $O(N\log N)$. Concretely, we wrapped the function that the scikit-learn's [39] implementation of BH-SNE uses to compute the approximate gradient, which also has an option to return an estimate of the cost function value. The latter works by using the resulting Z_{approx} from the gradient computation in Equation 3.22.

4.1.4. Computation and Updating of the Hessian Approximation

Among the Hessian approximations B that we consider, FP and SD are the most efficient to compute given that the latter shares the sparsity pattern of P, and the former is just SD's diagonal. An important result of this work is that we show that for tSNE, periodically adding low-dimensional information to SD can be beneficial ($t_{ij} \neq 1$ in Equation 3.25). Naively implemented, this updating can become an important bottleneck because it suggests computing the matrix of pairwise distances for the low-dimensional points, which is $O(N^2)$. To avoid this, we implemented a sparse pairwise distance calculation routine that is shown in Algorithm 1. It uses the sparsity pattern of P to avoid computing entries of the distance matrix that will not contribute. Given that we store our sparse matrices in Compressed Storage Row (CSR) format, we further accelerated this procedure by distributing the computation of each row, which is assigned a chunk of contiguous memory, across processors. After computing the sparse matrix of low-dimensional affinities T, we can efficiently multiply it against P to obtain the weight matrix of the positive Laplacian, which corresponds to SD. The latter can be obtained efficiently using sparse linear algebra operations.

Algorithm 1 Sparse Pairwise Distance Matrix Computation

```
procedure SPARSEPDISTS(Y, P)        ▷ Embedding and sparse matrix of high-dimensional affinities
    T ∈ ℝ^{N×N}                     ▷ Output sparse matrix
    for r ← 1, N do                 ▷ Rows of P
        for c ∈ nonzero(P_r) do     ▷ Indices of nonzero entries in row r of P
            t_rc = ||y_r − y_c||    ▷ Points r and c of the embedding
        end for
    end for
    return T
end procedure
```

The computation of SDLD is less efficient given that the complete matrix of low-dimensional pairwise distance, which has N^2 entries, needs to be evaluated. We accelerate this by using different threads for differ-

ent rows of the matrix. Nevertheless, similarly to the exact computations of the gradient and the cost function value, it does not scale well to datasets with more than a couple thousand points. For this reason, we only consider it in the smaller scale experiments of Chapter 5. As has been suggested throughout this subsection, the efficiency of the optimization process will depend to a great extent on the sparsity of B, given that the fewer the number of non zero values, the faster the different operations will be. In Figure 4.2, we present the sparsity pattern of the different Hessian approximation, with FP being the sparsest of all and SDLD a dense matrix. It is important to note how even if FP and SD are initially matrices of shape $N \times N$. In practice, we transform all the Hessian approximations to matrices of shape $dN \times dN$ by applying the Kronecker product SD = SD $\otimes I_d$, where I_d is the d-dimensional identity matrix. Then, when we need to multiply them by a vector, for example, to solve $B\boldsymbol{p} = -\nabla C$, we flatten ∇C before carrying out such operation. When using CSR matrices, as is the case in this work, this sparse matrix-vector multiplication has a cost of $O(NNZ_B)$, where NNZ_B is the number of nonzero entries of the approximation B.

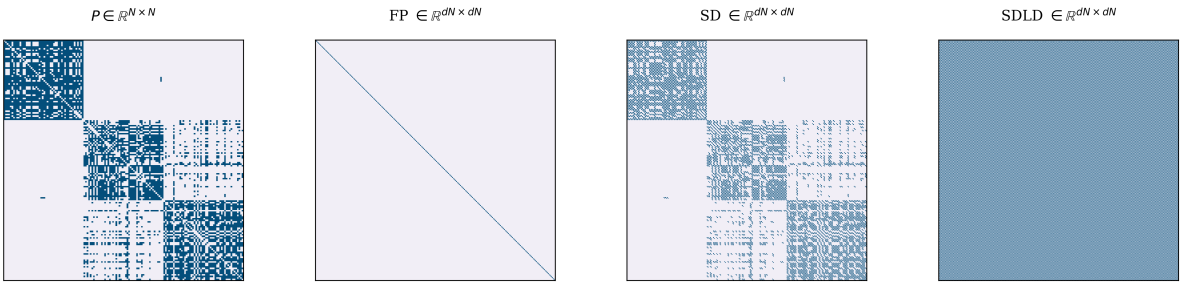

Figure 4.2: The sparsity patterns of the different matrices for the Iris dataset ($N = 150$ and $D = 4$) with target dimensionality $d = 2$. The darker shade of blue corresponds to entries of the matrices that are not zero. The left-most figure corresponds to the matrix of high-dimensional affinities P and the remaining three correspond to the different Hessian approximations FP, SD, and SDLD. Note how the latter three, are larger than P to be able to accommodate the coordinate-dependent terms of the Hessian matrix. Also note how, besides the diagonal, SD has the same sparsity structure of P, and how SDLD is the densest of all.

4.1.5. Finding the Next Iterate

Our experimentation pipeline implements three different approaches for finding the next iterate: gradient descent with momentum and adaptive step length (VDM), and line search (LS) and trust region (TR) quasi-Newton methods. In the following paragraphs we describe their implementations, and the relevant parameters that they accept.

VDM. This is the most popular optimization algorithm for tSNE. For its implementation and the values of the different parameters we followed [54]. In addition to the embedding and the gradient, the method requires two additional vectors that are used to implement momentum and the adaptive learning rate scheme. The first one, **incs**, is initialized to $\vec{1}$ and the second one, **gains**, to $\vec{0}$. In addition to these vectors, two scalar parameters are required: min-gain and momentum. For our experiments, we set the former to 0.01 and the latter to 0.5 for the first 250 iterations after which we switched it to 0.8. Finally, α corresponds to the step size or learning rate which we set to 200.

Algorithm 2 Computing Next Iterate With Momentum and Adaptive Learning Rate

1: **procedure** COMPUTENEXTITERATEVDM($Y_k, \nabla C(Y_k), \alpha$, **incs**, **gains**, min-gain, momentum)
2: $\boldsymbol{se} \leftarrow \text{sign}(\nabla C(Y_k)) = \text{sign}(\textbf{incs})$ ▷ Boolean vector (0s and 1s)
3: $\textbf{gains} \leftarrow (\vec{1} - \boldsymbol{se})(\textbf{gains} + 0.2) + \boldsymbol{se}(\textbf{gains} \times 0.8)$
4: $\textbf{gains}_i \leftarrow$ min-gain $\forall i \in \{0, ..., dN\}$ **if** $\textbf{gains}_i <$ min-gain
5: $\textbf{incs} \leftarrow \text{momentum} \times \textbf{incs} - \alpha(\textbf{gains} \times \nabla C(Y_k))$
6: $Y_{k+1} \leftarrow Y_k + \textbf{incs}$
7: **return** Y_{k+1}
8: **end procedure**

LS. As was explained in the previous chapter, the next iterate is found by determining a suitable direction $\boldsymbol{p}$ that is scaled with a step size α. The former is found by solving the linear system in line 2 of Algorithm 3. This can be done either directly or iteratively. For the former, we use a Python wrapper of CHOLMOD library to compute the Cholesky factors of B_k which can be then cached and used at each subsequent iteration (if

the matrix does not change) to solve the system using forward and backward substitution. For the latter, we implemented the CG algorithm presented in [36]. Important parameters of CG are the preconditioner, which unless stated otherwise is the identity matrix I; the maximum number of iterations K_{CG}, which we set to 100 by default; and the residual tolerance which is set dynamically, taking into consideration the current gradient to $\epsilon_{CG} = \min(0.5, ||\nabla C(Y_k)||^{0.5})||\nabla C(Y_k)||$. Having the direction, we find the step size α_k using backtracking line search. At each iteration, the Armijo condition in Equation 3.5 is tested to decide whether to stop backtracking should or to try a new step size $\alpha_k = \tau\alpha_k$. In our experiments we set $\tau = 0.8$ and $c = 0.1$. Finally, the backtracking line search receives an initial step size α from which the search is started, we followed the same conservative strategy of [58], where α_{k-1} was used as a starting point to find α_k.

Algorithm 3 Computing Next Iterate Quasi-Newton with Line Search

1: **procedure** COMPUTENEXTITERATELS($Y_k, C(Y_k), \nabla C(Y_k), B_k, \alpha_k$)
2: **Solve** $B_k \boldsymbol{p}_k = -\nabla C_k$ ▷ Can be solved with Cholesky or CG
3: $\alpha_k, C(Y_{k+1}) \leftarrow$ backtrackingLinesearch$(Y_k, \boldsymbol{p_k}, C(Y_k), \nabla C(Y_k), \alpha_k)$ ▷ With Armijo condition (Eq. 3.5)
4: $Y_{k+1} \leftarrow Y_k + \alpha_k \boldsymbol{p}_k$
5: **return** $(\alpha_k, Y_{k+1}, C(Y_{k+1}))$
6: **end procedure**

TR. In this case, the goal is to solve the subproblem presented in line 2 of Algorithm 4. We do this using a preconditioned CG procedure that includes safeguards for the trust region constraint [47]. Similarly to LS, we use the identity as preconditioner, $K_{CG} = 100$, and $\epsilon_{CG} = \min(0.5, ||\nabla C(Y_k)||^{0.5})||\nabla C(Y_k)||$. Having the tentative next iterate within the trust region, Y'_{k+1}, the next step consists of validating its quality, which is done using Equation 3.7 presented in the previous chapter. Finally, having ρ, the radius is updated and the new iterate is either accepted or rejected depending on its quality. For the former, Equation 3.8 is used with the parameter values recommended in [36]: $\eta_1 = 0.8$, $\eta_2 = 0.25$ and $\eta_3 = 0.75$.

Algorithm 4 Computing Next Iterate Quasi-Newton with Trust Region

1: **procedure** COMPUTENEXTITERATETRUSTREGION($Y_k, C(Y_k), \nabla C(Y_k), B_k, \Delta_k, \Delta_{\max}$)
2: **Solve** $\min m(\boldsymbol{p}_k) = C(Y_k) + \nabla C(Y_k)^T \boldsymbol{p}_k + 0.5 \boldsymbol{p}_k^T B_k, \boldsymbol{p}_k$ s.t. $||\boldsymbol{p}_k||_2 \leq \Delta_k$
3: $Y'_{k+1} = Y_k + \boldsymbol{p}_k$ ▷ Tentative new iterate
4: $\rho \leftarrow$ checkSolutionQuality$(C(Y_k), C(Y'_{k+1}), m(\boldsymbol{p}_k))$ ▷ Eq. 3.7
5: $\Delta_{k+1} \leftarrow$ updateRadius$(\rho, \Delta_k, \Delta_{\max})$ ▷ Eq. 3.8
6: **if** $\rho > \eta$ **then** ▷ Do not accept Y'_{k+1} otherwise
7: $Y_{k+1} \leftarrow Y'_{k+1}$
8: **end if**
9: **return** $(\Delta_{k+1}, Y_{k+1}, C(Y_{k+1}))$
10: **end procedure**

4.1.6. Optimization Loop Stopping Criteria and Logging

To finalize this section we cover the aspects of stopping criteria and logging. In the case of the former, we included different ways to terminate the optimization loop that ensure that separate runs are directly comparable. Concretely, the optimization is stopped when a number of iterations K have taken place, a given time limit has been reached or two consecutive iterates are below a given threshold according to the expression $\epsilon_Y < (||Y_k - Y_{k+1}||_\infty)/(1 + ||Y_{k+1}||_\infty)$. It is important to note how we do not include stopping criteria based on the cost function given that not all the compared methods compute it and doing so would add a significant overhead. On the other hand, regarding the logging. In general, for all the runs we record the cost function value, the partial embedding, and the elapsed time. In the case of the former, if it was not needed during the optimization we compute the cost function values offline to avoid affecting the performance of the methods. Finally, for the quasi-Newton methods, we also record relevant information such as the step sizes, the trust region radii, the number of CG iterations, the CG reached tolerance, and keep track of potential errors such as using Cholesky decomposition with an indefinite matrix. The usefulness of these data will become evident once we show the experimental results in the following chapters.

4.2. Datasets

An important part of our experimental pipeline is the datasets that we use to compare the optimization algorithms in different scenarios. We surveyed the literature and selected six datasets that have been used to measure both the quality of the resulting embeddings and the efficiency of different algorithms. In the experiments, depending on the setting, we will not be using all the datasets. To be able to refer to them consistently across the document we divided them into three groups according to their size. Given that we focus on how our methods to scale as a function of the number of points N and not the number of dimensions D, in all cases, but Digits, we used PCA to reduce the dimensionality of the dataset to $D = 100$. We briefly present the datasets below.

- **Coil-20 (Small):** the Coil-20 database [35] contains 1440 128×128 gray-scale images of 20 different objects. To generate the different images, each object was placed on top of a turntable and then rotated in five-degree intervals. Therefore, there are 72 different images associated with each object. To generate the input dataset X, the 2-dimensional array of pixel values associated with each image was flattened and stacked, yielding a dataset with $N = 1440$ and $D = 16384$.

- **Digits (Small):** the Digits database [62] is a collection of 8×8 gray-scale images of handwritten digits. In this work, we only use the test set which has consists of $N = 1797$ images and we flatten each one of these to obtain a vector of $D = 64$ features. In this case, because $D < 100$, we did not use PCA in the preprocessing.

- **Coil-100 (Small):** the Coil-100 database [34] contains 7200 128×128 colored images (3 channels) of 100 different objects. This is a superset of the Coil-20 dataset and therefore shares the collection process. First, we convert each image to grayscale using the ITU-R 601-2 luma transform and then we flatten them, which yields 16384-dimensional vectors. The resulting dataset has shape $N = 7200$ and $D = 16384$.

- **PenDigits (Small):** the PenDigits dataset [2] contains the x, y coordinate information of 10992 handwritten digits. The coordinate information was obtained by sampling 8 points along the trajectory of each digit, yielding 8×2 features. Therefore, the PenDigits dataset has shape $N = 10992$ and $D = 16$, and, given that $D < 100$, we did not use PCA in the preprocessing.

- **MNIST (Medium):** similarly to Digits, the MNIST database [26] is a collection of gray-scale images of handwritten digits. In this case, the images have a size of 28×28 pixels. It has a training and test sets with 60000 and 10000, respectively, which are used in supervised learning scenarios. Here, we merge both and consider them as a unique dataset with $N = 70000$ and $D = 784$.

- **F-MNIST (Medium):** the Fashion-MNIST dataset [61] follows the same structure of MNIST, but instead of using digits, it contains images of ten different classes of fashion items. As with MNIST, we joined the training and test sets into a unique dataset with $N = 70000$ and $D = 784$.

- **ImageNet (Large):** ImageNet is a database consisting of millions of images that are often used to benchmark deep learning applications [12]. In this work, we used the activations of different layers of a Deep Neural Network (DNN) trained on ImageNet's validation set, which has 100000 images. The DNN that was used in this case is the Google Inception [50]. We considered the two layers used in [43], which are: **Mixed3a** and **Head0**. Each of these constitutes a separate dataset. While the former has $D = 256$, the latter has $D = 128$.

5

Comparative Study of Techniques for Optimizing tSNE Cost Function

The goal of the experiments that we present in this chapter is to replicate the results obtained in [58], where the Spectral Direction (SD), combined with the line search (LS) framework, was shown to outperform several alternative methods for optimizing tSNE. We extend their analysis in two ways. First, we try different variations of the SD that we tailored for tSNE by adding iteration-dependent information. Second, we consider the trust region (TR) optimization framework, which so far had been neglected in the context of quasi-Newton optimization methods for tSNE. Having said this, we start in Section 5.1 by exploring the effect of two of the most important parameters of LS and TR, which are the initial step size α_0 and trust region radius Δ_0, respectively. Then, in Section 5.2 we proceed to determine which is the best performing Hessian approximation for tSNE. After that, in Section 5.3, we compare the performance of resulting methods, SD-LS and SD-TR, with other popular optimization techniques. Table 5.1 presents a summary of the experiments that we performed, all of which we ran on a MacBook Pro with 16GB of RAM and a Quad-Core Intel Core i5 clocked at 2.3 GHz. Finally, it is important to consider that, given that in this chapter we have not yet addressed the bottlenecks of our methods, we were still restricted to the smaller datasets, Coil-20, Coil-100, Digits and PenDigits, for the experiments.

Experiment	Page	Description
5.1	27	Exploring the effect of α_0 and Δ_0 on LS and TR, respectively.
5.2	30	Adding low-dimensional information to the Hessian approximations.
5.3	31	Comparison of different Hessian Approximations for LS and TR.
5.4	33	Comparative study of different optimization techniques for tSNE.

Table 5.1: Experiments covered in this chapter.

5.1. Parameter Choice in Quasi-Newton Methods

In LS and TR, the step size and trust region radius have the important role of determining the scaling of the directions that will lead to the following iterate. A bad selection of these values will lead to optimization procedures that take longer to converge or fall trap to local optima. In preliminary experiments, we noticed that the performance of LS and TR depended significantly on the initial values α_0 and Δ_0, respectively. Therefore, in this section, we explore their effect and find values that perform well across datasets. Concretely, we investigate the following questions.

Experiment 5.1. *What is the effect that the initial step size α_0 and the initial trust region Δ_0 have on the convergence of LS and TR, respectively? Which are the best performing values for these parameters?*

Experimental setup. For the experiments in this section we focused on the constant Spectral Direction (SD) proposed in [58] ($t_{ij} = 1$ in Equation 3.25), used within the LS and TR frameworks. To keep the methods comparable, in both cases we used the conjugate gradient (CG) solver to find the new iterate. To keep

Dataset	$\|\|\boldsymbol{p}_{ls}\|\|$
Coil-20	300
Digits	300
Coil-100	10200
PenDigits	6200

Table 5.2: Baseline values to set the initial trust region radius Δ_0 for each dataset.

computation times low, we truncated the latter to $K_{CG} = 500$ iterations. For LS we tried different values of the initial step size $\alpha_0 \in \{1, 10, 100, 1000\}$. For TR, taking inspiration from [14, 46], we used an adaptive approach for Δ_0. For this, we selected a baseline value $\boldsymbol{p}_{ls}$ for each dataset by running 10 iterations of LS with $\alpha_0 = 1$, logging the norms of steps that were taken, and picking the largest one. This yielded Table 5.2. Having this, we tried different radii by scaling $||\boldsymbol{p}_{ls}||$ with a multiplier $m_{\Delta_0} \in \{0.01, 0.1, 1, 10\}$ so the resulting initial radius corresponds to $\Delta_0 = m_{\Delta_0}||\boldsymbol{p}_{ls}||$. Finally, we considered three stopping criteria to guarantee that enough progress had been made: more than $K = 500$ iterations had taken place, the elapsed time was greater than 5000 seconds, or the variation between iterates, as defined in Chapter 4, was less than 1×10^{-6}. It is important to note that the α or Δ can become 0 at some point in the optimization. As this means that no further progress can be made, we also stopped the iterations in this case.

