GPU Accelerated T-Distributed Stochastic Neighbor Embedding

David Chan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-89
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-89.html

May 28, 2020

Acknowledgement

I would like to express my deepest gratitude towards my advisor Professor John Canny, for his support of my work, the second reader of my thesis, Professor James Demmel, and my research collaborators Roshan Rao and Forrest Huang for their contributions to the original tsnecuda algorithm. Without them, this would not have been possible. I would also like to thank the other members of the CannyLab for their contribution and feedback during the design process of the algorithm.

Finally, I would like to express my gratitude to my family and friends, who without their support, this work would not have been possible.

This work has been supported by the Berkeley Institute of Artificial Intelligence, and Berkeley Deep Drive.

GPU Accelerated T-Distributed Stochastic Neighbor Embedding

by David McCloud Chan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John F. Canny
Research Advisor

May 28, 2020

(Date)

* * * * * * *

James Demmel

Professor James Demmel
Second Reader

May 28, 2020

(Date)

Abstract

Modern datasets and models are notoriously difficult to explore and analyze due to their inherent high dimensionality and massive numbers of samples. Existing visualization methods which employ dimensionality reduction to two or three dimensions are often inefficient and/or ineffective for these datasets. For example, T-Distributed Neighbor Embedding (T-SNE) is a popular technique for dimensionality reduction, and visualization of high dimensional point structures, however T-SNE is an inherently slow algorithm, requiring pairwise computation between each of the points in high dimension. This thesis explores GP-GPU accelerated algorithms for approximate T-SNE, and demonstrates multiple algorithms achieving state of the art performance on, and novel visualizations of, common machine learning datasets.

To all of my family and friends

Contents

List of Figures

Acknowledgments

I would like to express my deepest gratitude towards my advisor Professor John Canny, for his support of my work, the second reader of my thesis, Professor James Demmel, and my research collaborators Roshan Rao and Forrest Huang for their contributions to the original tsnecuda algorithm. Without them, this would not have been possible. I would also like to thank the other members of the CannyLab for their contribution and feedback during the design process of the algorithm, and being excellent beta testers for tsnecuda in their own development.

Finally, I would like to express my gratitude to my family and friends, who without their support, this work would not have been possible.

This work has been supported by the Berkeley Institute of Artificial Intelligence, and Berkeley Deep Drive, as well as NVIDIA, which has donated some of the GPUs used in this research.

Chapter 1

Introduction

The recent emergence of high-dimensional data, from large-scale datasets and network activations to state spaces in robotics, has been a major factor contributing to advances in the areas of Machine Learning and Artificial Intelligence. While researchers have developed numerous methods for visualizing medium-sized data-sets, such visualizations are often inefficient or ineffective for high-dimensional or large-scale data. This inefficiency in visualization leads to major bottlenecks in a data scientist's research pipeline. Because developing conceptual understandings of the global and local structures of these datasets is vital for successfully developing and improving models, we introduce a fully GPU-based implementation of t-Distributed Stochastic Neighbor Embedding which allows researchers to explore structure in high-dimensional data efficiently and reduce the burden of forming understandings of the data and models in modern-day machine learning tasks.

t-Distributed Stochastic Neighbor Embedding (t-SNE) [1] is a dimensionality reduction method that has recently gained traction in the deep learning community for visualizing model activations and original features of datasets. t-SNE attempts to preserve the local structure of data by matching pairwise similarity distributions in both the higher-dimensional original data space and the lower-dimensional projected space. As opposed to methods such as PCA which attempt to preserve squared distances in the data, t-SNE has been shown to generate interesting low-dimensional clusters of data faithful to the distributions in the original data-space by preserving proximity in the data directly [1].

Unfortunately, current t-SNE implementations are inefficient for visualizing large-scale datasets. All current publicly available implementations execute on the CPU and can require large amounts of time to operate on even modest-sized data; running t-SNE on larger datasets can be extremely expensive. In this work, we introduce tsnecuda, an optimized implementation of the t-SNE algorithm on the GPU. By taking advantage of the natural parallelism in the algorithm, as well as techniques designed for computing the n-body problem, tsnecuda scales the t-SNE algorithm to large-scale vision datasets such as ImageNet [2]. In addition to scaling to larger datasets, t-SNE-CUDA provides a quality of life improvement for researchers who wish to iterate rapidly on smaller data.