Results. Starting with LS, in the upper left chart of Figure 5.1, it can be seen how increasing α_0 improved the convergence of the method. Nevertheless, the chart below shows how the largest values of α_0 caused the algorithm to take more time. Although we saw this effect in all the small datasets, it got more pronounced as their size grew. We found that the reason for this is that larger initial step sizes force more cost function evaluations in the backtracking procedure, which are expensive. As can be observed in Figure 5.2, LS only uses $\alpha > 1$ for a few iterations at the beginning, when the initial large drop in the cost function value is taking place. After that, it converges to a smaller value for the rest of the optimization. Keeping this in mind, if α_0 is set too large, the line search is going to spend too much time, in the beginning, trying to lower the step size to a suitable value.

Regarding TR, we found a similar effect. As can be observed in the right column of Figure 5.1, an initial larger radius was beneficial for the convergence of the algorithm. We found that smaller radii, caused the optimization to be more susceptible to local optima, which is reflected by the less step decrease in the cost function. When the radius is large enough, TR will follow a similar downward trajectory to LS. Nevertheless, in contrast with LS, larger initial values of Δ_0 did not result in slower runs since the latter does not rely on a backtracking procedure and therefore no extra cost function evaluations are going to be performed. Finally, as can be observed in Figure 5.2, the radius follows the same path as the step size. After a few iterations with a large value, it converges to lower ones for the rest of the optimization.

Conclusions. The performance of LS and TR depends to a large extent to the correct setting of α_0 and Δ_0, respectively. For the former, we found that setting it to a value slightly larger than 1 enabled the method to take the crucial initial large steps without significantly increasing the number of backtracking iterations that are required. Similarly, for TR, we found that a larger initial radius resulted in better convergence. Concretely, setting $\Delta_0 = m_{\Delta_0}||\boldsymbol{p}_{ls}||$ with $m_{\Delta_0} = 1$, we obtained consistently good results across datasets. A problematic aspect of this approach to finding Δ_0 is that it must be set ad hoc for each dataset, which might not be ideal for the end-user. Finally, an interesting aspect that we observed is the important role that SD plays in forming well-separated clusters in the final embedding, akin to the role of the early exaggeration phase in tSNE. This effect takes place in the first few iterations where large steps are taken. In the case of TR, we noticed that when we used a smaller radius, even if the method reached the same final cost function value, there was a high chance that it had converged to a local minimum, where some clusters were split.

The values of α_0 and Δ_0 can have a significant impact on the performance of LS and TR, respectively. We obtained the best performance using $\alpha_0 = 10$ and $\Delta_0 = ||\boldsymbol{p}_{ls}||$.

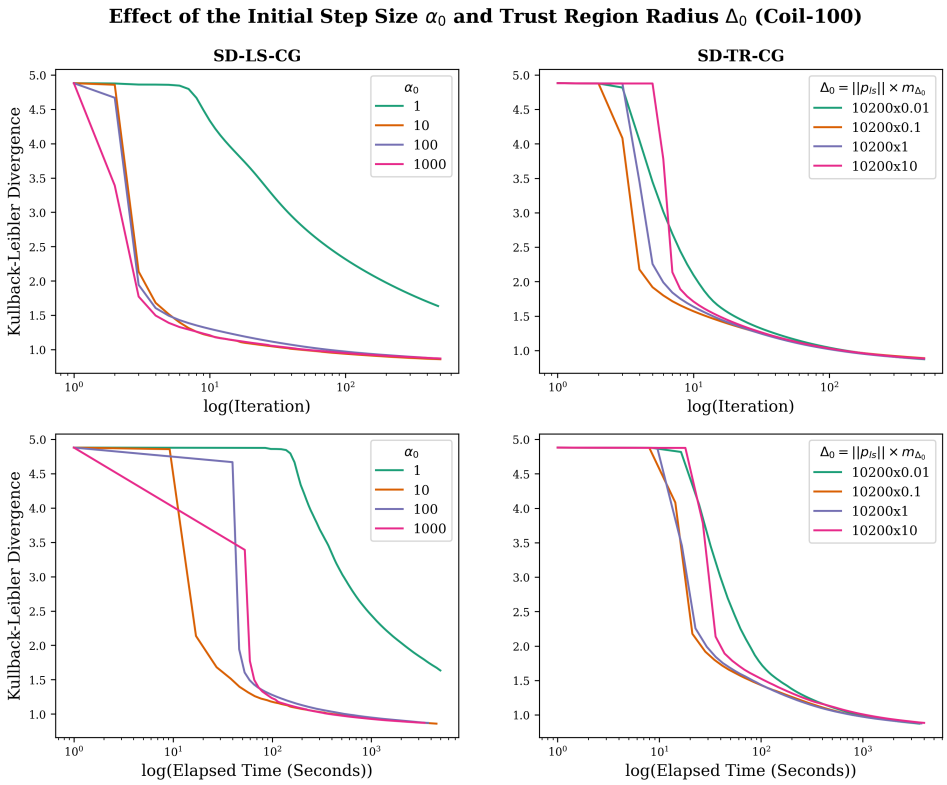

Figure 5.1: The behavior of the Kullback-Leibler divergence (KL) when embedding the Coil-100 dataset using SD-LS-CG (left) and SD-TR-CG (right) for different values of the initial step size α_0 and trust region radius Δ_0, respectively. The first row presents KL as a function of the iteration, and the second row, KL as a function of the elapsed time in seconds. A logarithmic scale was used for the x-axes to improve the chart's readability.

5.2. The Best Hessian Approximation

The effectiveness of quasi-Newton methods depends on how much information of the true Hessian $\nabla^2 C$ can be used without affecting the performance of the algorithm. In this work we consider the fixed-point iteration (FP), the Spectral Direction (SD), and the Spectral Direction with extra low-dimensional information SDLD, all of which we introduced in Chapter 3. In [58], the authors show how the constant SD ($t_{ij} = 1$ in Equation 3.25) was the best performing one across several NLEs such as the Elastic Embedding, symmetric SNE, and tSNE. We argue that in the case of the latter, SD can be made more effective, without sacrificing performance, by periodically updating it with low-dimensional information ($t_{ij} \neq 1$). We start this section by testing this hypothesis after which we compare their performance against FP and SD using both LS and TR. We used the same setup for the experiments of this section, which we detail below.

Experimental setup. We tried varying the rate at which SD was updated with low-dimensional information. Concretely, we denote the updating schedule Rx, where x is the number of iterations between updates. On one extreme we have $R0$, which corresponds to the constant SD, and on the other $R1$, which means that the SD is updated at every iteration. Given that Hessian-based methods converge in fewer iterations, we did not try values of x larger than 50 given that at this point the methods usually had converged and the effect of the update would be minimal. Specifically, we considered $x \in 0, 1, 10, 50$. As for the other approximations, given that they are not our focus, we used their "original" versions, which means that we computed FP based on SD with $R0$ and updated the SDLD at every iteration ($R1$). In this section, we do not report results using the Cholesky solver given that it is not compatible with SDLD, which is often indefinite. Therefore, to make the comparison fair, we considered only the conjugate gradients (CG) based LS and TR. We truncated the CG solvers at $K_{CG} = 500$ iterations to avoid long execution times per iteration. Regarding relevant parameters of LS and TR, we set $\alpha_0 = 10$ for the former and $\Delta_0^{\text{Coil-20}} = \Delta_0^{\text{Digits}} = 300$ for the latter. In the next chapter, we present an in-depth explanation of how we arrived at these values. Finally, we considered three stopping criteria to guarantee that enough progress had been made: more than $K = 500$ iterations had taken place, the elapsed time was greater than 5000 seconds, or the variation between iterates, as defined in Chapter 4, was

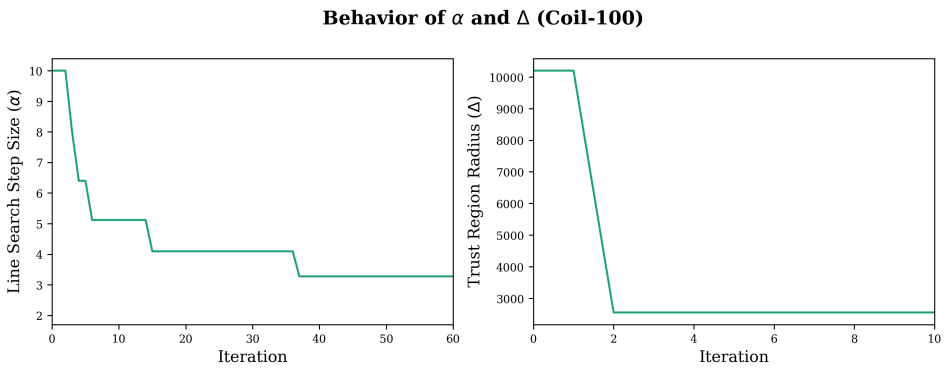

Figure 5.2: The behavior of α and Δ for the Coil-100 dataset with initial values $\alpha_0 = 10$ and $\Delta_0^{\text{Coil-100}} = 10200$. Note how in LS, α spends few iterations at relatively high values and eventually stabilizes to a lower one for the rest of the optimization. This behavior can also be observed in TR, where Δ significantly shrinks after a few iterations.

less than 1×10^{-6}. It is important to note that the α or Δ can become 0 at some point in the optimization. As this means that no further progress can be made, we also stopped the iterations in this case.

5.2.1. Adding Low-Dimensional Information to the Spectral Direction

In this subsection, we focus on determining the best update schedule Rx for SD-LS and SD-TR. Concretely, we seek to answer the following questions.

Experiment 5.2. *Does adding low-dimensional information improve the convergence of the SD-based quasi-Newton methods? Which is the best performing Rx for LS and TR?*

Results. Starting with LS, we observed an improvement in the convergence rate when periodically added low-dimensional information. In the top left part of Figure 5.3, it is possible to appreciate how $R10$ and $R50$ accelerate the convergence of the algorithm when the first update is performed. Interestingly, in the different datasets, we did not observe a significant advantage by using $R1$. On one hand, even if it reduces the number of iterations that are needed, it involves more updates, which slows down the algorithm. On the other hand, we noted how the low-dimensional information tended to be more useful to the algorithm once it had gotten closer to an optimizer as is the case with $R10$ and $R50$. In contrast, $R1$ includes information that, in the beginning, does not have much meaning given that the embedding was randomly initialized.

Moving to TR, we observed that with the smallest datasets, such as Coil-20 and Digits, adding low-dimensional information worked similarly to the LS case. Nevertheless, with the largest ones the effect was the opposite: adding low-dimensional information degraded the convergence. This can be observed in the right column of Figure 5.3 for the PenDigits dataset. For $R10$, it is possible to observe how after the 10^{th} iteration, the reduction in the cost function value flattens. To understand why this happened, we analyzed the residuals of the CG solvers for both LS and TR. We found that after updating SD with low-dimensional information, it took them more iterations to converge which is a sign of ill-conditioning. In preliminary experiments, we tried using the diagonal (Jacobi) and Cholesky preconditioners but did not obtain significant improvement.

Conclusions. The results suggest that adding low-dimensional information can improve the convergence of SD. Nevertheless, it also seems to produce an ill-conditioned matrix, which affects the performance of the CG solvers. We found that the LS and TR methods reacted differently to this issue. In the case of the former, truncating CG is important because, after the update of SD with low-dimensional information, LS will need significantly more iterations of CG to achieve the desired tolerance, making each iteration of the global optimization procedure more expensive. In the case of the latter, the combination of ill-conditioning and a small radius forces the CG procedure to end earlier, yielding iterates of poor quality. With this in mind, we observed that while LS can benefit from the iteration-dependent information, TR cannot, at least with the current parameters. We believe this can be fixed with a preconditioner that reduces the number of CG iterations that are needed and/or by lowering the acceptance threshold for new iterates. As for the former of these suggestions, we tried the diagonal and Cholesky preconditioners and did not find a significant improvement.

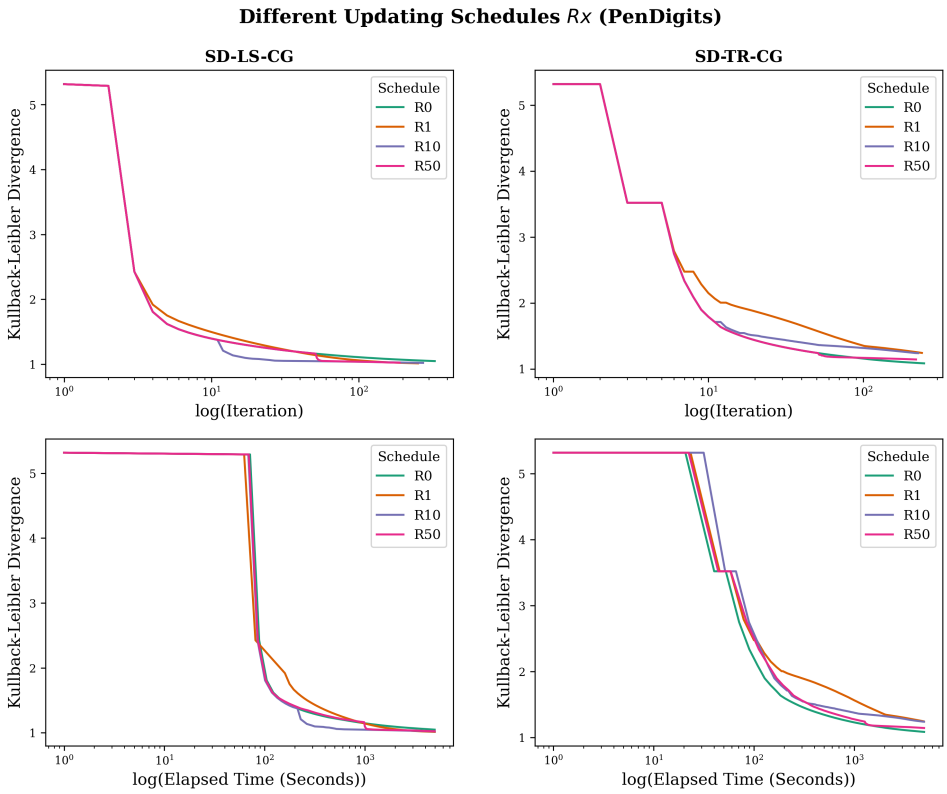

Figure 5.3: The behavior of the Kullback-Leibler divergence (KL) when embedding the PenDigits dataset using SD-LS-CG (left) and SD-TR-CG (right) for different update schedules Rx of the low-dimensional information in SD. The first row presents KL as a function of the iteration, and the second row, KL as a function of the elapsed time in seconds. A logarithm scale was used for the x-axes to improve the chart's readability.

Adding low-dimensional information to SD improves the convergence of LS, but not of TR. Therefore, we consider for the rest of the experiments LS with $R10$ and TR with $R0$.

5.2.2. Comparison of Hessian Approximations for tSNE

Having the best performing SD for LS and TR, we proceed to compare it with the other Hessian approximations. Particularly, we investigate the following questions.

Experiment 5.3. *Among FP ($R0$), SD ($R10$ for LS and $R0$ for TR), and SDLD ($R1$) which is the best performing Hessian approximation? Does this hold for the LS and TR frameworks?*

Results. In Figure 5.4, we present the comparison of the different approximations for the Coil-100 dataset. The first thing to notice is that in both, LS and TR, FP is the least effective approximation, taking significantly more iterations, and time, to reach the same cost function value. This shows how having fast iterations is not enough if the Hessian approximation does not generate substantially better search directions that reduce the number of iterations that are needed. As for SD and SDLD, it is possible to observe how the latter is slower due to the $O(N^2)$ cost of updating the matrix at each iteration. Additionally, the extra information that L^{yy} provides does not seem to significantly help the convergence of the method. This effect is more clearly observed in TR, where the SDLD line lags behind SD's for the whole optimization procedure both in terms of iterations that are needed and elapsed time. We also tried updating the low-dimensional information of SDLD periodically but found that it did not improve over SD and was still limited to smaller datasets due to its higher per-iteration complexity. We noted that these effects held across datasets and became more accentuated as their size grew.

Conclusions. The results obtained in this section indicate that SD is indeed the best performing Hessian approximation of those that were considered in [58]. For both LS and TR, SD can reduce the cost function

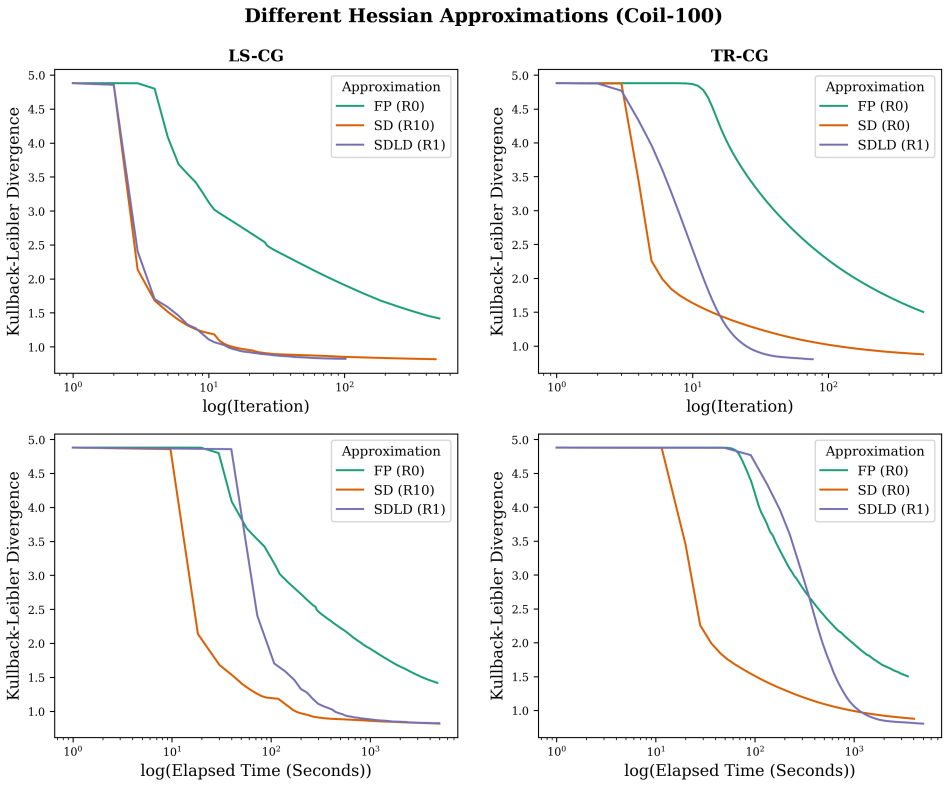

Figure 5.4: The behavior of the Kullback-Leibler divergence (KL) when embedding the PenDigits dataset using LS-CG (left) and TR-CG (right) with different Hessian approximations: FP ($R0$), SD ($R10$ for LS and $R0$ for TR), and SDLD ($R1$). The first row presents KL as a function of the iteration, and the second row, KL as a function of the elapsed time in seconds. A logarithm scale was used for the x-axes to improve the chart's readability.

value in fewer iterations than SDLD and FP, while being computationally efficient due to its sparsity. Furthermore, we observed that, in the LS case, it can be further improved by adding low-dimensional information periodically, and found that doing so every ten iterations is the best tradeoff of speed and decrease of the cost function value. Although approximations that include even more information such as SDLD are similarly effective to SD in terms of convergence rate, they are limited by their computational complexity which scales quadratically with the number of data points.