Chapter 2

Background and Preliminaries

In this section, we introduce the background and preliminary information that a reader needs to understand the acceleration techniques that we have used in tsnecuda. In section 2.1 we cover the t-SNE algorithm in-depth, walking through the steps that the original algorithm implements. Then, in Section 2.2 we discuss how some CPU-based methods have accelerated the t-SNE algorithm. Finally, in Sections 2.3 we discuss some followup works to tsnecuda that we use for comparison and analysis.

2.1 T-Distributed Stochastic Neighbor Embedding

2.1.1 What is t-SNE? Why should it be fast?

t-Distributed Stochastic Neighbor Embedding (t-SNE) [1] is a non-parametric technique for dimensionality reduction which is well suited to the visualization of high dimensional datasets. Visualizing such large dimensional data can be extremely important in domains such as natural language processing [3], image processing for breast-cancer detection [4], spatio-temporal video data analysis ([5], [6]), and many others. DeCAF [7], DeVise [8] and other tools [9] all use t-SNE to help understand the activation space of deep convolutional networks. In many of these works, the analysis was restricted by the performance of the t-SNE algorithm, and thus researchers could only analyze subsets of the data or projections of the data into smaller spaces. Our work allows for complete visualizations at the scale required by these papers.

Another potential application for fast t-SNE is low-latency, interactive visualization of neural networks. Such visualizations have been shown by [10] to increase the productivity of data scientists, and several previous works have explored using t-SNE for such active and interactive visualization. [11] suggests using t-SNE for progressive visual analysis of deep neural networks while [12] suggests t-SNE as a method for increasing user involvement in the training process of DNNs. While we do not deeply explore applications of tsnecuda to this field of visualization, we believe that it is intriguing and exciting future work, as tsnecuda is fast enough to visualize training-time embeddings in real-time.

2.1.2 The t-SNE Algorithm

Dimensionality reduction techniques convert a high-dimensional dataset $\mathcal{X} = \{x_1, \ldots, x_n\}$ into a two or three dimensional dataset $\mathcal{Y} = \{y_1, \ldots, y_n\}$ that can be displayed on a traditional scatter-plot. The goal of such dimensionality reduction is to preserve as much information as possible from the high-dimensional data in the low-dimensional map. A significant number of techniques have been proposed to do such dimensionality reduction with classical methods such as principal component analysis [13], [14], multi-dimensional scaling [15], and more modern methods such as Sammon mapping [16], curvilinear component analysis [17], stochastic neighbor embedding [18], Isomap [19], maximum variance unfolding (MVU) [20], locally linear embedding (LLE) [21], laplacian eigenmaps [22], randomized PCA [14], Johnson-Lindenstrauss Embedding [23], and UMAP [24]. We redirect interested readers to Van der Maaten's and Arora *et al.* 's works [1], [25] for a thorough comparison between t-SNE and these visualization methods.

t-SNE begins by converting high-dimensional distances between data points into conditional probabilities representing the conditional probability $p_{j|i}$ that some point x_i would pick x_j as its neighbor in the space. This conditional probability is obtained by centering a Gaussian distribution at the point x_i:

$$p_{j|i} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-||x_i - x_k||^2/2\sigma_i^2)} \tag{2.1}$$

Where σ_i is the variance of the Gaussian centered at x_i, and $||\cdot||$ is the distance metric. In all cases $p_{i|i} = 0$, since we are interested in modeling neighborhood relationships. t-SNE is heavily based on Stochastic Neighbor Embedding (SNE) [18], which for any set of points in the lower-dimensional space, defines a similar probability:

$$q_{j|i} = \frac{\exp(-||y_i - y_j||^2)}{\sum_{k\neq i}\exp(-||y_i - y_k||^2)} \tag{2.2}$$