SD and SDLD have a similar effect on the reduction of the cost function value and perform better than FP. Nevertheless, each iteration of SD is significantly faster because of its sparsity. Therefore, SD is the best performing Hessian approximation for both LS ($R10$) and TR ($R0$).

5.3. Comparative Study of Optimization Techniques

In this section, we compare the performance of SD-LS and SD-TR with other optimization algorithms that have been tried with different NLEs. We start by introducing the contenders in Section 5.3.1, and then proceed to present the setup and results of our comparison in Section 5.3.2. We aim at replicating the results of the comparative study that was performed in [58], extending it with the trust region framework and a wider range of datasets.

5.3.1. The Contenders

In this subsection, we present the different optimization techniques that we considered for the comparative review. We briefly explain each method and mention relevant parameters values and implementation details. For VDM, and the quasi-Newton methods, the rest of the parameters can be found in Chapter 4.

- **Vanilla gradient descent (VGD)**. VGD is theoretically similar to LS methods with two important con-

siderations. First, we use the identity as the Hessian approximation ($\nabla^2 C = I$). Second, the step size is defined by the user at the beginning and remains fixed throughout the optimization procedure. For the latter, we chose a value of $\alpha = 200$, which is often used in the context of tSNE.

- **Gradient descent with momentum and adaptive step size (VDM)**. This is the implementation that is often used to optimize tSNE and which we presented in Chapter 4. Instead of fixing the step size throughout the iterative procedure, to accelerate convergence, momentum, and an adaptive scheme are used. The algorithm includes the "early exaggeration" trick which is useful to help clusters to form correctly, we used a value of 12 for this parameter as recommended in [55] and stopped exaggerating at the 250^{th} iteration.

- **Nonlinear conjugate gradients (nlCG).** NlCG attempts to find the minimum of a given function by generating successive conjugate directions. Here we used SciPy's implementation of nonlinear conjugate gradients which uses the Polak-Ribiere updating formula to generate these directions [44]. To find the suitable step size it uses a more sophisticated interpolation-based line search with strong Wolfe conditions. We used this method with its default parameters.

- **L-BFGS.** This is a quasi-Newton method that generates an approximation of the Hessian of the matrix using the information of the gradient from successive iterations. To keep memory requirements low, we reset the approximation every $m = 100$ iterations as recommended in [58]. As with nlCG, we used Scipy's implementation of L-BFGS which is a wrapper to the original Fortran implementation [64], which uses a line search based on strong Wolfe conditions.

- **Quasi-Newton methods.** We consider the LS and TR frameworks with SD. For LS, we use $R10$ to include low-dimensional information and try two different solvers: Cholesky factorization (direct) and CG (iterative). We denominate the resulting methods SD-LS-CHOL and SD-LS-CG, respectively. As for TR, we do not update SD ($R0$) and use the CG solver, which yields SD-LS-CG. We truncate the CG solvers at $K_{CG} = 500$ iterations to bound the computation times in case of ill-conditioning. Finally, considering the results of the first section of this chapter, we set $\alpha_0 = 10$ for LS and $\Delta_0 = m_{\Delta_0}||\boldsymbol{p}_{ls}||$, with $m_{\Delta_0} = 1$ and $||\boldsymbol{p}_{ls}||$ set ad hoc for each dataset taking into account Table 5.2.

5.3.2. The Comparative

Having described the contenders, we now specify the questions that we seek to answer in this comparative study.

Experiment 5.4. *Which is the best optimization method for tSNE using exact cost function values and gradients? Are SD-LS and SD-TR competitive with the current state of the art, VDM?*

Experimental setup. In addition to the method-specific configuration that we listed in the previous subsection, there are important aspects of the experimental setup that we now detail. First, because we are using exact cost function values and gradients, we only considered the small datasets: Coil-20, Digits, Coil-100, and PenDigits. Second, to guarantee that the results were held across different initializations, for each method, we performed five runs. Four with a random initial embedding, and the fifth one with the PCA initialization described in Chapter 4. Finally, we stop the methods when one of three conditions is satisfied: the variation between consecutive iterates is less than 1×10^{-6}, a predefined number of iterations have been performed (500 for quasi-Newton methods and 2000 for the rest), or 5000 seconds have elapsed. We also considered the change in the cost function value to validate convergence but did so offline to avoid affecting the runtime of the algorithms. It is also important to keep in mind that methods that use LS or TR can also stop earlier if the step size or the radius becomes 0, which means that no more progress can be achieved.

Results. In Figure 5.5, we present the Kullback-Leibler divergence with respect to the number of iterations and elapsed time for a single run of the considered methods for the different datasets. Starting with the quasi-Newton methods based on SD, it can be observed how they outperformed all other methods both in terms of iterations needed and elapsed time. Focusing on the differences in performance among these methods, we noticed that even if in most cases LS and TR approach converged to similar values, in general, LS-based methods achieved a steeper initial reduction of the cost function value, which gave them a slight edge. This is because these methods can take longer directions that seem to be crucial for the aggressive initial reduction that characterizes SD. Finally, it is worth noting how for the small datasets, the convergence rate of SD-LS-CHOL and SD-LS-CG is almost the same. This is to be expected, given that with fewer data points and a

Dataset	VGD	VDM	nlCG	L-BFGS	SD-LS-CHOL	SD-LS-CG	SD-TR-CG
Coil-20	0.58 ± 0.02	**0.45 ± 0.00**	0.47 ± 0.01	0.46 ± 0.00	0.49 ± 0.02	0.47 ± 0.00	0.47 ± 0.01
Digits	0.92 ± 0.02	**0.76 ± 0.00**	0.79 ± 0.02	0.78 ± 0.01	0.78 ± 0.01	0.79 ± 0.01	0.79 ± 0.01
Coil-100	1.95 ± 0.03	**0.78 ± 0.00**	1.36 ± 0.19	1.15 ± 0.18	0.88 ± 0.07	0.83 ± 0.02	0.88 ± 0.02
PenDigits	3.10 ± 0.09	1.51 ± 0.23	2.26 ± 0.48	2.05 ± 0.51	**1.01 ± 0.01**	1.02 ± 0.02	1.04 ± 0.01

Table 5.3: Average Kullback-Leibler divergence value for the different methods and datasets at the end of the allotted time for the optimization procedure (5000 seconds). The results indicate that, when VDM converges, it can achieve slightly lower values of KL than quasi-Newton methods. Nevertheless, it takes more time for VDM to reduce cost function value due to the "early exaggeration" phase.

Dataset	VGD	VDM	nlCG	L-BFGS	SD-LS-CHOL	SD-LS-CG	SD-TR-CG
Coil-20	0.58 ± 0.02	**0.45 ± 0.00**	0.54 ± 0.03	0.46 ± 0.00	0.49 ± 0.02	0.47 ± 0.00	0.47 ± 0.01
Digits	0.92 ± 0.02	**0.76 ± 0.00**	0.90 ± 0.06	0.78 ± 0.01	0.78 ± 0.01	0.79 ± 0.01	0.79 ± 0.01
Coil-100	3.02 ± 0.05	1.56 ± 0.15	1.89 ± 0.34	2.24 ± 0.66	0.90 ± 0.07	**0.87 ± 0.03**	0.99 ± 0.05
PenDigits	5.08 ± 0.13	3.93 ± 0.40	2.95 ± 0.77	2.79 ± 0.74	**1.05 ± 0.03**	1.05 ± 0.01	1.17 ± 0.02

Table 5.4: Average Kullback-Leibler divergence value for the different methods and small datasets at the end of the first 1000 seconds of the optimization procedure. Note how at this point, when the methods have not yet converged, as is the case with Coil-100 and PenDigits, quasi-Newton methods outperform the alternatives.

relatively high limit on the number of CG iterations of $K_{CG} = 500$, the residuals of CG are generally close to the desired tolerance.

Moving to the general picture of the experiment, we observed that among the other techniques, the highly tunned VDM was the most effective one, converging to the same minimum as the quasi-Newton methods. At the start, the convergence of VDM is slower because of the plateau produced by the early exaggeration phase to avoid local optima. Nevertheless, once the plateau is resolved, it sharply reduces the cost function value, outperforming the alternatives. Regarding L-BFGS and nlCG, as can be observed in Table 5.3, these can achieve the same, or even lower, optimums that the other methods. Nevertheless, it is important to note how their behavior is less consistent: with some initializations, they converged to a minimum, and, with others, the optimization procedure was stopped early because of convergence failures of the line searches, which use stronger conditions.

Even though most methods converge to similar optimums, it is important to note that their process can significantly differ. As can be observed in Figure 5.5, and corroborated in Table 5.4, methods such as SD-LS-CHOL, SD-LS-CG, SD-TR-CG, L-BFGS, and nlCG produce a larger initial decrease of the cost function value than VDM. Nevertheless, VDM catches up towards the end of the optimization, being able in some cases to reach lower KL values. It is also worth noting the visual differences in the resulting embeddings. In Figure 5.6, we present the embeddings of the PenDigits dataset, using the PCA initialization. It can be seen how VDM is better at untangling local optima (note the lower left region of the embedding with the clusters of purple and red points), and LS methods, which use SD with $R10$, produce more white space between clusters.

Conclusions. In the experiments, we observed that the quasi-Newton methods that result by using SD within the LS and TR optimization frameworks are consistently more effective at reducing the cost function value than other common optimization techniques such as VDM, L-BFGS, and nlCG. This confirms the results of [58], where it was shown that a carefully constructed Hessian approximation could outperform state-of-the-art Hessian-free methods. Furthermore, we extended the results of this work by assessing the effectiveness of adding low-dimensional information to the SD. The experiments indicate that this extra information is more effective when the LS framework is used. We also evaluated the TR framework which, besides [33], had not been used to optimize the cost functions of graph-based nonlinear embedding techniques. Despite its inferior performance with respect to LS and the fact that as-is it cannot take advantage of the addition of low-dimensional information to SD, we found that, by correctly setting parameters such as the initial radius, it could be a competitive alternative, taking in some cases less time to converge.

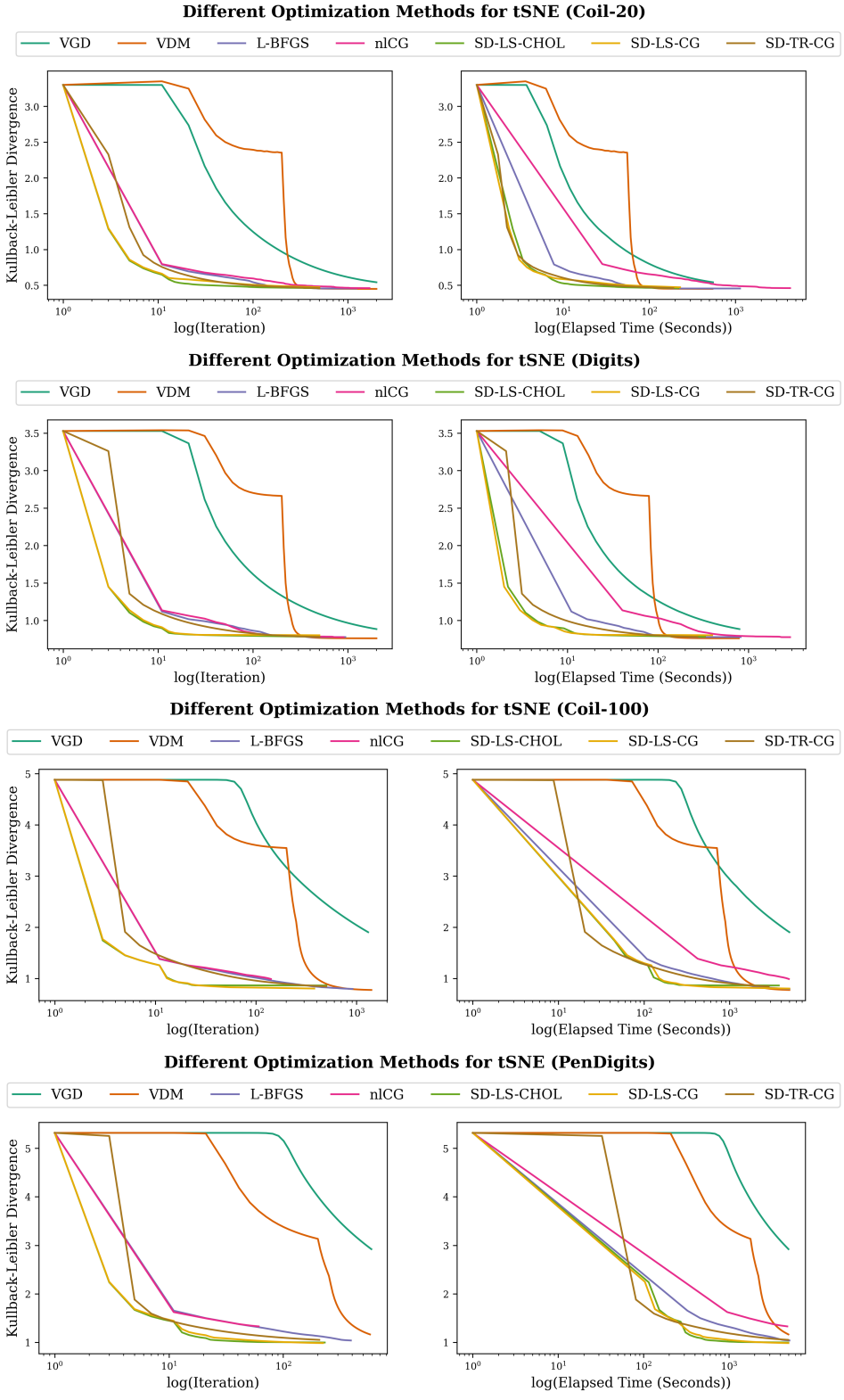

Figure 5.5: Comparison of different methods for optimizing tSNE's Kullback-Leibler divergence (KL) with different datasets. In most cases, the execution of the algorithms stopped because they reached the time limit of 5000 seconds. The left column presents KL as a function of the iteration, and the right column, KL as a function of the elapsed time. A logarithmic scale was used for the x-axes to improve the chart's readability. Note how quasi-Newton methods perform consistently better than the alternatives, both in terms of elapsed time and iterations.

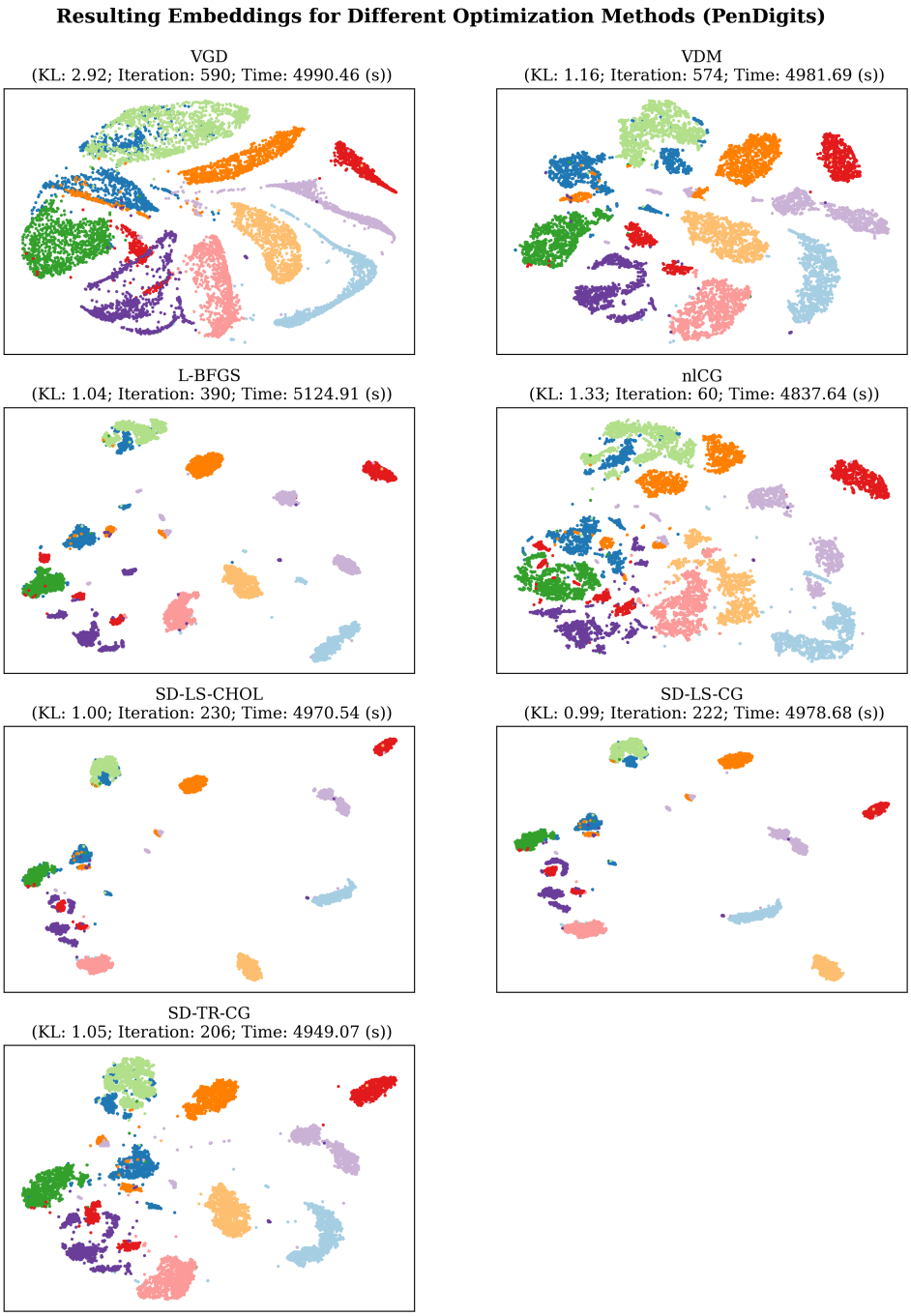

Figure 5.6: Resulting embeddings of the PenDigits dataset after 5000 seconds with tSNE, using different optimization techniques. Note how quasi-Newton methods have lower cost function values and also look better than nlCG and VGD. In contrast to VDM, the embeddings of quasi-Newton methods have more white space between clusters, especially in SD-LS, where we used $R10$.

6

Fast Quasi-Newton Methods for tSNE

In the previous chapter, we showed how Quasi-Newton methods based on the Spectral Direction can converge faster than other optimization techniques for the case of tSNE. We found that for LS, updating the SD with low-dimensional information every 10 iterations could further accelerate its convergence and that for TR, not doing it worked best. Despite the potential of SD-LS and SD-TR, in the experiments, we had to restrict ourselves to smaller datasets of the order of thousands of points. The reason for this is that there are several steps in the computational pipeline that do not scale well with the size of the data. We identified the following bottlenecks:

1. In LS, if we use Cholesky decomposition to solve the linear system, then we also need to recompute the factors every time SD changes. Given that we found $R10$ to be the best performing updating schedule, this operation, which depends on the size of the input data, can add a significant overhead.