Notice that we have only defined $p_{j|i}$, the conditional distribution and not the joint distribution p_{ij}. A reasonable looking way to define the joint distribution is:

$$p_{ij} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k\neq l}\exp(-||x_k - x_l||^2/2\sigma_i^2)} \tag{2.3}$$

however this can cause problems when a high-dimensional data point x_i is an outlier: the values of p_{ij} are very small for all j, meaning the the low-dimensional point y_j has little effect on the cost function (and thus, the point x_j is not well represented). To solve this problem, t-SNE defines the joint probabilities to be symmetrized conditional probabilities:

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2n} \tag{2.4}$$

In SNE, we symmetrize $q_{j|i}$ to q_{ij} naturally (In t-SNE, we use the distribution in Equation 2.7):

$$q_{ij} = \frac{\exp(-||y_i - y_j||^2)}{\sum_{k \neq l} \exp(-||y_k - y_l||^2)} \tag{2.5}$$

The overall goal of SNE (and t-SNE) is to minimize the Kullback–Leibler divergence (KL Divergence, [26]) between the two distributions, p_{ij} and the distribution q_{ij} by minimizing

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{2.6}$$

with respect to the positions of the points in the lower dimensional space.In doing so, the algorithm moves the points around in the lower dimensional space until they are arranged in such a way that the likelihood that two points are close in the low dimensional space is similar to the likelihood that two points are close in the high dimensional space.

The peaked nature of the distribution means that local space is modeled more accurately than global space. Indeed, t-SNE is very good at determining if there is a cluster-like structure in a high-dimensional space, however, it can struggle with global relationships between clusters.

So far, we have kept our discussion to SNE. Unfortunately, while using a Gaussian in the lower dimensional space, such as SNE does, is attractive, it has several issues discussed in Maaten and Hinton [1] including a difficult to optimize cost function, and the "crowding problem", where clusters of points are pushed together in the low dimensional space because the long-range repulsive forces fall off too quickly (See [1] for a full discussion of the crowding problem). To solve these issues, instead of using a Gaussian distribution which falls off quickly, t-SNE instead uses a long-tailed Student t-distribution with one degree of freedom (a Cauchy distribution) in the low-dimensional space. Thus, we define the joint probabilities:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l} (1 + ||y_k - y_l||^2)^{-1}} \tag{2.7}$$

To optimize Equation 2.6, we use standard gradient descent on the gradient (derived as Equation 5 in Maaten and Hinton [1]):

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1} \tag{2.8}$$

$$y_i^{t+1} = y_i^t + \alpha \frac{\partial C}{\partial y_i} \tag{2.9}$$

where α is a learning rate that is tunable by the user.

The overall algorithm for t-SNE is given in Algorithm 1.

Algorithm 1 Naive $O(N^2)$ T-SNE Algorithm

```
Input (X): N × d−dimensional array of data
Output (Y): N × 2−dimensional projection
1: Compute pairwise distances of the input $\mathcal{X}$
2: Use pairwise distances to compute $p_{ij}$ according to Equation 2.4
3: Randomly initialize a set of $N$ points $\mathcal{Y} \in \mathbb{R}^2$
4: for $i = 1$ to convergence do
5:     Compute gradients for each point according to Equation 2.8
6:     Apply gradients to $\mathcal{Y}$ according to Equation 2.9
7: end for
8: return $\mathcal{Y}$
```

2.2 Accelerated t-SNE

While t-SNE is an excellent algorithm for generating low-dimensional representations, its naive algorithm can be computationally expensive. This begins on the first line of Algorithm 1, where to compute the conditional probabilities $p_{j|i}$ we need to compute the pairwise distances between elements in the high-dimensional space. This step costs $O(N^2k)$ with N points in k dimensions. Further, in each iteration of the gradient descent, we have to perform an additional $O(N^2)$ operation when calculating the gradients. In this section, we explore some methods that have been proposed to accelerate the internal steps of the t-SNE algorithm on the CPU.