2. The cost function and gradient calculation have a complexity of $O(N^2)$ because all pairwise interactions between points in the embedding need to be computed. And both of these computations need to be carried out at least once per iteration in LS and TR.

3. If the method of conjugate gradients is used to solve the system approximately, the speed of each LS and TR iteration depends on how many CG iterations are needed to find a suitable solution. This number depends on the conditioning of the matrix, which in the last chapter we show can get worse with the addition of low-dimensional information.

Experiment	Page	Description
6.1	38	Measuring the scaling behavior of the Cholesky decomposition.
6.2	39	Approximating the gradient and cost function value using Barnes-Hut.
6.3	40	Effect of different thresholds for truncating the CG solvers.
6.4	43	SD-LS-CG vs SD-TR-CG vs BH-SNE: performance comparison.
6.5	45	SD-LS-CG vs SD-TR-CG vs BH-SNE: different initializations and local optima.

Table 6.1: Experiments covered in this chapter.

We start this chapter in Section 6.1 by addressing these bottlenecks. This process yields a pair of fast quasi-Newton methods, SD-LS-CG and SD-TR-CG, that can scale to datasets with hundreds of thousands of points. In Section 6.2, we compare these methods with their Hessian-free alternative BH-SNE, which is currently one of the most popular tSNE implementations for embedding large datasets. A summary of the experiments that we present in this chapter is listed in Table 6.1.

6.1. Fast Quasi-Newton Methods for tSNE

In this section, we construct a pair of fast quasi-Newton optimization methods through a series of experiments that aim to address the aforementioned bottlenecks. We are concerned with three quasi-Newton

variants that use the Spectral Direction (SD), these are line search with Cholesky factorization (SD-LS-CHOL with $R10$), line search with conjugate gradients (SD-LS-CG with $R10$) and trust region with conjugate gradients (SD-TR-CG with $R0$). All the experiments in this section were run on a MacBook Pro with 16GB of RAM and a Quad-Core Intel Core i5 clocked at 2.3 GHz.

6.1.1. Scaling of the Cholesky Factorization

The first bottleneck that we address is the usage of Cholesky factorization for solving the linear system that arises in the Line Search framework. Even though using the Cholesky factors to solve the system is efficient, considering the sparsity of SD matrices, their initial construction takes a considerable amount of time. We argue that for LS to be usable with large datasets, this setup step should be avoided. Concretely, we are interested in the following questions.

Experiment 6.1. *How does the Cholesky solver scale with the size of the data N? How does it fare against the alternative, CG?*

Experimental setup. To measure how the difference in the performance of the Cholesky decomposition (CHOL) and conjugate gradients (CG) solvers, we used ten samples of the MNIST dataset with sizes $N^{'}$ evenly spaced along the line between 10 and 30000. For each sample, we computed its Cholesky factors using a fill reducing permutation (found with the nested dissection algorithm) and used them to solve the linear system $B\boldsymbol{p} = -\nabla C$ using forward-backward substitution, where B is SD with or without low-dimensional information, and $\nabla C \in \mathbb{R}^{N^{'} \times d}$ is the gradient of a random point $Y \in \mathbb{R}^{N^{'} \times d}$. For each sample size, did this five times. We followed the same process for the CG solver, using the same initializations and truncating the solver after $K_{CG} = 1000$ iterations. In each case, we measured the CPU time and the residual of the linear system.

Results. As can be observed in the central panel of Figure 6.1, solving the linear system using CHOL is significantly faster than with CG especially as the size of the sample increases. This is because larger datasets generally need more CG iterations which are bounded by the cost of sparse matrix-vector multiplication. Despite this advantage, in contrast with CHOL, CG does not require an initialization step. As can be observed in the left panel of the figure, this cost significantly increases with the size of the data. For example, while it took on average 0.86 minutes to compute the Cholesky factors for $N^{'} = 10000$, it took more than 9 minutes to do the same for the largest sample with $N^{'} = 30000$. Finally, regarding the accuracy of the solvers, the right panel of the figure shows how CHOL has the advantage with lower residuals on average. In contrast, CG has residuals that are several orders of magnitude larger than those of CHOL (10^{-6} vs 10^{-23}). This is especially the case when the matrix is ill-conditioned and CG needs to be truncated before reaching the desired tolerance. Despite this issue, we will show at the end of this section how larger errors do not seem to significantly affect the quality of the results obtained by tSNE and can even help in some cases to avoid local optima.

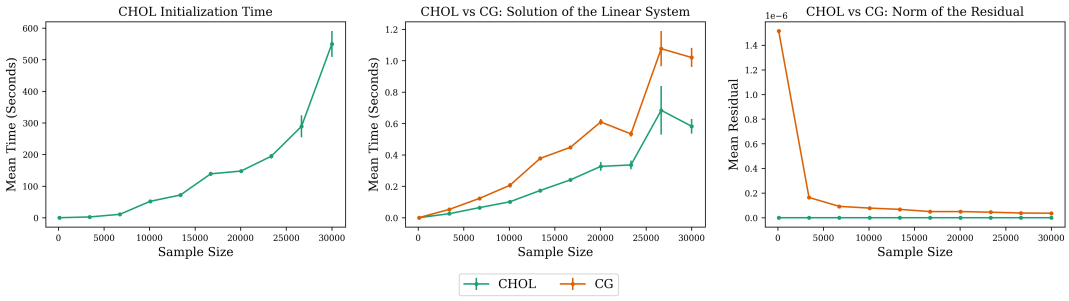

Figure 6.1: Comparison of direct (CHOL) and iterative (CG) solvers for LS for samples of different size of the MNIST dataset. The left panel shows the average time that it takes to setup the Cholesky factors using the nested dissection algorithm. In the center and right panels we measure the average times and residuals for these methods, respectively.

Conclusions. The Cholesky factorization is usually recommended for matrices with up to 10^8 that do not change frequently during the optimization procedure. This is because, once the Cholesky factors are computed, these can be cached and used to solve the linear system efficiently. In the previous chapter, we found that updating SD every 10 iterations could improve the convergence of SD-LS. Using CHOL, the constant recomputation of the factors would severely affect the performance of our methods. Additionally, even if we were to use SD without updating ($R0$), as is the case with TR, the initial setup of the matrix can offset the

convergence gains of quasi-Newton methods for the larger datasets. For instance, more than 80% of the total reduction of tSNE cost function value for MNIST can be achieved in around 17 minutes using BH-SNE (more on this in Section 6.2); while it takes up to 23 minutes to compute the Cholesky factors of SD for this dataset. This illustrates how, by the time SD-LS with CHOL starts iterating, BH-SNE would already be far ahead in the optimization. For this reason, we opt to use CG instead of CHOL.

> Instead of solving the LS linear system directly using the Cholesky factors (LS-CHOL), we use the iterative CG solver (LS-CG) instead, which does not need an expensive initialization phase.

6.1.2. Approximating the Gradient and the Cost Function Value

Different ways to accelerate tSNE based on approximations of the gradient and the cost function have been proposed. One of the most popular ones is the Barnes-Hut algorithm which was presented back in Chapter 3. In this subsection, we evaluate the effect of this approximation on LS and TR. In addition to ∇C, which is used to compute the descent direction, these frameworks also make use of C to determine an optimal scaling of the step size. Although our final aim is to approximate both of these quantities (both take $O(N^2)$ time), in this subsection we also evaluate the effect that they have separately on the robustness of our quasi-Newton methods. Therefore, we investigate the following questions.

Experiment 6.2. *Can the Barnes-Hut approximation be used to accelerate SD-LS-CG and SD-TR-CG, without compromising their robustness? Which are the individual effects of approximating C and ∇C on the performance of these optimization procedures?*

Experimental setup. In this experiment we considered three types of run in which we varied the approximation degree of the gradient $\theta_{\text{grad}} \in \{0, 0.5, 1, 2\}$, the cost function $\theta_{\text{KL}} \in \{0, 0.5, 1, 2\}$, and both $\theta_{\text{grad}} = \theta_{\text{KL}} = \theta \in \{0, 0.5, 1, 2\}$. Given that we still use exact quantities for comparison ($\theta = 0$), we were restricted to the small datasets. We evaluated the effects of the different approximations for SD-LS-CG and SD-TR-CG. For the former, we set $\alpha_0 = 10$ and for the latter, we set the initial radius $\Delta_0 = ||\boldsymbol{p}_{ls}||$ according to the values in Table 5.2. We let the methods run for 500 iterations or until 5000 seconds had passed, which is generally enough for them to converge. Note that the runs can also stop before these limits if α or Δ become 0, which means that no further progress can be made.

Results. Starting from the effect of using approximate gradients, we found that for the largest $\theta_{\text{grad}} = 2$, the methods did not converge, stopping in the middle of the optimization because the step size α or the radius Δ became 0. For $\theta_{\text{grad}} \leq 1$ not only the methods converge, but did so faster than the exact version, which suggests that the errors are not large enough to impact the Armijo condition in LS or the radius update in TR. Moving to the usage of approximate cost function values, we found that they did not seem to affect the convergence of the algorithms. Nevertheless, they did harm their runtime, especially of SD-TR. With larger θ_{KL}, Δ oscillated between large and small values, which caused the generation of iterates that did not significantly reduce the cost function value. It is important to note that the effects that we just mentioned appeared when these quantities were more coarse, with common parameters such as $\theta_{\text{grad}} = \theta_{\text{KL}} = 0.5$ we observed that the methods had similar convergence behavior to their exact counterpart, but with a shorter runtime.

When we approximated both quantities, $\theta = \theta_{\text{grad}} = \theta_{\text{KL}}$, we observed that the aforementioned effects were compounded. Namely, as can be observed in Figure 6.2 for $\theta \leq 1$, we observed a significant reduction in runtime without degrading the convergence. In contrast, when we used $\theta = 2$, the methods failed to converge. As we mentioned in the previous paragraph, this is because the errors in the gradient approximation, when amplified by the Hessian approximation, SD, seem to yield poor directions that do not reduce the cost function value. This is why the gap between this and the other approximations is not because of the speed of the computations, but because of the slow convergence.

Conclusions. When using BH approximation with LS and TR, it is important to take into account that the effects of using approximate gradients and cost function values differ. Between these two, we observed that approximating the gradient had a more significant role in the convergence of the algorithms. We believe the reason for this is that the gradient is used to obtain the direction to the next iterate and, if it is noisy, then the latter might be close to not being a descent direction, which will hurt the convergence of the method. Interestingly, the backtracking procedure with the Armijo condition in LS and the ratio ρ for determining

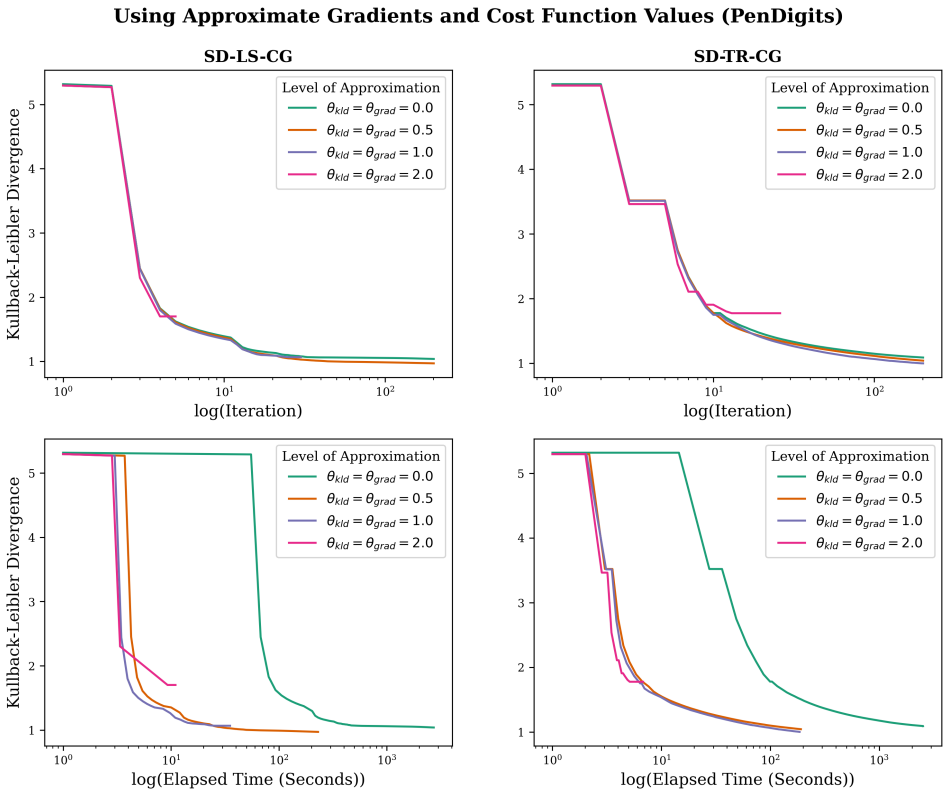

Figure 6.2: The effect of using approximate gradients and cost function values for the PenDigits dataset. The left column presents the results for LS and the right one for TR. The first row displays the Kullback-Leibler divergence (KL) as a function of the iteration, and the second row, as a function of the elapsed time in seconds. For the approximation degree, we considered $\theta \in \{0, 0.5, 1, 2\}$. Note how, with $\theta > 1$, the algorithms have convergence problems, while, with $\theta <= 1$, both LS and TR exhibit a steady decrease of KL.

the quality of the next iterate in TR seem to be robust to these approximations. We acknowledge that this might be due to the specific error that BH induces and that this might change if other approximations such as Flt-SNE or GPGPU tSNE are used. Finally, we found that SD-LS-CG and SD-TR-CG with $\theta = 0.5$, which is the recommended value for BH-SNE, performed consistently good across different datasets, reducing the runtime of each iteration without affecting the convergence of the quasi-Newton methods.

In the rest of the experiments in this chapter, we use approximate gradients and cost function values to be able to handle larger datasets. We set $\theta = \theta_{\text{grad}} = \theta_{\text{KL}} = 0.5$, which we found to perform consistently good across the different quasi-Newton methods.

6.1.3. The Effect of the Number of CG Iterations

The final bottleneck that we address in this section is the number of iterations that take the CG solvers of LS and TR to converge. Back in the first subsection, we showed how, for LS, CG can be a good alternative to the Cholesky factorization because of the steep initialization costs of the latter, which become especially problematic with the updating interval $R10$. Despite its advantages, solving a linear system with CG can be significantly more costly than using the Cholesky factors because of the potentially high number of iterations that are needed to reach convergence. To mitigate this issue, we propose truncating the number of maximum CG iterations K_{CG} that are allowed. Concretely, we seek to answer the following questions.

Experiment 6.3. *Can we truncate the number of iterations in the CG solvers that are used in quasi-Newton without affecting their performance? What should be the threshold value K_{CG}? Are there differences in the behavior of LS and TR in this regard?*

Experimental setup. We tried different values of the threshold $K_{CG} \in \{10, 50, 100, 500\}$ for both the SD-LS-CG and SD-TR-CG algorithms. We set the maximum value to 500, taking into consideration that, in the

K_{CG}	SD-LS-CG		SD-TR-CG	
	Time (s)	KL	Time (s)	KL
10	866.76	2.64	874.57	2.73
50	1922.47	2.22	1498.44	2.43
100	2806.76	2.17	2063.80	2.38
500	8834.51	2.11	1789.44	2.38

Table 6.2: The time that it took to perform 200 iterations and final Kullback-Leibler divergence value (KL) for different truncating thresholds K_{CG} for SD-LS-CG and SD-TR-CG for the F-MNIST dataset.

case of LS, which is the one that on average needs more CG iterations, larger values resulted in significantly higher run times, which would be unrealistic in a practical scenario. For the rest of the parameters, we set the approximation degree $\theta = 0.5$; $\alpha_0 = 10$ and $R10$ for LS; and for TR $\Delta_0 = m_{\Delta_0}||\boldsymbol{p}_{ls}||$ with $m_{\Delta_0} = 1$, $||\boldsymbol{p}_{ls}||$ according to Tables 5.2 and 6.3, and $R0$. Note for this experiment we also consider the medium datasets given that we use the BH approximation. Finally, we stopped the runs when 200 iterations had passed or when either α or Δ became 0.

Results. First, for the largest threshold value $K_{CG} = 500$, we analyzed the number of CG iterations at every iteration and compared it with the behavior of the cost function value for both LS and TR. This is presented in Figure 6.3 for the F-MNIST dataset. As can be observed, both algorithms follow similar patterns. In LS, at the start, a relatively low number of iterations are needed, exiting out of the CG procedure because the tolerance has been reached (meaning that the solution is accurate enough). Then, once the cost function value has finished its significant initial drop, around the 10^{th} iteration, and the SD is updated with low-dimensional information, the number of CG iterations that are required increased to the limit, significantly slowing down the algorithm. In the case of TR, the sudden drop of the cost function value is also accompanied by a low number of CG iterations. Then, once KL stabilizes, it uses more CG iterations, but, in contrast to LS, it does not reach the limit. We must note that this is likely due to the fact that we are using $R10$ for LS and $R0$ for TR. Therefore, in LS, when SD is updated, the matrix becomes ill-conditioned, which affects the number of iterations that it needs. When K_{CG} is large, this difference can cause a significant performance gap. For instance, with $K_{CG} = 500$, the LS procedure took 150 minutes to complete 200 iterations versus the 30 minutes that took TR.

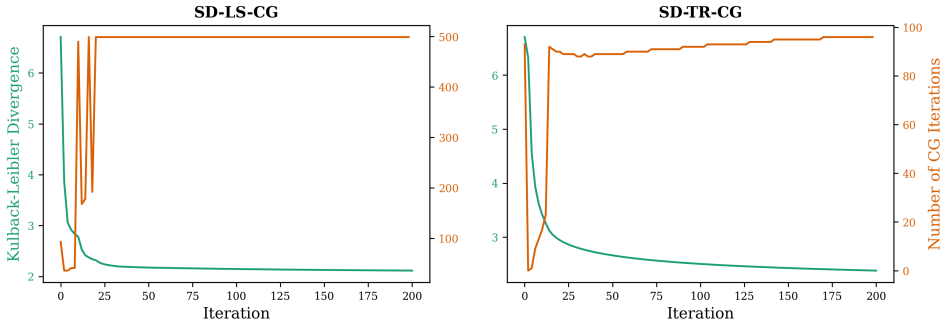

Figure 6.3: The behavior of the Kullback-Leibler divergence (KL) and the number of CG iterations as a function of the iteration for LS and TR, with the F-MNIST dataset. It is possible to observe how the CG solvers of both methods behave similarly throughout the optimization procedure, converging to a large number of CG iterations after a few initial iterations with a low number of CG iterations. The starting phase of the iterative algorithm that requires fewer CG iterations matches the initial sharp drop in KL. Finally, it is worth noting how, in general, LS uses more CG iterations than TR, being closer to the defined threshold K_{CG}.

Next, we analyzed the effect of using different thresholds. The results are shown in Figure 6.4. There, it is possible to observe how for both LS and TR, using $K_{CG} \geq 50$ results in similar convergence behavior. Because TR does not need as many CG iterations as LS in general, there was also not much difference in runtimes between TR runs. In contrast, in LS, it is possible to observe how $K_{CG} = 500$ takes significantly longer than the other runs (take into account the logarithmic scale). To see the more clearly, in Table 6.2, we present, for each threshold value K_{CG}, the time that it took for the 200 iterations to complete and the final cost function value for both LS and TR.