The traditional gradient from Equation 2.8 can be re-written using a normalization constant:

$$\frac{\partial C}{\partial y_i} = 4\sum_{j\neq i}(p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1} = 4\sum_{j\neq i}(p_{ij} - q_{ij})q_{ij}Z(y_i - y_j) \tag{2.10}$$

where

$$Z = \sum_{k\neq l}(1 + ||y_k - y_l||^2)^{-1} \tag{2.11}$$

Van Der Maaten [27] introduces a new way of thinking about the t-SNE gradient from Equation 2.10, splitting the gradient into an ”Attractive force” and a ”Repulsive force” [1]:

$$\frac{\partial C}{\partial y_i} = 4(F_{attr}(i) + F_{rep}(i)) = 4\left(\sum_{j\neq i} p_{ij}q_{ij}Z(y_i - y_j) - \sum_{j\neq i} q_{ij}^2 Z(y_i - y_j)\right) \tag{2.12}$$

Both the computation of the attractive forces and the repulsive forces are $O(N^2)$ operations, however in the next few sections we will explore how to efficiently approximate these $O(N^2)$ operations.

[1] See Van Der Maaten [27] for a full derivation

2.2.1 Approximating the Attractive Forces with Nearest Neighbors

We begin by looking closely at the attractive forces:

$$F_{attr}(i) = \sum_{j \neq i} p_{ij} q_{ij} Z(y_i - y_j) \tag{2.13}$$

To make Equation 2.13 more efficient, we look at the computation of p_{ij}. Because p_{ij} is a Gaussian distribution with limited support, it is possible to approximate p_{ij} as :

$$p_{j|i} = \begin{cases} \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \in \mathcal{N}_i} \exp(-||x_k - x_l||^2/2\sigma_i^2)} & \text{if } j \in \mathcal{N}_i \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2n} \tag{2.15}$$

where $\mathcal{N}_i$ are a set of nearest-neighbors for the point i. While somewhat efficient, better approximations might threshold the Gaussian density, or make use of the gradient of the density function. These methods would avoid computing far-field interactions when points have relatively few neighbors, and improve accuracy in dense clusters, however these methods require adapted nearest neighbor techniques which are not compatible with current off the shelf efficient nearest neighbor methods.

Applying the nearest neighbors approximation, our new F_{attr} is

$$F_{attr}(i) = \sum_{j \in \mathcal{N}_i} p_{ij} q_{ij} Z(y_i - y_j) \tag{2.16}$$

Notice that $q_{ij}Z = (1 + ||y_i - y_j||^2)^{-1}$ can be computed in $O(1)$ time. Letting $|\mathcal{N}_i| = u$, this reduces the computational complexity of the iteration to $O(uN)$ where u is the number of neighbors, a significant improvement to the performance of the algorithm.

In Van Der Maaten [27], the authors used a vantage-point tree to compute the k-nearest neighbors in $O(N \log N)$ time. While this works well for low dimensional spaces, vantage point trees suffer from the curse of dimensionality, meaning that as the dimensionality of the input space increased, the complexity of computing the nearest neighbors approaches $O(N^2)$ [28]. This means that computing the exact nearest neighbors is intractable for modern datasets. Dimitriadis [29] addresses this problem by parallelizing the brute-force nearest neighbors search on the GPU, however even with the improved efficiency, the model is unable to handle large numbers of points due to the quadratic scaling factor.

Often, to solve problems with the curse of dimensionality, users apply another dimensionality reduction technique, such as PCA, to first reduce the input to a suitable dimension for t-SNE algorithms [27]. Shah and Silwal [30] argues that by reducing the dimensionality first using PCA, there is little loss of information, however, their results are limited to the simple MNIST dataset, the quality of their embeddings are difficult to evaluate in the limited data setting. Truly, the PCA reduction approach is unsatisfying. It reduces the amount of information available to the dimensionality reduction algorithm while introducing bias along the dimensions of highest variance.

To make use of the full data, Pezzotti, Lelieveldt, Maaten, *et al.* [31] address the curse of dimensionality by employing approximate nearest neighbors using a forest of randomized Kd trees to compute approximate nearest neighbors for the t-SNE algorithm in a steerable manner to emphasize points users deemed important. Their implementation is based on the FLANN [32] library. While this can be an effective technique for guiding t-SNE visualizations, it requires user input to be efficient, and due to the inefficient FLANN implementation, the technique still runs into dimension scaling issues. While this method works, randomized Kd trees still suffer significantly from the curse of dimensionality, and additional performance can be gained by using modern techniques for approximate nearest neighbors. We describe the approach that tsnecuda takes to compute approximate nearest neighbors in Section 3.1.

2.2.2 Approximating Repulsive Forces

While we can easily approximate the attractive forces with an $O(uN)$ algorithm (where u is the number of neighbors, and N is the number of points), approximating the repulsive forces given by

$$F_{rep}(i) = \sum_{j \neq i} q_{ij}^2 Z(y_i - y_j) \tag{2.17}$$

is much more difficult, due to the long-tailed Cauchy distribution. While the Gaussian distribution falls off exponentially, t he Cauchy distribution does not, meaning that we cannot merely use an approximate nearest neighbor method.

2.2.2.1 Barnes-Hut t-SNE

One method of approximating the repulsive forces is discussed in Van Der Maaten [27], which uses the Barnes-Hut algorithm [33] to approximate the long-distance forces, as Z (Equation 2.16) is an inverse square function.

At each iteration of Barnes-Hut t-SNE [27], the lower dimensional points $\mathcal{Y}$ are placed in a quad tree, diving the points spatially (See Figure 2.1). Then, for each point y_i a depth-first-search is performed on the quad tree. When looking at a quad tree cell centered at y_{cell} with radius r_{cell}, the following condition is evaluated:

$$\frac{r_{cell}}{||y_i - y_{cell}||} < \theta \tag{2.18}$$

If this evaluates to true, or the cell contains one or fewer points, then the cell is deemed far enough away to be used as a summary of the forces for all children and the recursion halts. θ is a parameter that controls the accuracy of the approximation with $\theta = 0$ giving