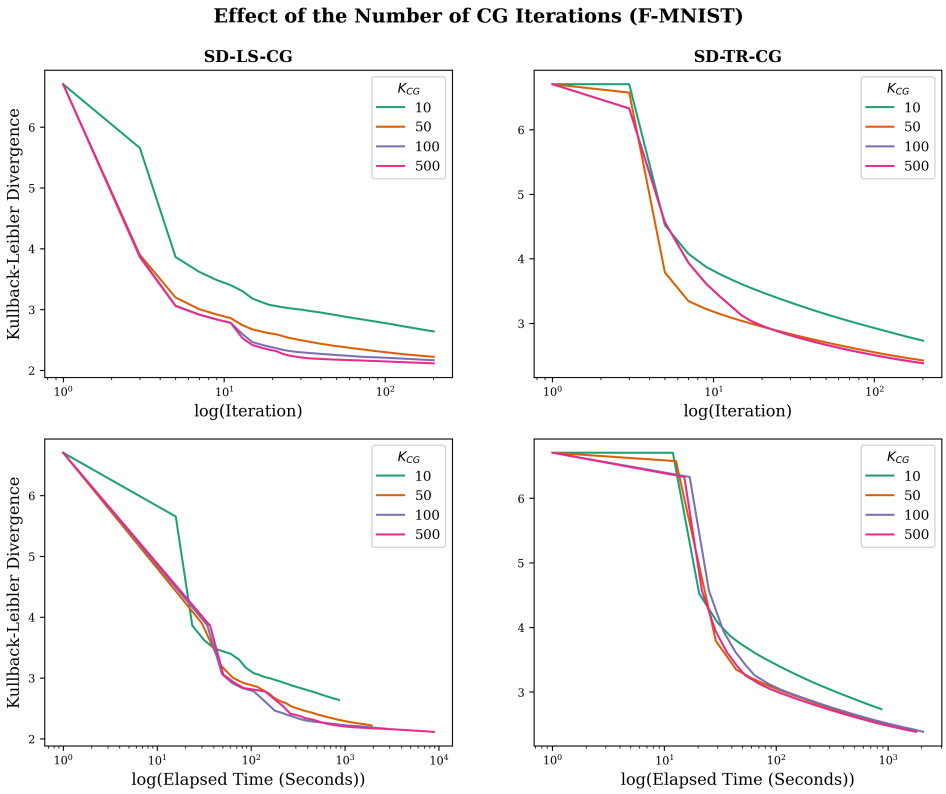

Figure 6.4: The effect of different thresholds K_{CG} for the number of CG iterations in the Kullback-Leibler divergence (KL) of the LS and TR based algorithms. The first row presents KL as a function of the iteration, and the second, as a function of the elapsed time. Note how, when using $K_{CG} = 50$, LS takes less time to reach a similar KL than when using $K_{CG} = 500$.

Conclusions. The results indicate that using a threshold to truncate the number of CG iterations can be especially beneficial for LS. In general, we found that for both LS and TR, using a lower limit of $K_{CG} = 50$ keeps their computational costs down, while preserving their convergence properties. Despite the similarity in the behavior of the number of CG iterations for LS and TR, we found that the former usually reached the threshold limit, making the performance of the algorithm reliant on a correct setting of K_{CG}. We believe this might have to do with the regular updating of the SD matrix. Concretely, once we add low-dimensional information at the 10^{th} iteration, the resulting matrix can have a higher condition number, which is directly linked to the number of iteration that takes the CG solver to converge. The reason this does not happen in TR is that, first, we do not update SD with low-dimensional information; and, second, the trust region radius acts as an additional threshold that halts the CG process. Taking into account the two phases that we identified: a sudden decrease of KL followed by its stabilization, we believe that adaptive schemes that use a higher threshold value in the first phase and a much lower one in the second phase could further improve the performance of LS. Finally, by examining at the embeddings individually, we noted that truncating the number of CG iterations could potentially help to avoid local optima, but we acknowledge that this claim requires further exploration. An example of this for TR is shown in Figure 6.5, where the purple cluster that was split with $K_{CG} = 100$, does not present this behavior with $K_{CG} = 50$.

> Truncating the number of CG iterations reduces the runtime of the algorithms without significantly affecting their convergence properties. We found that with $K_{CG} = 50$ both LS and TR performed consistently well.

6.2. Quasi-Newton tSNE vs BH-SNE

In the previous section, we addressed the most important performance bottlenecks of the line search (LS) and trust region (TR) quasi-Newton methods that we consider, which resulted in algorithms that can scale

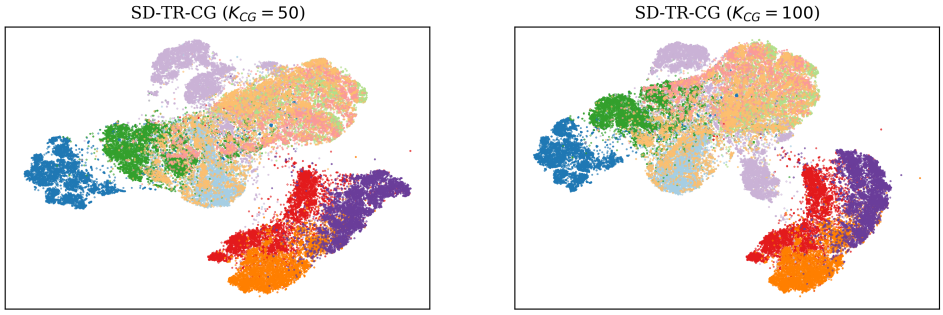

Figure 6.5: Truncating the number of CG iterations could help to avoid local optima. In the figure, we present the final embeddings of the F-MNIST dataset with SD-TR-CG with different $K_{CG} = 50$ and $K_{CG} = 100$. It is possible to observe how, when using fewer CG iterations, the algorithm was able to avoid splitting the purple cluster.

Method	Parameters
BH-SNE	$\theta_{grad} = 0.5$
SD-LS-CG	$\theta = 0.5$; $K_{CG} = 50$; $\alpha_0 = 10$; $R10$
SD-TR-CG	$\theta = 0.5$; $K_{CG} = 50$; $\Delta_0 = \|\|\boldsymbol{p}_{ls}\|\|$; $R0$

Dataset	$\|\|\boldsymbol{p}_{ls}\|\|$
MNIST	1000
F-MNIST	1750
ImageNet (Head0)	400
ImageNet (Mixed3a)	1000

Table 6.3: Parameter values for the experiments. The table on the left presents relevant parameters for the optimization methods, the rest of the implementation details can be found in Chapter 4.

to medium and large datasets: SD-LS-CG and SD-TR-CG. In this section, we compare them against their Hessian-free alternative, BH-SNE. We divided the comparison into two parts. First, we analyze the performance of the optimization algorithms in terms of their effectiveness at reducing tSNE's cost function. Then, taking into account the context in which tSNE is used, which is the visualization of high-dimensional data, we evaluate aspects such as the robustness of the algorithms to different initializations and the way they handle local optima.

Experimental setup. For the experiments in this section, we used a common setup. Starting with the principal parameters of the different methods, we used those presented in Table 6.3. As for the datasets, we considered both the medium and large ones, which are: MNIST, F-MNIST, and ImageNet (Mixed3a and Head0). To guarantee that the results were significant, we ran each combination of method and dataset five times using different initializations. Four of which were random (using randomly picked seeds) and aa fifth one that corresponded to the PCA decomposition of the input data (see Chapter 4). To guarantee convergence, we stopped the runs when one of the following conditions were met: 6000 iterations passed, the elapsed time reached 10000 seconds or the relative variation between consecutive iterates $(Y_k - Y_{k+1})/(1 + Y_{k+1})$ was less than $\epsilon = 1 \times 10^{-6}$. We did not use stopping criteria based on the cost function value since this would slow down BH-SNE, which does not need it for its computations. Nevertheless, we did inspect the variation of cost function value between consecutive iterations offline to have an idea of the value of this metric for the different runs. Finally, we ran all the experiments in this section on a MacBook Pro with 16GB of RAM and a Quad-Core Intel Core i5 clocked at 2.3 GHz.

6.2.1. Performance Evaluation

In this subsection we are concerned with the effectiveness of each method at minimizing tSNE's cost function. For this, we investigate the following questions.

Experiment 6.4. *Which method is the most effective at minimizing the Kullback-Leibler divergence both in terms of iterations and elapsed time? In a time constrained setting, which method produces the best embeddings?*

Results. We start by analyzing the effectiveness of the different methods at minimizing the Kullback-Leibler divergence. In general, we observed that the SD-based quasi-Newton methods were able to produce a significant decrease in the cost function value sooner than BH-SNE. As can be observed in Figure 6.6, for all the considered datasets, SD-LS-CG and SD-TR-CG are ahead of BH-SNE both in terms of iterations and

elapsed time for most of the optimization process. Regarding this phenomenon, note that an important factor for the delay of BH-SNE is its initial plateau, which is caused by the exaggeration of the matrix P. Even if a BH-SNE run that did not use the early exaggeration phase converges to the same cost function value of a run that did use it, it is likely that the clusters of similar points in embedding produced by the former will be fragmented, reflecting that it converged to a poor local optimum.

Despite the fact that quasi-Newton methods were ahead for most of the optimization procedure, we observed that towards the end of the time limit, after BH-SNE had performed thousands of iterations, the three algorithms tended to converge to a similar cost function value, with SD-LS-CG and BH-SNE contending for the first place and SD-TR-CG consistently in the third one. To have a complete picture of the performance of these methods, we measured their average cost function value at two points in time. In Table 6.5, we present the status of the algorithms 1000 seconds after the optimization had started. In this case, quasi-Newton methods have significantly lower cost function values than BH-SNE. In contrast, in Table 6.4, where we present the results at the end of the allowed time (10000 seconds), it is possible to observe that SD-LS-CG and BH-SNE reach similar optimums that have a lower value than SD-TR-CG.

Dataset	BH-SNE	SD-LS-CG	SD-TR-CG
MNIST	2.41 ± 0.01	$\mathbf{2.39 \pm 0.01}$	2.59 ± 0.02
F-MNIST	2.16 ± 0.01	$\mathbf{2.10 \pm 0.01}$	2.20 ± 0.01
ImageNet (Head0)	$\mathbf{3.69 \pm 0.02}$	3.76 ± 0.02	3.95 ± 0.02
ImageNet (Mixed3a)	$\mathbf{3.95 \pm 0.01}$	4.02 ± 0.02	4.22 ± 0.02

Table 6.4: Average Kullback-Leibler divergence value for the different methods and datasets at the end of the allotted time for the optimization procedure (10000 seconds). Note how for MNIST and F-MNIST, SD-LS-CG achieves the lowest cost function value, while for ImageNet, BH-SNE manages to reduce it the most.

Dataset	BH-SNE	SD-LS-CG	SD-TR-CG
MNIST	2.48 ± 0.02	$\mathbf{2.42 \pm 0.01}$	2.68 ± 0.02
F-MNIST	2.21 ± 0.02	$\mathbf{2.13 \pm 0.01}$	2.27 ± 0.01
ImageNet (Head0)	3.83 ± 0.03	$\mathbf{3.80 \pm 0.02}$	4.03 ± 0.02
ImageNet (Mixed3a)	4.08 ± 0.02	$\mathbf{4.05 \pm 0.02}$	4.31 ± 0.02

Table 6.5: Average Kullback-Leibler divergence value for the different methods and datasets at the end of the first 5000 seconds of the optimization procedure. Note how SD-LS-CG has the lowest cost function value for all datasets.

Besides the convergence of the algorithms, we also considered the intermediate results that they produced given that these are often used for interactive data visualizations. To do this, we visualized the embeddings at different points in time. In Figure 6.8, we present an example of this process for the F-MNIST dataset. As can be observed in the grid of embeddings in the lower portion of the chart, when 300 seconds have passed, 3rd column from left to right, quasi-Newton methods, two lower rows, have already settled on an embedding. Meanwhile, BH-SNE, in the first row, is still rearranging the clusters. In general, we observed that as a consequence of the early exaggeration phase in BH-SNE, it took longer for this method to produce embeddings that resembled the final shape.

Conclusions. The results in this section indicate that the SD-based quasi-Newton methods are a competitive alternative to fast Hessian-free alternatives, particularly, BH-SNE. Of the two frameworks that we considered, LS and TR, the former performed consistently better, converging in most cases to cost function values that were below those of BH-SNE. An important thing to note is that the performance of TR seemed to degrade as the size of the dataset increased, which could be a consequence of a poor selection of the initial trust region radius Δ_0 and the fact that, with TR, we did not add low-dimensional information to SD. Finally, we observed that quasi-Newton methods were able to produce reasonable embeddings of the data sooner in the optimization procedure than BH-SNE, which uses an early exaggeration phase to avoid local optima. This feature can play an important role in the progressive visual analytics workflow [48], which argues that it is important to be able to produce reasonable embeddings as soon as possible in the optimization procedure to enhance the user experience. An example of this more rapid decrease of the cost function value by quasi-Newton methods can be observed for the MNIST dataset. In this case, quasi-Newton methods achieved 80%

of the reduction of the Kullback-Leibler divergence in 10 iterations, while it took around 500 iterations for BH-SNE to do the same.

6.2.2. Different Initialization and Local Optima

Given that tSNE is mainly used for visualizing high-dimensional data, and that the Kullback-Leibler divergence can have multiple local optima, the cost function value is not the only criterion that should be considered for evaluating different optimization algorithms. In this section, we cover two additional performance aspects of graph-based nonlinear embedding techniques: their robustness to different initializations and the way they handle local optima. Concretely, we seek to answer the following questions.

Experiment 6.5. *Are quasi-Newton methods robust to different initializations? How much final embeddings change when the initial configuration is altered? How susceptible are quasi-Newton methods to local optima?*

Results. Table 6.4 shows how there is little variation in the final cost function values of quasi-Newton methods. We can analyze the variation in the optimization from two other perspectives: the trajectory and the final embedding. For the former, in the top chart of Figure 6.8, we present the optimization trajectory of the different runs of each method for the MNIST dataset. As can be observed, even if the algorithms reach similar optimums, they exhibit some variation in the middle of the optimization procedure. This is a direct consequence of the way the points were arranged in the initial embedding Y_0. This effect is more noticeable for BH-SNE where the runs can exit the plateau at different points in time.

Regarding the final embeddings, in the bottom grid of Figure 6.8 we present the outcome of the runs with different initializations for every method for the MNIST dataset. As can be observed, even though they have similar cost function values, their appearance can vary. The issue that concerns us is the presence of local optima, which, in the embedding, corresponds to points of the same class that were separated. BH-SNE seems to handle this well by using the early exaggeration phase. The clusters in the embeddings generated by this algorithm tend not to split, regardless of the initialization. In contrast, we observed more variation in the outcomes of SD-LS-CG and SD-TR-CG. Of these, LS tended to "fall" less often in local optima, likely because it can take longer steps at the start of the optimization procedure, which seems to be crucial for the correct separation of the clusters. Finally, across the different datasets, we observed that by initializing the embedding with the PCA decomposition of X, most of the problems related to local optima in quasi-Newton methods were mitigated. This can be observed in Figure 6.9, where we present the final embeddings of the PCA run for the different methods and datasets.

Conclusions. An important feature of tSNE is the early exaggeration phase, which is fundamental to avoid local optima. In the experiments, we noticed that the SD Hessian approximation has a similar effect, guiding the data that belong to clusters in the high-dimensional space towards the same areas of the embedding. Nevertheless, we noted that for SD to be effective at separating the clusters, two things need to happen. First, long steps should be allowed in the first several iterations, for this reason, LS performed better than TR. Second, we found that the algorithms were able to produce better-looking embeddings if a more careful initialization such as PCA was employed. Taking into consideration that even with PCA the embeddings of quasi-Newton methods can suffer from minor cluster-splitting issues, it would be interesting to see whether this could be further improved by using early exaggeration. In preliminary experiments, we found that when the P matrix was exaggerated in quasi-Newton methods, the optimization process could become unstable.

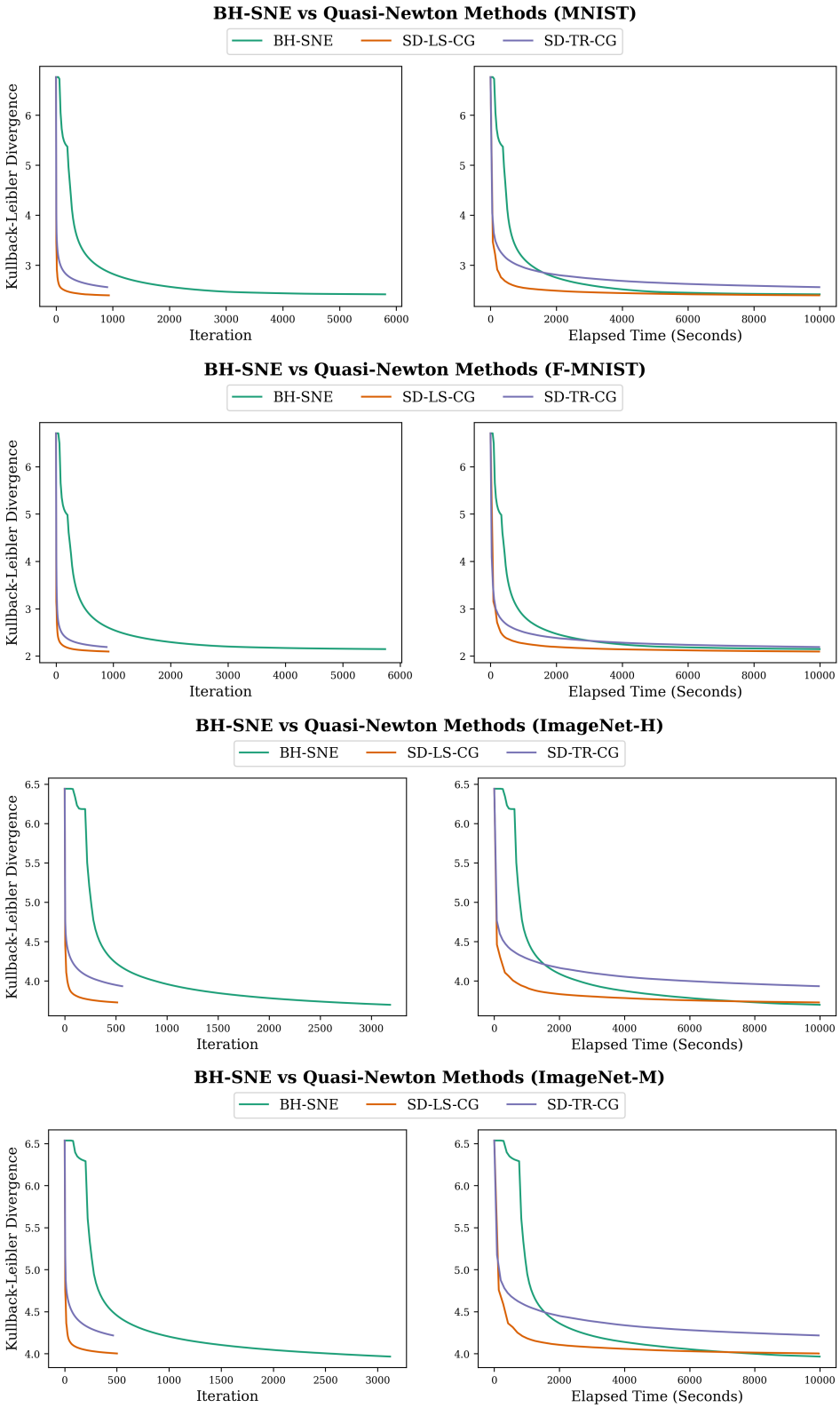

Figure 6.6: Comparison of different methods for optimizing tSNE's Kullback-Leibler divergence (KL) with different medium and large datasets. In most cases, the execution of the algorithms stopped because they reached the time limit of 10000 seconds. The left column presents KL as a function of the iteration, and the right column, KL as a function of the elapsed time. Note how, even though the quasi-Newton methods (SD-LS-CG and SD-TR-CG) and BH-SNE eventually converge to similar values of KL, the former do so in significantly fewer iterations and shorter time.