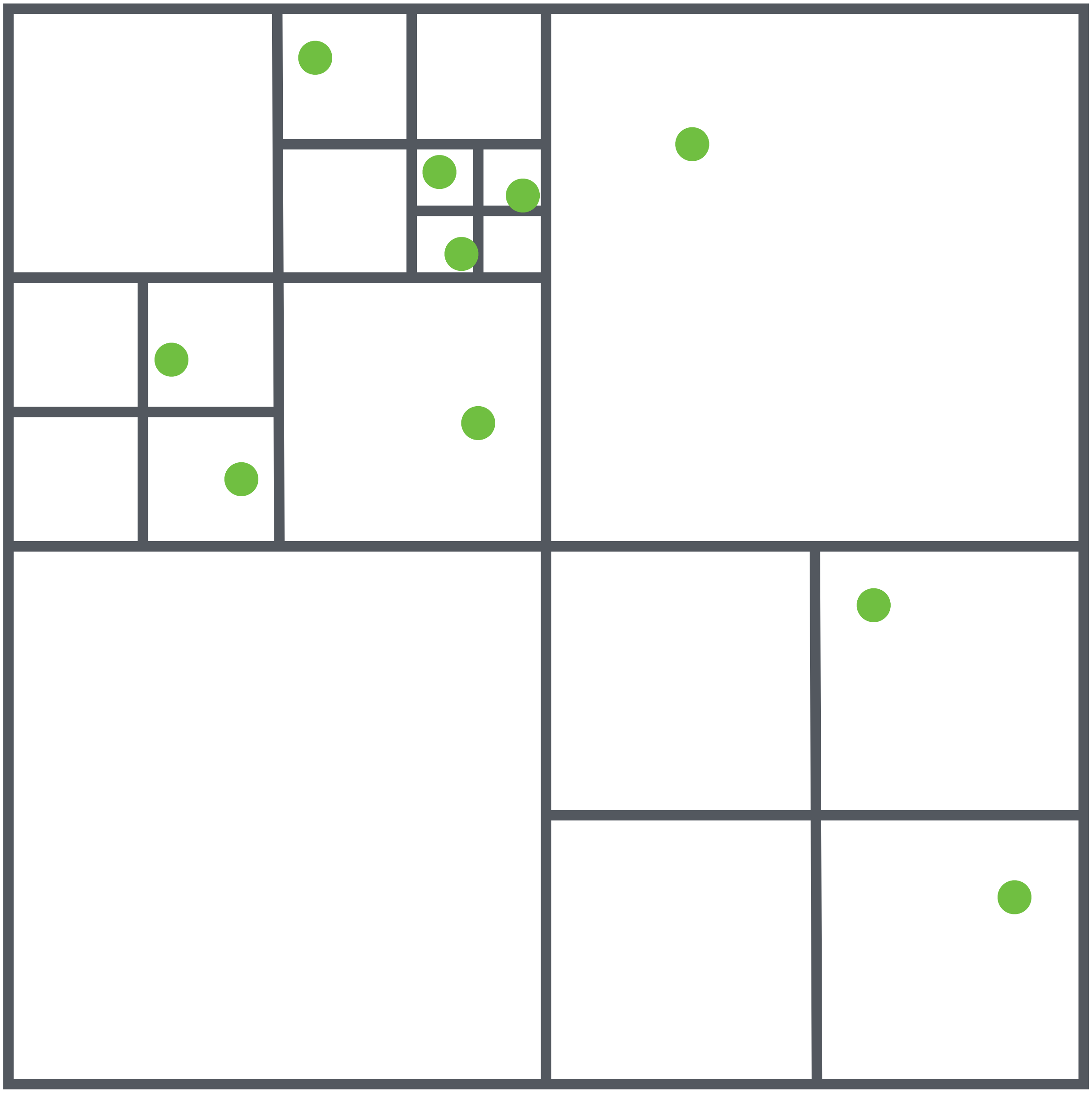


Figure 2.1: A space dividing quad-tree. If the center of a cell exceeds a threshold distance from the target point, then the forces of the child points are approximated by the cell center. This allows us to compute the repulsive forces in $O(N \log N)$ time.

the $O(N^2)$ algorithm. If a cell is deemed far enough away, the force on point y_i is given by

$$\frac{N_{cell}(y_i - y_{cell})}{(1 + ||y_i - y_{cell}||^2)^2} \approx \sum_{j \in cell} q_{ij}^2 Z^2 (y_i - y_j) \tag{2.19}$$

where N_{cell} is the number of elements in the cell. Because of the divisions of the quad tree, this approximation takes $O(\log N)$ approximations per point, and thus the whole algorithm operates in approximately $O(N \log N)^2$.

Many popular accelerated implementations of t-SNE use this method of approximation including Multi-Core t-SNE [34] and BH-t-SNE [27]. NVIDIA's RAPIDS implementation [35] is based on the tsnecuda implementation of the Barnes-Hut algorithm presented in Chan, Rao, Huang, *et al.* [36], however, tsnecuda no longer relies on this approximation, and instead explores higher performance approximations.

2.2.2.2 Fourier Approximation, FIT-SNE

While the Barnes-Hut algorithm reduces the complexity to $O(N \log N)$, the additional $\log N$ term can be limiting with the large numbers of points that we would like to work with. In addition, the Barnes-Hut algorithms suffer heavily from the curse of dimensionality, and higher dimensional approximations are both complex to implement, and significantly slower to execute. Further, Barnes-Hut is ill-suited to GP-GPU implementations since it relies on tree structures, which can be extremely inefficient to explore on GPUs due to the warp structure [33].

In an effort to simplify the algorithm, and improve the run-time, we turn to Linderman, Rachh, Hoskins, *et al.* [37], which introduces FFT-accelerated Interpolation-based t-SNE (FIt-SNE), which interpolates the far-field forces onto an equispaced grid, and uses FFT to perform the convolutions reducing the computation time to $O(Nq^2)$ where q is the number of points on the underlying interpolation grid, which significantly outperforms the Barnes-Hut implementation. First, we can re-write the repulsive forces from 2.12 as:

[2] This time can vary when working with real-world distributions. The $O(N \log N)$ time requires a fairly uniform dataset, as well as a true inverse-square (or smaller) force decay. These properties may not always be present, however we find in practice (See Section 4.2) that the growth is roughly $O(N \log N)$ in the best cases

$$\begin{aligned}
F_{rep}(k) &= \sum_{l=1}^{N} q_{lk}^2 Z(y_k - y_l) && (2.20)\\
&= \sum_{l=1}^{N} q_{lk}(1 + ||y_l - y_k||^2)^{-1}(y_k - y_1) && (2.21)\\
&= \sum_{l=1}^{N} \frac{(1 + ||y_l - y_k||^2)^{-1}}{\sum_{i\neq j}(1 + ||y_i - y_j||^2)^{-1}}(1 + ||y_l - y_k||^2)^{-1}(y_k - y_1) && (2.22)\\
&= \sum_{l=1}^{N} \left[\left(\frac{y_k - y_l}{(1 + ||y_l - y_k||^2)^2}\right)\left(\frac{1}{\sum_{i\neq j}(1 + ||y_i - y_j||^2)^{-1}}\right)\right] && (2.23)\\
&= \left(\frac{1}{\sum_{l=1}^{N}\sum_{j=1}^{N}(1 + ||y_l - y_j||^2)}\right)\sum_{l=1}^{N}\left(\frac{y_k - y_l}{(1 + ||y_l - y_k||^2)^2}\right) && (2.24)\\
&= \left(\sum_{l=1}^{N}\frac{y_l - y_k}{(1 + ||y_l - y_k||^2)^2}\right) \Big/ \left(\sum_{l=1}^{N}\sum_{j=1}^{N}\frac{1}{1 + ||y_l - y_j||^2}\right) && (2.25)
\end{aligned}$$

Thus,

$$F_{rep}(k) = \left(\sum_{l=1}^{N}\frac{y_l - y_k}{(1 + ||y_l - y_k||^2)^2}\right) \Big/ \left(\sum_{l=1}^{N}\sum_{j=1}^{N}\frac{1}{1 + ||y_l - y_j||^2}\right) \tag{2.26}$$

and, we can see clearly that the repulsive forces consist of N^2 pairwise interactions. Linderman, Rachh, Hoskins, *et al.* [37] demonstrate additionally that the repulsive forces can be re-written in form:

$$\phi(y_i) = \sum_{j=1}^{N} K(y_i, y_j)q_j \tag{2.27}$$

Where the kernel $K(y, z)$ is either

$$K_1(y, z) = \frac{1}{1 + ||y - z||^2} \quad \text{or} \quad K_2(y, z) = \frac{1}{(1 + ||y - z||^2)^2} \tag{2.28}$$

for $y, z \in \mathbb{R}^d$ where d is the dimension of the embedding space. See Linderman, Rachh, Hoskins, *et al.* [37] for a full treatment.

The kernels are smooth functions of the inputs, so they can be approximated via polynomial interpolation. A question remains: are K_1 and K_2 sufficiently low rank to approximate well with Lagrange polynomials? The short answer is yes, however we refer interested readers to Linderman, Rachh, Hoskins, *et al.* [37] and Linderman [38] which demonstrate that both K_1 and K_2 are sufficiently low rank in practice, and can be approximated to machine precision with less than 20 polynomials. The computation of the polynomial interpolation can then be accelerated via Fast Fourier Transforms. The resulting algorithm has three expensive steps. First, it must compute the polynomial coefficients for each of the $d + 2$

terms for each interpolation point. Then it must perform a FFT on resulting coefficients, to approximate the far-field forces. Finally, it must compute the $d+2$ terms $\phi(y_i)$ for each y_i. An avid reader might ask: why not compute the distance function transforms ahead of time, and then perform only the convolution during the algorithm. Because the equispaced interpolation grid is dynamically spread across the lower-dimensional points, the distance function transforms are not fixed, and thus, we need to recompute them at each time step of the algorithm. It is interesting future work to explore an algorithm which operates in a fixed interpolation space, which could be further accelerated.