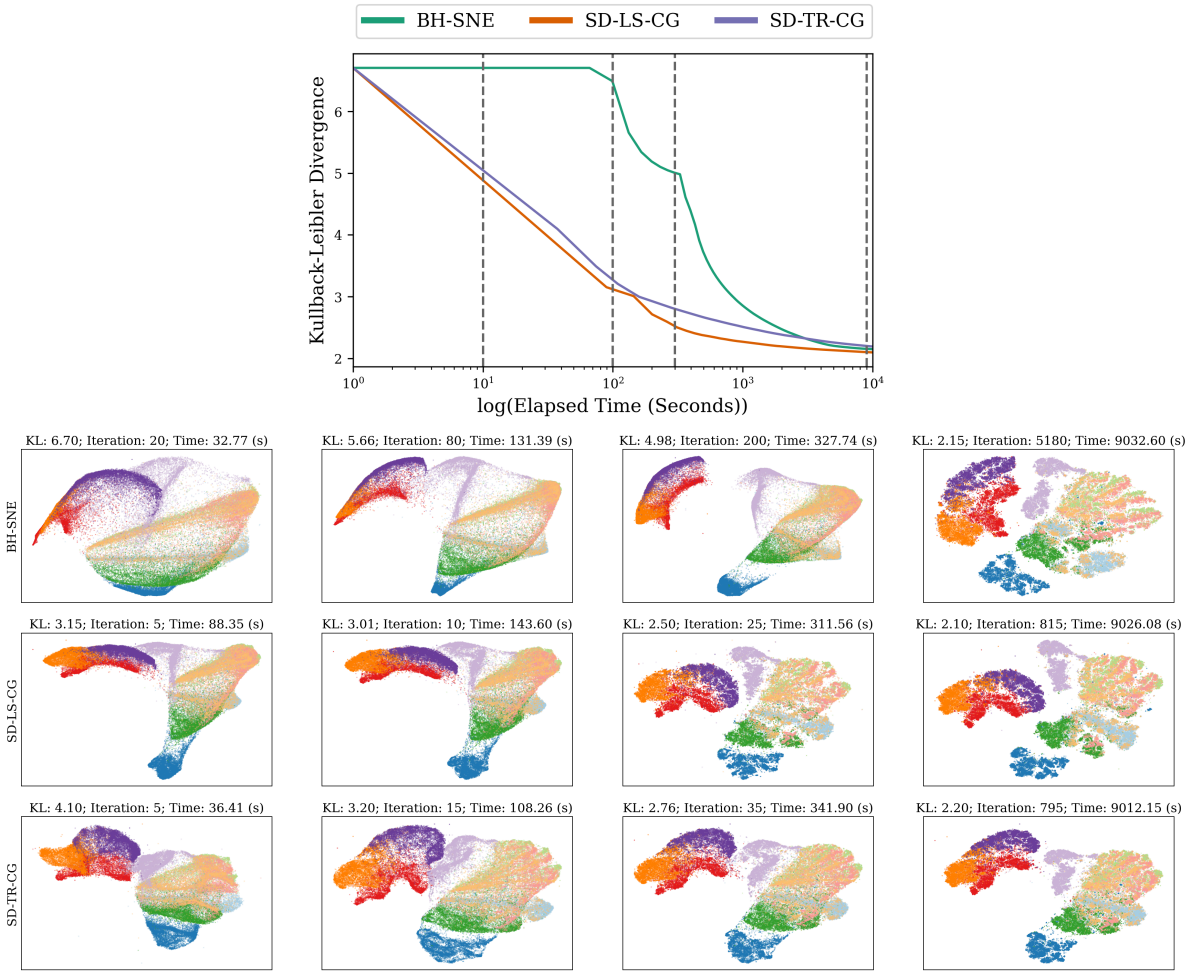

Figure 6.7: Snapshots of the optimization process of BH-SNE, SD-LS-CG, and SD-TR-CG for the F-MNIST dataset. The line chart on the top presents the Kullback-Leibler divergence (KL) as a function of the elapsed time for the different methods. The gray vertical dashed lines correspond to specific points in time that we inspect, these are 10, 100, 300, and 9000 seconds. In the grid of charts below, we present the corresponding partial embeddings for these points in time for the different techniques. Note how within 100 seconds of having started (second column of the grid from left to right), the SD-based methods have already separated the clusters, which corresponds to a lower KL value. In contrast, at this time, BH-SNE is still in the plateau phase, which is needed to avoid "falling" into local optima.

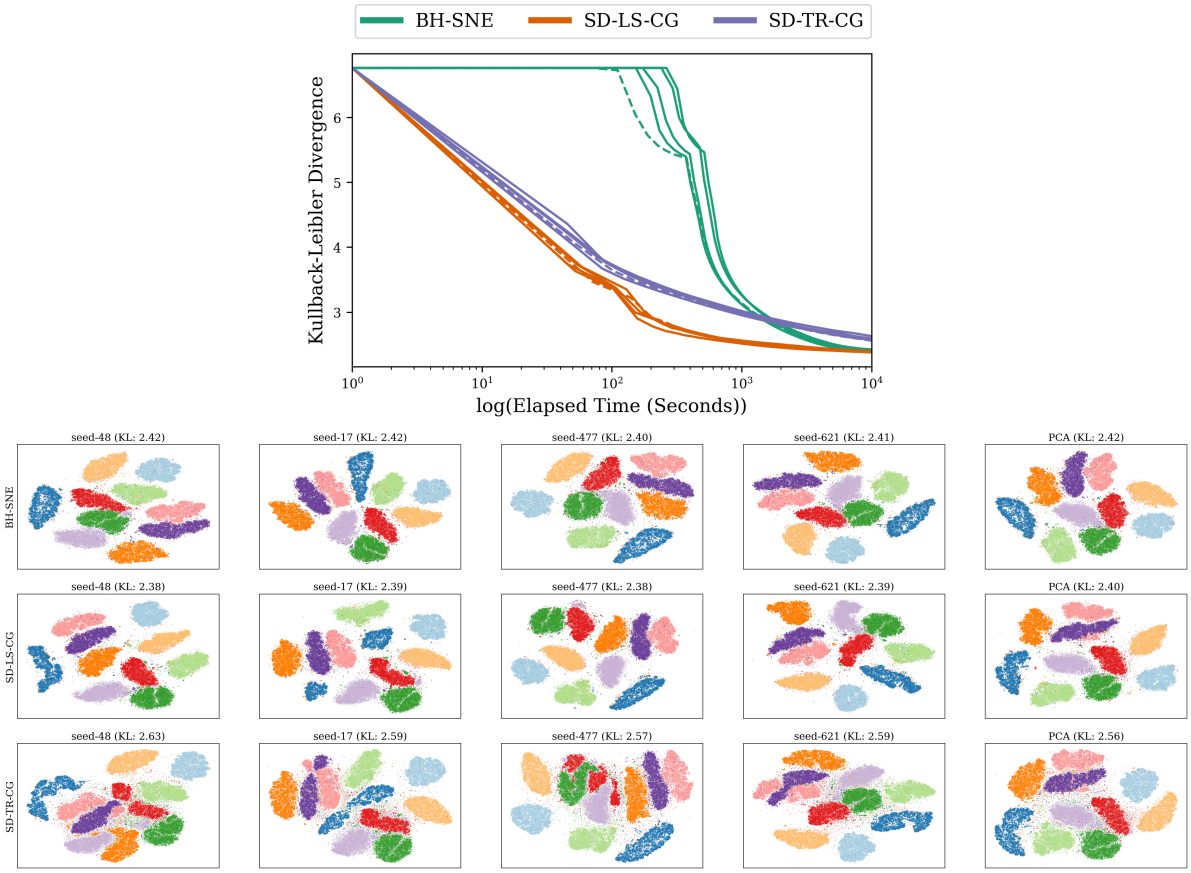

Figure 6.8: The effect of using different initializations with BH-SNE, SD-LS-CG, and SD-TR-CG for the MNIST dataset. The top chart presents the Kullback-Leibler divergence (KL) as a function of the elapsed time. Solid lines correspond to a random initialization and dashed ones to runs with PCA initialization. The grid of charts below presents the resulting embeddings with the different initializations. Note how, even though they all have similar cost function value, the appearance might differ due to the convergence to a local optimum. This is especially noticeable for TR (last row).

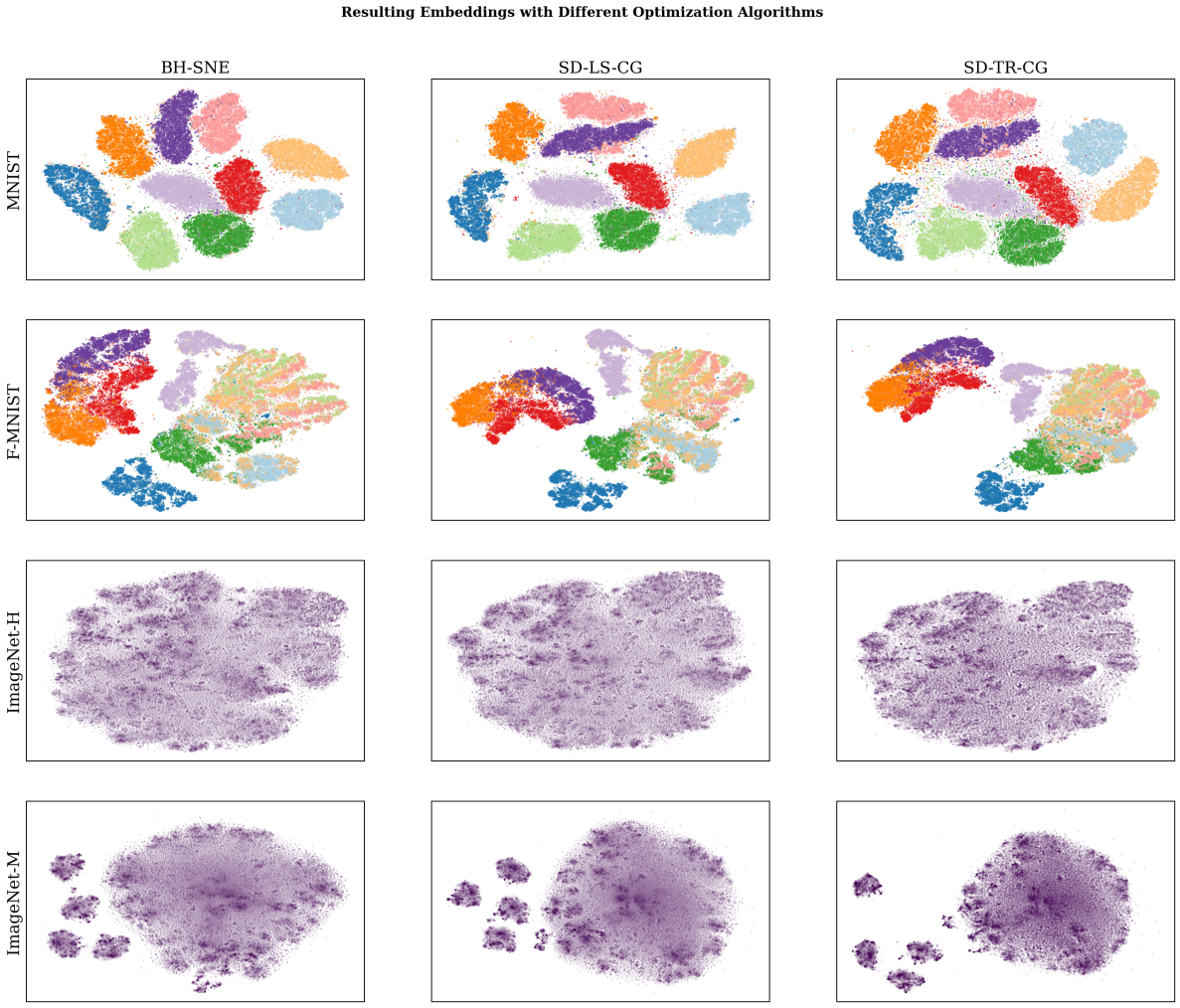

Figure 6.9: Embeddings obtained for the different datasets using BH-SNE, SD-LS-CG, and SD-TR-CG.

7

Discussion and Conclusions

In this work, we departed from the Spectral Direction (SD), a Hessian approximation proposed in [58], and explored its usage within the Line Search (LS) and Trust Region (TR) optimization frameworks. By addressing several of their performance bottlenecks, we found that the resulting quasi-Newton methods, SD-LS-CG and SD-TR-CG, are faster than BH-SNE (the most popular Hessian-free alternative) at visualizing large high-dimensional datasets. In this chapter, we discuss the results of our experiments, analyze the limitations of our methods, and present possible future avenues for investigation.

7.1. Discussion

In the following paragraphs, we will discuss the main results of this work, which are categorized into three sections: the addition of low-dimensional information to SD, the scaling of quasi-Newton methods to handle large datasets, and the comparison of their performance against BH-SNE.

Adding iteration-dependent information to SD can improve its performance. In [58], it was argued that to keep computational costs down, it was preferred a constant version of SD, which only depends on the high-dimensional matrix of affinities P. The experimental results indicate that, for tSNE, the convergence of SD can be further accelerated by using iteration-dependent information coming from the embedding Y. Particularly, we found that adding this low-dimensional information to the SD periodically offered the best trade-off between the decrease of the cost function value and computation time.

We observed that this periodical updating was beneficial for two reasons. First, it avoided the recomputation of SD at each iteration, which reduced the overall computational cost. Second, the algorithm was able to converge faster if it could carry out several iterations at the beginning without low-dimensional information. Once an initial steep reduction in the cost function value was achieved, usually around the 10^{th} iteration, adding low-dimensional information helped to further improve the convergence. In the experiments, we noted that this was because in the first iterations with the constant SD, clusters of similar data points tend to form. Then, the low-dimensional information helps to reduce the cost function value by generating more white space between these clusters. This indicates that the low-dimensional information is more effective when the algorithm is in the neighborhood of a minimizer of the Kullback-Leibler divergence.

Interestingly, in the experiments, we observed that using low-dimensional information had a positive effect on LS, but not on TR. We attribute this to the degradation of the conditioning of the SD matrix that happened when we added low-dimensional information. When a matrix has a high condition number, it can take significantly more iterations for the CG solver to find a solution with the desired tolerance. In TR, the quality of this new iterate is used to decide whether the trust region should be enlarged, shrunk, or left unchanged. We found that after we updated the SD matrix, the quality of the new iterates was not of enough good quality, causing the radius to shrink. The progress of the algorithm would plateau once the radius reached a small enough value from which it was complicated to recover.

Scaling to datasets with hundreds of thousands of points. Despite their fast convergence, the quasi-Newton methods that we considered were not able to scale to large datasets due to several performance bottlenecks. We identified and addressed three of them, which yielded two fast variants: SD-LS-CG and SD-TR-CG. The most important source of slowdowns was the exact computation of the gradient and the cost function value, which takes $O(N^2)$ time. We used the Barnes-Hut algorithm to approximate these quantities

in $O(N \log N)$ and found that, with reasonable values of θ, these approximations did not hurt the convergence of the algorithms. The result indicates that quasi-Newton methods are more susceptible to gradient than to cost function approximations. We believe this is because, when using the SD Hessian approximation to find the new iterates, the errors in the gradient can get amplified, which in some cases generated bad iterates that increased the cost function value. In some cases, the algorithms could not recover from this, which resulted in the flatlining and eventual halting of the reduction of the cost function value.

The other two bottlenecks that we addressed have to do with the linear solvers that are used in the LS and TR frameworks. For the former, we found that using the Cholesky factorization is not viable due to its large setup times, which, when paired with the regular updating of the SD with low-dimensional information, offset the convergence gains. In general, the results indicate that the CG procedures in LS and TR can be truncated without affecting their convergence. With this bound on the computational complexity of the CG procedure, and using approximate gradients and cost function evaluations, we obtained two quasi-Newton methods: SD-LS-CG and SD-TR-CG, which have a complexity of $O(N \log N + K_{CG}\text{NNZ}_{\text{SD}})$ per iteration, where K_{CG} corresponds to the CG iteration threshold and NNZ_{SD} to the number of nonzero values of SD.

Quasi-Newton methods are a good alternative to BH-SNE for the visualization of high-dimensional data. Although the experimental results indicate that SD-LS-CG and SD-TR-CG are more effective at optimizing tSNE's cost function, it is important to highlight the nuisance in this conclusion. The reason why BH-SNE takes more time to converge is because of the initial plateau that arises as a result of using early exaggeration [54], which is needed to avoid local optima. Without this exaggeration phase, we observed that BH-SNE was able to reduce the cost function value in a similar or, in some cases, a smaller number of iterations than the quasi-Newton methods, at the expense of embeddings where clusters of similar data points where fragmented.

Interestingly, we observed that even without this exaggeration phase, the embeddings produced by the SD-powered quasi-Newton methods did not suffer significantly from the problem of cluster fragmentation. Furthermore, we found that, if we employed a more careful initialization like PCA, instead of a random one, we could obtain better-looking embeddings that had similar cost function values. A second phenomenon we observed in the resulting embeddings is related to the late exaggeration phase described in [28]. In the paper, the authors show how this helps to generate more space between the clusters of similar data points. As was mentioned before, we found that we could obtain a similar result in SD-LS-CG when we added low-dimensional information every 10 iterations.

To conclude, we believe that the quasi-Newton methods that we constructed are a good alternative to BH-SNE. Even if they all eventually converge to similar cost function values, the SD-based methods do not need the early exaggeration phase to generate well-separated clusters, avoiding the plateau that delays the optimization. Because of this, they can generate reasonable embeddings of the data in less time than BH-SNE which is a desirable trait in the progressive visual analytics workflow [48].

7.2. Limitations and Future Work

We believe that for our method to be widely applicable, certain limitations need to be addressed. In the experiments, we observed the potential of pairing SD with the TR framework. In contrast to LS, TR does not use a within-iteration iterative procedure to set the step size and instead updates the radius once per iteration based on the quality of the iterate, which tends to make it faster. Nevertheless, we found that it was not able to make use of low-dimensional information effectively as a result of the degradation of the conditioning of the SD matrix, which has the effect of increasing the number of CG iterations that are required to find a reasonable solution. We the TR framework could be used with the SD with low-dimensional information if we could find an effective preconditioner and by revising the threshold values that the algorithm uses to adjust the size of the TR radius, making it less sensitive to low-quality iterates.

A second aspect that needs further experimentation is whether other approximations of the gradient and the cost function values have a different effect on the robustness of the algorithms. This is important because, recently, faster approximations such as Flt-SNE and GPGPU tSNE have arisen. Given that they are conceptually different from BH, it is not clear whether the errors that they produce can affect quasi-Newton methods more severely, or in a different manner. A way to deal with this potential issue may be to add safeguards to LS and TR such as those presented in [30].

Finally, we observed that the iterative process of quasi-Newton methods that use SD can be divided into two phases. In the first one, a sudden drop in the cost function value takes place. This corresponds to the algorithms quickly approaching an optimizer. Then, in the second phase, once they are in the neighborhood

of the minimizer, the progress slows down. We believe different aspects of our method, such as the updating of SD with low-dimensional information and the thresholding of the number of CG iterations can be adapted to take advantage of this, potentiating the progress of the algorithms in the last stage.

A

The tSNE Gradient

In this chapter, we present a detailed derivation of the gradient of the Kullback-Leibler (KL) divergence, tSNE's cost function. As can be observed in Equation A.1, KL depends only on the embedding $Y \in \mathbb{R}^{Nd}$ (this is a flattened version of the $N \times d$ embedding matrix with N points and d coordinates), which is used to compute the low-dimensional matrix of affinities Q. The other input of KL, the matrix of high-dimensional affinities P, is constant given that it is computed using the high-dimensional points $X \in \mathbb{R}^{N \times D}$.

$$C(Y) = KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} = \sum_{i \neq j} p_{ij} \log p_{ij} - p_{ij} \log q_{ij} \tag{A.1}$$

Before deriving the gradient, we state some useful variables. First, Equation A.2 presents the expression for computing Q. The numerator corresponds to a Student's t-distributed kernel and the denominator to a normalization term that adds the value of this kernel over all pairs of points. Having this, the first supplementary variable that we define, presented in Equation A.3, d_{ij}, corresponds to the Euclidean distance between low-dimensional points. In this case $y_i \in \mathbb{R}^d$ gathers the two coordinates of the point. The second one, shown in Equation A.2, is the normalization term Z that is used when computing Q.

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l} (1 + ||y_k - y_l||^2)^{-1}} \tag{A.2}$$

$$d_{ij} = ||y_i - y_j|| \tag{A.3}$$

$$Z = \sum_{k \neq l} (1 + ||y_k - y_l||^2)^{-1} = \sum_{k \neq l} (1 + d_{kl}^2)^{-1} \tag{A.4}$$

To compute the gradient $\nabla C \in \mathbb{N}$, it suffices to compute the partial derivative of C with respect to a parameter y_i^c where $c \in \{0, 1\}$ corresponds to the coordinate c of point y_i. As can be observed in Equation A.5, we decompose this derivative in two terms, corresponding to the numerator and denominator of q_{ij}. Then, we use the chain rule on each of these to obtain an expression that is simpler to compute. Note that in the third line of the equation, we add a 2 to each term and change the indices from e and f to i and j. This reflects the fact that only the terms that involve the point y_i are considered in the derivative (the rest evaluate to zero). this reduces the number of comparisons to take into account from N^2 to $2N$. Since $d_{ij} = d_{ji}$, we write both groups of N terms into a single, common, one with a 2 upfront.

$$\begin{aligned}\frac{\partial C}{\partial y_i^c} &= \frac{\partial(\sum_{e\neq f} p_{ef}\log p_{ef} - p_{ef}\log q_{ef})}{\partial y_i^c} \\ &= \frac{\partial(\sum_{e\neq f} p_{ef}\log Z - p_{ef}\log(1+d_{ef}^2)^{-1})}{\partial y_i^c} \\ &= \frac{2\partial\log Z}{\partial Z}\frac{\partial Z}{\partial d_{ij}}\frac{\partial d_{ij}}{\partial y_i^c} - \frac{2\partial\sum_j p_{ij}\log(1+d_{ij}^2)^{-1}}{\partial d_{ij}}\frac{\partial d_{ij}}{\partial y_i^c} \\ &= 2(\frac{\partial\log Z}{\partial Z}\frac{\partial Z}{\partial d_{ij}} - \frac{\sum_j p_{ij}\partial\log(1+d_{ij}^2)^{-1}}{\partial d_{ij}})\frac{\partial d_{ij}}{\partial y_i^c}\end{aligned} \tag{A.5}$$

Having this expression for the gradient, we need to compute the individual partial derivatives. Starting with the common derivative of the two terms, $\partial d_{ij}/\partial y_i^c$, it corresponds to,

$$\begin{aligned}\frac{\partial d_{ij}}{\partial y_i^c} = \frac{\partial||y_i - y_j||}{\partial y_i^c} &= \frac{\partial(\sum_d (y_i^{(d)} - y_j^{(d)})^2)^{1/2}}{\partial y_i^c} \\ &= \frac{1}{2d_{ij}}(2(y_i^{(c)} - y_j^{(c)})) = \frac{(y_i^{(c)} - y_j^{(c)})}{d_{ij}}\end{aligned} \tag{A.6}$$

The next step is to compute the two terms of Equation A.5 which are surrounded by brackets. Starting with the one on the left, $\partial Z/\partial y_i$, the two relevant partial derivatives are presented below.

$$\frac{\partial\log Z}{\partial Z} = \frac{1}{Z} \tag{A.7}$$

$$\frac{\partial Z}{\partial d_{ij}} = \frac{\partial\sum_{k\neq l}(1+d_{kl}^2)^{-1}}{\partial d_{ij}} = \frac{\partial\sum_j(1+d_{ij}^2)^{-1}}{\partial d_{ij}} = -2\sum_j d_{ij}(1+d_{ij}^2)^{-2} \tag{A.8}$$

Moving to the term on the right of the difference in Equation A.5, we only need to take the derivative of $\log(1+d_{ij}^2)^{-1}$ with respect to d_{ij}. This is done as follows.

$$\frac{\sum_j p_{ij}\partial\log(1+d_{ij}^2)^{-1}}{\partial d_{ij}} = \sum_j p_{ij}\frac{-2d_{ij}(1+d_{ij}^2)^{-2}}{(1+d_{ij}^2)^{-1}} = -2\sum_j p_{ij}(1+d_{ij}^2)^{-1} \tag{A.9}$$

Finally, having these partial derivatives, we can obtain the gradient of tSNE by plugging Equations A.7, A.8, A.9, and A.6 in Equation A.5. Which yields the following expression for the gradient of KL.

$$\begin{aligned}\frac{\partial C}{\partial y_i^c} &= 2(\frac{\partial\log Z}{\partial Z}\frac{\partial Z}{\partial d_{ij}} - \frac{\partial\sum_{i\neq j} p_{ij}\log(1+d_{ij}^2)^{-1}}{\partial d_{ij}})\frac{\partial d_{ij}}{\partial y_i^c} \\ &= 2(\frac{-2\sum_j d_{ij}(1+d_{ij}^2)^{-2}}{Z} + 2\sum_j p_{ij}(1+d_{ij}^2)^{-1})\frac{(y_i^{(c)} - y_j^{(c)})}{d_{ij}} \\ &= 4\sum_j (p_{ij} - q_{ij})(1+d_{ij}^2)^{-1}(y_i^{(c)} - y_j^{(c)})\end{aligned} \tag{A.10}$$

B

The tSNE Hessian

In this chapter, we present a detailed derivation of the Hessian $\nabla^2 C(Y) \in \mathbb{R}^{Nd \times Nd}$ of the Kullback-Leibler (KL) divergence, tSNE's cost function. Note how, in contrast to the gradient $\nabla C(Y) \in \mathbb{R}^{Nd}$, which is a vector, the Hessian is a square matrix. An entry (i, j) of the Hessian, corresponds to the partial derivative of the i^{th} entry of the gradient with respect to the j^{th} variable. Particularly, recall the equation of the gradient that we derived in Appendix A,

$$\begin{aligned}\frac{\partial C}{\partial y_i^{(c)}} &= 4\sum_j (p_{ij} - q_{ij})(1 + d_{ij}^2)^{-1}(y_i^{(c)} - y_j^{(c)}) \\ &= 4\sum_j (p_{ij}(1 + d_{ij}^2)^{-1}(y_i^{(c)} - y_j^{(c)}) - (1 + d_{ij}^2)^{-2} Z^{-1}(y_i^{(c)} - y_j^{(c)}))\end{aligned} \tag{B.1}$$

where we assume $d = 2$ for simplicity (i.e. $c \in \{0, 1\}$) and $y^{(c)i}$ corresponds to the d^{th} coordinate of point y_i. We can avoid computing every entry of the Hessian by focusing on four cases:

1. $\partial^2 C / \partial {y_i^{(0)}}^2$: the same point and the same coordinate.
2. $\partial^2 C / \partial y_i^{(0)} y_i^{(1)}$: the same point, but different coordinate.
3. $\partial^2 C / \partial y_i^{(0)} y_j^{(0)}$: different points, but the same coordinate.
4. $\partial^2 C / \partial y_i^{(0)} y_j^{(1)}$: different points and different coordinate.

In the following sections, we present a detailed derivation of each of these cases. Then, we relate them with Laplacian-based Hessian expression introduced in [58]. Before continuing, we present some general derivatives that will be useful throughout the chapter to avoid cluttering the explanation.

B.1. Useful Variables and Derivatives

In this section we present some useful variables and their derivatives that we take from the terms in Equation B.1.

- Euclidean distance and its derivative with respect to the coordinate of a point:

$$d_{ij} = ||y_i - y_j|| \tag{B.2}$$

$$\frac{\partial d_{ij}}{\partial y_i^{(c)}} = \frac{(y_i^{(c)} - y_j^{(c)})}{d_{ij}} \qquad \frac{\partial d_{ij}}{\partial y_j^{(c)}} = -\frac{(y_i^{(c)} - y_j^{(c)})}{d_{ij}} \tag{B.3}$$

- Derivative of Student's t-Distributed kernel:

$$\frac{\partial (1 + d_{ij}^2)^{-1}}{\partial d_{ij}} = -2 d_{ij} (1 + d_{ij}^2)^{-2} \tag{B.4}$$

- Derivative of squared Student's t-Distributed kernel:

$$\frac{\partial(1+d_{ij}^2)^{-2}}{\partial d_{ij}} = -4d_{ij}(1+d_{ij}^2)^{-3} \tag{B.5}$$

- Derivative of normalization term Z:

$$\begin{aligned}\frac{\partial Z}{\partial d_{ij}} &= \frac{\partial \sum_{k\neq l}(1+d_{ij}^2)^{-1}}{\partial d_{ij}} \\ &= -4\sum_j d_{ij}(1+d_{ij}^2)^{-2}\end{aligned} \tag{B.6}$$

- Derivative of the inverse of Z:

$$\frac{\partial Z^{-1}}{\partial Z} = -Z^{-2} \tag{B.7}$$

B.2. Case 1: Same Point, Same Coordinate

The first case is presented in Equation B.8. We divide its computation in each of the individual partial derivatives inside the sum.

$$\frac{\partial^2 C}{\partial y_i^{(0)^2}} = 4\sum_j (p_{ij}\frac{\partial(1+d_{ij}^2)^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} - \frac{\partial(1+d_{ij}^2)^{-2}Z^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}}) \tag{B.8}$$

For the left term, we apply the product and chain rules, obtaining the following derivative,

$$\begin{aligned}\frac{\partial(1+d_{ij}^2)^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} &= \frac{\partial(1+d_{ij}^2)^{-1}}{\partial y_i^{(0)}}(y_i^{(0)}-y_j^{(0)}) + (1+d_{ij}^2)^{-1}\frac{\partial(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} \\ &= \frac{\partial(1+d_{ij}^2)^{-1}}{\partial d_{ij}}\frac{\partial d_{ij}}{\partial y_i^{(0)}}(y_i^{(0)}-y_j^{(0)}) + (1+d_{ij}^2)^{-1}\frac{\partial(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}}\end{aligned} \tag{B.9}$$

Plugging in the expressions that we derived in the previous section, we obtain,

$$\begin{aligned}\frac{\partial(1+d_{ij}^2)^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} &= -2d_{ij}(1+d_{ij}^2)^{-2}\frac{(y_i^{(0)}-y_j^{(0)})^2}{d_{ij}} + (1+d_{ij}^2)^{-1} \\ &= -2(1+d_{ij}^2)^{-2}(y_i^{(0)}-y_j^{(0)})^2 + (1+d_{ij}^2)^{-1}\end{aligned} \tag{B.10}$$

Moving to the right term inside the sum of Equation B.8, we can also use the product and chain rules to compute the derivative,

$$\begin{aligned}\frac{\partial(1+d_{ij}^2)^{-2}Z^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} &= \frac{\partial(1+d_{ij}^2)^{-2}}{\partial y_i^{(0)}}Z^{-1}(y_i^{(0)}-y_j^{(0)}) \\ &\quad + (1+d_{ij}^2)^{-2}\frac{\partial Z^{-1}}{\partial y_i^{(0)}}(y_i^{(0)}-y_j^{(0)}) \\ &\quad + (1+d_{ij}^2)^{-2}Z^{-1}\frac{\partial(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} \\ &= \frac{\partial(1+d_{ij}^2)^{-2}}{\partial d_{ij}}\frac{\partial d_{ij}}{\partial y_i^{(0)}}Z^{-1}(y_i^{(0)}-y_j^{(0)}) \\ &\quad + (1+d_{ij}^2)^{-2}\frac{\partial Z^{-1}}{\partial Z}\frac{\partial Z}{\partial d_{ij}}\frac{\partial d_{ij}}{\partial y_i^{(0)}}(y_i^{(0)}-y_j^{(0)}) \\ &\quad + (1+d_{ij}^2)^{-2}Z^{-1}\frac{\partial(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}}\end{aligned} \tag{B.11}$$

Plugging in the expressions that we derived in the previous section, we obtain,

$$\begin{aligned}\frac{\partial(1+d_{ij}^2)^{-2}Z^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(0)}} &= -4d_{ij}(1+d_{ij}^2)^{-3}\frac{(y_i^{(0)}-y_j^{(0)})^2}{d_{ij}}Z^{-1} \\ &+4(1+d_{ij}^2)^{-2}Z^{-2}(\sum_b d_{ib}(1+d_{ib}^2)^{-2}\frac{(y_i^{(0)}-y_b^{(0)})}{d_{ib}})(y_i^{(0)}-y_j^{(0)}) \\ &+(1+d_{ij}^2)^{-2}Z^{-1} \\ &= -4q_{ij}(1+d_{ij}^2)^{-2}(y_i^{(0)}-y_j^{(0)})^2 \\ &+4q_{ij}^2(\sum_b(1+d_{ib}^2)^{-2}(y_i^{(0)}-y_b^{(0)}))(y_i^{(0)}-y_j^{(0)}) \\ &+q_{ij}(1+d_{ij})^{-1}\end{aligned} \tag{B.12}$$

To obtain the expression for the first case, we can plug Equations B.10 and B.12 in B.8, which yields,

$$\begin{aligned}\frac{\partial^2 C}{\partial y_i^{(0)^2}} &= 4\sum_j(-2p_{ij}(1+d_{ij}^2)^{-2}(y_i^{(0)}-y_j^{(0)})^2+p_{ij}(1+d_{ij}^2)^{-1} \\ &+4q_{ij}(1+d_{ij}^2)^{-2}(y_i^{(0)}-y_j^{(0)})^2 \\ &-4q_{ij}^2(\sum_b(1+d_{ib}^2)^{-2}(y_i^{(0)}-y_b^{(0)}))(y_i^{(0)}-y_j^{(0)})-q_{ij}(1+d_{ij})^{-1}) \\ &= 4\sum_j((p_{ij}-q_{ij})(1+d_{ij})^{-1} \\ &-2(p_{ij}-2q_{ij})(1+d_{ij}^2)^{-2}(y_i^{(0)}-y_j^{(0)})^2 \\ &-4q_{ij}^2(\sum_b(1+d_{ib}^2)^{-2}(y_i^{(0)}-y_b^{(0)}))(y_i^{(0)}-y_j^{(0)}))\end{aligned} \tag{B.13}$$

Notice how the result of the first case is a sum. This is because all the terms of the sum in the gradient include the point y_i and therefore they do not evaluate to zero. When we compute the derivative of the gradient with respect to a coordinate of another point y_j, only one term of the sum will remain. Another thing to highlight is that in this case, because we were dealing with the same coordinate, the term $(y_i^{(0)}-y_j^{(0)})^2$ arose frequently. When we differentiate with respect to other coordinates, this will also change. We explore these issues in the following sections.

B.3. Case 2: Same Point, Different Coordinates

As can be observed in Equation B.14, this case is similar to the previous one. We are still taking the derivative with respect to the same point, y_i, but with a different coordinate.

$$\frac{\partial^2 C}{\partial y_i^{(0)}y_i^{(1)}} = 4\sum_j(p_{ij}\frac{\partial(1+d_{ij}^2)^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(1)}}-\frac{\partial(1+d_{ij}^2)^{-2}Z^{-1}(y_i^{(0)}-y_j^{(0)})}{\partial y_i^{(1)}}) \tag{B.14}$$

The change of coordinate does not significantly impact the final outcome, which we present in Equation B.15. There are two main things to note. First, the term $(y_i^{(0)}-y_j^{(0)})^2$ was replaced with $(y_i^{(0)}-y_j^{(0)})(y_i^{(1)}-y_j^{(1)})$. Second, in Equations B.9 and B.11, the derivative of $(y_i^{(0)}-y_j^{(0)})$ with respect to $y_i^{(0)}$ was equal to one. In this case, since we derivate with respect to $y_i^{(1)}$, the term $(p_{ij}-q_{ij})$ in the final expression ends cancelling out.

$$\begin{aligned}\frac{\partial^2 C}{\partial y_i^{(0)}y_i^{(1)}} &= 4\sum_j(2(p_{ij}-2q_{ij})(1+d_{ij}^2)^{-2}(y_i^{(0)}-y_j^{(0)})(y_i^{(1)}-y_j^{(1)}) \\ &-4q_{ij}^2(\sum_b(1+d_{ib}^2)^{-2}(y_i^{(1)}-y_b^{(1)}))(y_i^{(0)}-y_j^{(0)}))\end{aligned} \tag{B.15}$$

B.4. Case 3: Different Points, Same Coordinate

In contrast with the previous two cases, where we derivate the gradient with respect to the same point y_i (but different coordinate), we now focus on the cases where the point changes. In this section we derive the expression in Equation B.16.

$$\frac{\partial^2 C}{\partial y_i^{(0)} y_j^{(0)}} = 4\sum_j (p_{ij} \frac{\partial (1+d_{ij}^2)^{-1}(y_i^{(0)} - y_j^{(0)})}{\partial y_j^{(0)}} - \frac{\partial (1+d_{ij}^2)^{-2} Z^{-1}(y_i^{(0)} - y_j^{(0)})}{\partial y_j^{(0)}}) \tag{B.16}$$

The outcome of this case is similar to the first one with two important changes. First, since we are differentiating the gradient with respect to another point y_j, all the terms of the sum, except for one, will evaluate to zero. Second, the signs of all terms flip as a consequence of differentiating $(y_i^{(0)} - y_j^{(0)})$ and d_{ij} with respect to $y_j^{(0)}$. With this in mind, Equation B.17, presents the final expression for this case.

$$\begin{aligned} \frac{\partial^2 C}{\partial y_i^{(0)} y_j^{(0)}} &= 4(-(p_{ij} - q_{ij})(1+d_{ij})^{-1} \\ &\quad + 2(p_{ij} - 2q_{ij})(1+d_{ij}^2)^{-2}(y_i^{(0)} - y_j^{(0)})^2 \\ &\quad + 4q_{ij}^2 (\sum_b (1+d_{bj}^2)^{-2}(y_b^{(0)} - y_j^{(0)}))(y_i^{(0)} - y_j^{(0)})) \end{aligned} \tag{B.17}$$

B.5. Case 4: Different Points, Different Coordinate

The last case, presented in Equation B.18, corresponds to different points with different coordinates. As was the case in the previous section, the signs will be flipped, given that we are differentiating with respect to $y_j^{(1)}$. Also, like case 3, certain terms will get cancelled when we find the derivative of $(y_i^{(0)} - y_j^{(0)})$ with respect to a different coordinate of a point.

$$\frac{\partial^2 C}{\partial y_i^{(0)} y_j^{(1)}} = 4\sum_j (p_{ij} \frac{\partial (1+d_{ij}^2)^{-1}(y_i^{(0)} - y_j^{(0)})}{\partial y_j^{(1)}} - \frac{\partial (1+d_{ij}^2)^{-2} Z^{-1}(y_i^{(0)} - y_j^{(0)})}{\partial y_j^{(1)}}) \tag{B.18}$$

The expression in Equation B.19 presents the outcome of this case.

$$\begin{aligned} \frac{\partial^2 C}{\partial y_i^{(0)} y_j^{(1)}} &= -4(2(p_{ij} - 2q_{ij})(1+d_{ij}^2)^{-2}(y_i^{(0)} - y_j^{(0)})(y_i^{(1)} - y_j^{(1)}) \\ &\quad + 4q_{ij}^2 (\sum_b (1+d_{bj}^2)^{-2}(y_b^{(1)} - y_j^{(1)}))(y_i^{(0)} - y_j^{(0)})) \end{aligned} \tag{B.19}$$

B.6. A Laplacian-Based Expression for the Hessian

In the previous sections, we computed the derivatives of the different cases of the Hessian matrix. It is possible to observe how cumbersome it can be to determine these expressions, given that we need to perform several, error-prone, algebraic operations. These expressions are also not helpful for finding Hessian approximations that work within the Line Search and Trust Region frameworks because they do not give the larger picture. In [58], the authors introduced a Laplacian-bases expression of the Hessian, which we present in Equation B.20. It is equivalent to the expressions that we derived, but its easier to analyze. For instance, using properties of Laplacian matrices and the weights in Equation B.21, we can see how the positive semidefinite Laplacian with weights $w^+ = p_{ij}(1+d_{ij}^2)^{-1}$ could be safely used in an iterative optimization procedure. This is indeed the Spectral Direction approximation, which was the main focus of this work.

$$\nabla^2 C(Y) = 4L \otimes I_d + 8L^{yy} - 16\text{vec}(L^q Y)\text{vec}(L^q Y)^T \tag{B.20}$$

$$\begin{aligned} w_{ij} &= (p_{ij} - q_{ij}) t_{ij} \\ w_{ij}^q &= -q_{ij} t_{ij}^2 \\ w_{ni,mj}^{yy} &= -(p_{ij} - 2q_{ij})(y_i^{(n)} - y_j^{(n)})(y_i^{(m)} - y_j^{(m)}) t_{ij}^2 \end{aligned} \tag{B.21}$$

Bibliography

[1] Thomas D. Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 0, pages 239–256. Association for Computing Machinery, 2017. ISBN 9781611974782. doi: 10.1137/1.9781611974782.16. URL http://www.siam.org/journals/ojsa.php.

[2] Fevzi Alimoglu and Ethem Alpaydin. Methods of combining multiple classifiers based on different representations for pen-based handwritten digit recognition. *Proceedings of the Fifth Turkish Artificial Intelligence and Artificial Neural Networks Symposium*, pages 1–8, 1996. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.6383.

[3] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096): 446–449, 1986. ISSN 00280836. doi: 10.1038/324446a0. URL https://www.nature.com/articles/324446a0.

[4] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, 2003. ISSN 08997667. doi: 10.1162/089976603321780317.

[5] Anna C Belkina, Christopher O Ciccolella, Rina Anno, Richard Halpert, Josef Spidlen, and Jennifer E Snyder-Cappione. Automated optimized parameters for T-distributed stochastic neighbor embedding improve visualization and analysis of large datasets. *Nature Communications*, 10 (1), 2019. ISSN 20411723. doi: 10.1038/s41467-019-13055-y. URL https://doi.org/10.1038/s41467-019-13055-y.

[6] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, sep 1975. ISSN 15577317. doi: 10.1145/361002.361007. URL http://portal.acm.org/citation.cfm?doid=361002.361007.

[7] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. 2011. ISBN 9780123849885. doi: 10.1016/B978-0-12-384988-5.00006-1.

[8] Miguel Á Carreira-Perpiñán. The elastic embedding algorithm for dimensionality reduction. *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, pages 167–174, 2010.

[9] David M. Chan, Roshan Rao, Forrest Huang, and John F. Canny. T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data. In *Proceedings - 2018 30th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2018*, pages 330–338. Institute of Electrical and Electronics Engineers Inc., feb 2019. ISBN 9781538677698. doi: 10.1109/CAHPC.2018.8645912. URL http://arxiv.org/abs/1807.11824.

[10] James Cook, Ilya Sutskever, Andriy Mnih, and Geoffrey Hinton. Visualizing similarity data with a mixture of maps. In *Journal of Machine Learning Research*, volume 2, pages 67–74, 2007. URL www.kyb.tuebingen.mpg.de/bs/people/carl/code/minimize/.

[11] Sanjoy Dasgupta and Yoav Freund. *Random Projection Trees and Low Dimensional Manifolds*. 2008. ISBN 9781605580470.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. pages 248–255. Institute of Electrical and Electronics Engineers (IEEE), mar 2010. doi: 10.1109/cvpr.2009.5206848.

[13] Wei Dong, Moses Charikar, and Kai Li. Efficient K-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011*, pages 577–586, 2011. ISBN 9781450306324. doi: 10.1145/1963405.1963487.

[14] Nicholas I.M. Gould, Dominique Orban, Annick Sartenaer, and Phillipe L Toint. Sensitivity of trust-region algorithms to their parameters. *4OR*, 3(3):227–241, 2005. ISSN 16142411. doi: 10.1007/s10288-005-0065-y.

[15] Alexander G Gray and Andrew W Moore. 'N-Body' problems in statistical learning. In *Advances in Neural Information Processing Systems*, 2001. ISBN 0262122413.

[16] Geoffrey Hinton and Sam Roweis. Stochastic neighbor embedding. *Advances in Neural Information Processing Systems*, 2003. ISSN 10495258.

[17] Geoffrey Hinton and Ruslan Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, jul 2006. ISSN 00368075. doi: 10.1126/science.1127647. URL http://arxiv.org/abs/physics/0601055www.sciencemag.org/cgi/content/full/313/5786/502/DC1.

[18] Thomas Höllt, Nicola Pezzotti, Vincent Van Unen, Frits Koning, Elmar Eisemann, Boudewijn P.F. Lelieveldt, and Anna Vilanova. Cytosplore: Interactive Immune Cell Phenotyping for Large Single-Cell Datasets. *Computer Graphics Forum*, 35(3):171–180, 2016. ISSN 14678659. doi: 10.1111/cgf.12893.

[19] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, sep 1933. ISSN 00220663. doi: 10.1037/h0071325. URL /record/1934-00645-001.

[20] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1 (4):295—-307, 1988. URL papers://d471b97a-e92c-44c2-8562-4efc271c8c1b/Paper/p602.

[21] Jeff Johnson, Matthijs Douze, and Herve Jegou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, pages 1–1, jun 2019. ISSN 2332-7790. doi: 10.1109/tbdata.2019.2921572.

[22] Minjeong Kim, Minsuk Choi, Sunwoong Lee, Jian Tang, Haesun Park, and Jaegul Choo. PixelSNE: Visualizing Fast with Just Enough Precision via Pixel-Aligned Stochastic Neighbor Embedding. 2016. doi: 10.475/123. URL http://arxiv.org/abs/1611.02568.

[23] Dmitry Kobak and Philipp Berens. The art of using t-SNE for single-cell transcriptomics. 2018. doi: 10.1101/453449. URL https://doi.org/10.1101/453449.

[24] Dmitry Kobak, George Linderman, Stefan Steinerberger, Yuval Kluger, and Philipp Berens. Heavy-Tailed Kernels Reveal a Finer Cluster Structure in t-SNE Visualisations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11906 LNAI, pages 124–139, 2020. ISBN 9783030461492. doi: 10.1007/978-3-030-46150-8_8.

[25] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT Compiler. 2015. doi: 10.1145/2833157.2833162. URL http://dx.doi.org/10.1145/2833157.2833162.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86 (11):2278–2323, 1998. ISSN 00189219. doi: 10.1109/5.726791. URL https://ieeexplore.ieee.org/abstract/document/726791/?casa{_}token=i9wmB-4XKJIAAAAA:6AIHToodE9z42jjxWbth4uMeJtiaiolGbJ8gZAhATzLb6OBH9dIsROxsM2kVKqeaDo5kf9dXEg.

[27] George Linderman and Stefan Steinerberger. Clustering with t-SNE, Provably. *SIAM Journal on Mathematics of Data Science*, 1(2):313–332, 2019. doi: 10.1137/18m1216134. URL http://www.siam.org/journals/simods/1-2/M121613.html.

[28] George Linderman, Manas Rachh, Jeremy G. Hoskins, Stefan Steinerberger, and Yuval Kluger. Fast interpolation-based t-SNE for improved visualization of single-cell RNA-seq data. *Nature Methods*, 16 (3):243–245, dec 2019. ISSN 15487105. doi: 10.1038/s41592-018-0308-4. URL http://arxiv.org/abs/1712.09005http://dx.doi.org/10.1038/s41592-018-0308-4.

[29] Laurens Van Der Maaten. Fast Optimization for t-SNE. *Neural Information Processing Systems (NIPS) 2010 Workshop on Challenges in Data Visualization*, 1(1):1–5, 2010. URL http://homepage.tudelft.nl/19j49/t-SNE.http://homepage.tudelft.nl/19j49/Publications{_}files/nips2010.pdf.

[30] Maren Mahsereci and Philipp Hennig. Probabilistic line searches for stochastic optimization. *Journal of Machine Learning Research*, 18:1–59, feb 2017. ISSN 15337928. URL http://arxiv.org/abs/1502.02846.

[31] Leland McInnes, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. 2018. URL http://arxiv.org/abs/1802.03426.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations ofwords and phrases and their compositionality. Technical report, 2013.

[33] Kijoeng Nam, Hongmo Je, and Seungjin Choi. Fast stochastic neighbor embedding: A trust-region algorithm. Technical report, 2004.

[34] Sameer A Nene, Shree K Nayar, and Hiroshi Murase. Columbia Object Image Library (COIL-100). Technical report.

[35] Sameer A Nene, Shree K Nayar, and Hiroshi Murase. Columbia Object Image Library (COIL-20). Technical report, 1996. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.5914.

[36] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science \& Business Media, 2006. ISBN 9780387303031. doi: 10.1007/0-387-33477-7.

[37] Mohammad Norouzi, Ali Punjani, and David J. Fleet. Fast search in Hamming space with multi-index hashing. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3108–3115, 2012. ISBN 9781467312264. doi: 10.1109/CVPR.2012.6248043.

[38] Olga I. Ornatsky, Robert Kinach, Dmitry R. Bandura, Xudong Lou, Scott D. Tanner, Vladimir I. Baranov, Mark Nitz, and Mitchell A. Winnik. Development of analytical methods for multiplex bio-assay with inductively coupled plasma mass spectrometry. *Journal of Analytical Atomic Spectrometry*, 23(4): 463–469, 2008. ISSN 02679477. doi: 10.1039/b710510j. URL https://pubs.rsc.org/--/content/articlehtml/2008/ja/b710510j.

[39] Fabian Pedregosa, Olivier Grisel, Ron Weiss, Alexandre Passos, Matthieu Brucher, Gael Varoquax, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier\ Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and Matthieu Brucher. Scikit-learn: Machine Learning in Python. Technical report, 2011. URL http://scikit-learn.sourceforge.net.

[40] Nicola Pezzotti, Thomas Höllt, Boudewijn P.F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. Hierarchical Stochastic Neighbor Embedding. *Computer Graphics Forum*, 35(3):21–30, 2016. ISSN 14678659. doi: 10.1111/cgf.12878.

[41] Nicola Pezzotti, Boudewijn P.F. Lelieveldt, Laurens Van Der Maaten, Thomas Höllt, Elmar Eisemann, and Anna Vilanova. Approximated and user steerable tSNE for progressive visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 23(7):1739–1752, jul 2017. ISSN 10772626. doi: 10.1109/TVCG.2016.2570755.

[42] Nicola Pezzotti, Thomas Höllt, Jan Van Gemert, Boudewijn P.F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. DeepEyes: Progressive Visual Analytics for Designing Deep Neural Networks. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):98–108, jan 2018. ISSN 10772626. doi: 10.1109/TVCG.2017.2744358.

[43] Nicola Pezzotti, Julian Thijssen, Alexander Mordvintsev, Thomas Höllt, Baldur Van Lew, Boudewijn P.F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. GPGPU Linear Complexity t-SNE Optimization. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1172–1181, 2020. ISSN 19410506. doi: 10.1109/TVCG.2019.2934307. URL http://arxiv.org/abs/1805.10817.

[44] Elijah Polak and Gerard Ribiere. Note sur la convergence de méthodes de directions conjuguées. Technical report, 1969. URL http://www.numdam.org/conditionshttp://www.numdam.org/legal.php.

[45] Sam Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, dec 2000. ISSN 00368075. doi: 10.1126/science.290.5500.2323. URL https://science.sciencemag.org/content/290/5500/2323https://science.sciencemag.org/content/290/5500/2323.abstract.

[46] Annick Sartenaer. Automatic determination of an initial trust region in nonlinear programming. *SIAM Journal of Scientific Computing*, 18(6):1788–1803, 1997. ISSN 10648275. doi: 10.1137/S1064827595286955.

[47] Trond Steihaug. The Conjugate Gradient Method and Trust Regions in Large Scale Optimization. *SIAM Journal on Numerical Analysis*, 20(3):626–637, jun 1983. ISSN 0036-1429. doi: 10.1137/0720042.

[48] Charles D. Stolper, Adam Perer, and David Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12): 1653–1662, 2014. ISSN 10772626. doi: 10.1109/TVCG.2014.2346574.

[49] John Strain. The Fast Gauss Transform with Variable Scales. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1131–1139, 1991. ISSN 0196-5204. doi: 10.1137/0912059. URL `http://www.siam.org/journals/ojsa.php`.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 07-12-June, pages 1–9. IEEE Computer Society, oct 2015. ISBN 9781467369640. doi: 10.1109/CVPR.2015.7298594. URL `https://arxiv.org/abs/1409.4842v1`.

[51] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. Visualizing large-scale and high-dimensional data. *25th International World Wide Web Conference, WWW 2016*, pages 287–297, 2016. doi: 10.1145/2872427.288304l.

[52] Joshua Tenenbaum, Vin De Silva, and John Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, dec 2000. ISSN 00368075. doi: 10.1126/science.290.5500.2319. URL `https://science.sciencemag.org/content/290/5500/2319https://science.sciencemag.org/content/290/5500/2319.abstract`.

[53] Dmitry Ulyanov. Multicore-tsne. *GitHub Repos. GitHub*, 2016.

[54] Laurens Van Der Maaten. Visualizing Data using t-SNE Laurens. *New Scientist*, 2008.

[55] Laurens Van Der Maaten. Accelerating t-SNE using tree-based algorithms. *Journal of Machine Learning Research*, 15:3221–3245, 2015. ISSN 15337928.

[56] Laurens Van der Maaten, Eric Postma, and Jaap Van den Herik. Dimensionality reduction: A comparative review; Tilburg University Centre for Creative Computing, Technical Report TiCC-TR 2009-005. 2009. URL `http://www.uvt.nl/ticc{%}0Ahomepage.tudelft.nl/19j49/Matlab{_}Toolbox{_}for{_}Dimensionality{_}Reduction{_}files/TR{_}Dimensiereductie.pdf`.

[57] Vincent Van Unen, Thomas Höllt, Nicola Pezzotti, Na Li, Marcel J.T. Reinders, Elmar Eisemann, Frits Koning, Anna Vilanova, and Boudewijn P.F. Lelieveldt. Visual analysis of mass cytometry data by hierarchical stochastic neighbour embedding reveals rare cell types. *Nature Communications*, 8 (1):1–10, dec 2017. ISSN 20411723. doi: 10.1038/s41467-017-01689-9. URL `www.nature.com/naturecommunications`.

[58] Max Vladymyrov and Miguel Á Carreira-Perpiñán. Partial-Hessian strategies for fast learning of nonlinear embeddings. *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*, 1: 345–352, 2012.

[59] Max Vladymyrov and Miguel Á Carreira-Perpiñán. Entropic affinities: Properties and efficient numerical computation. *30th International Conference on Machine Learning, ICML 2013*, (PART 2):1514–1522, 2013.

[60] Max Vladymyrov and Miguel Á Carreira-Perpiñán. Linear-time training of nonlinear low-dimensional embeddings. *Journal of Machine Learning Research*, 33:968–977, 2014. ISSN 15337928.

[61] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. Technical report, 2017. URL `http://arxiv.org/abs/1708.07747`.

[62] Lei Xu, Adam Krzyżak, and Ching Y. Suen. Methods of Combining Multiple Classifiers and Their Applications to Handwriting Recognition. *IEEE Transactions on Systems, Man and Cybernetics*, 22(3):418–435, 1992. ISSN 21682909. doi: 10.1109/21.155943. URL https://ieeexplore.ieee.org/abstract/document/155943/?casa{_}token=jFMoNlkbUlAAAAAA:0rXJadIoztg{_}oZ4L3sfmzKUMI2W1KTYiV1cJCWDTJb8EstsSn79942zVRKbgKR--0{_}-u6kQyng.

[63] Peter N Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. Technical report.

[64] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-Scale Bound-Constrained Optimization. *ACM Transactions on Mathematical Software*, 23(4):550–560, dec 1997. ISSN 00983500. doi: 10.1145/279232.279236. URL http://portal.acm.org/citation.cfm?doid=279232.279236.