The first and third steps are similar. Let I_{total} be the total number of interpolation points used. Then the first can be viewed as a kernel that sums a sparse $[(d+2) \times I_{total} \times N]$ tensor over the third dimension, generating $[(d+2) \times I_{total}]$ tensor of coefficients. The third step can be viewed as a kernel that sums a sparse $[(d+2) \times I_{total} \times N]$ tensor over the second dimension, generating a $[(d+2) \times N]$ tensor of potentials $\phi(y_i)$. The coarseness of the approximation is controlled by the number of interpolation points I_{total}.

In Section 3.2, we detail our implementation of FIt-SNE on GPU, and in Section 4.2 we show how our GPU implementation outperforms the GPU-Accelerated Barnes-Hut algorithms[3].

2.3 GPU Approximations to TSNE

This thesis outlines the development and design of tsnecuda, a GPU-accelerated approximate t-SNE embedding algorithm that achieves the state of the art performance on high-dimensional problems. While tsnecuda[4] was the first GP-GPU implementation of t-SNE, there have been several recent GPU implementations of t-SNE. In this section, we will discuss some of the derivative works, and their differences from tsnecuda.

Most prominent among these new implementations is the NVIDIA RAPIDS AI cuML implementation of the Barnes-Hut algorithm [35]. This implementation is based on the tsnecuda Barnes-Hut implementation which we discuss in Section 2.2.2.1. Raschka, Patterson, and Nolet [35] claim additional performance increases over tsnecuda, however we demonstrate in Section 4.2 that tsnecuda outperforms their implementation in almost all metrics.

Recently, Pezzotti, Thijssen, Mordvintsev, *et al.* [40] introduced GPGPU t-SNE, which approximates the repulsive forces by splatting kernel textures for each data point. This allows them to reformulate the t-SNE minimization problem as a set of tensor operations which can be accelerated using any publicly available tensor library (such as Tensorflow [41]). We compare directly to GPGPU t-SNE in Section 4.2.

A final recent tool for GPU accelerated t-SNE is Anchor t-SNE (AtSNE) [42]. AtSNE uses very similar approximation tools to tsnecuda for the attractive forces, however, it uses k-

[3] This work is previously published in the Journal of Parallel and Distributed Computing [39]

[4] Previously published in the High-Performance Machine Learning Conference (HPML2018) [36] and the Journal of Parallel and Distributed Computing (JPDC 2019) [39]

means to choose anchor points which can be used to approximate the repulsive forces. AtSNE is more memory efficient than tsnecuda, however, we see in Section 4.2 that tsnecuda can trade memory for computation performance efficiently.

Chapter 3

Algorithm: tsnecuda

In Chapter 2 we explored the methods that have been presented for accelerating t-SNE. In this section we present tsnecuda, a GP-GPU accelerated t-SNE algorithm which uses approximate nearest neighbors to efficiently approximate the attractive forces, while using either Barnes-Hut or FFT-based approximations of the repulsive forces.

3.1 Attractive Forces

In this section, we discuss the approximations and implementations that tsnecuda uses to compute the attractive forces.

3.1.1 Finding $p_{j|i}$

As discussed in Section 2.2.1 we can approximate the attractive forces with an approximate nearest neighbors algorithm. While most techniques used FLANN [32] or ANNOY [43] to compute nearest neighbors using vantage-point trees, tsnecuda pioneers using more complex techniques for approximate nearest neighbor computation, namely, tsnecuda uses a combination of a flat inverted index using locality sensitive hashing with exact post-verification for low-resource computations and IVFADC [44] for high-dimensional (and high workload) computations. Both algorithms are implemented using the FAISS [45] library for efficient GPU implementations of the algorithms. In this section, we discuss these algorithms and discuss the trade-offs that are made in the tsnecuda algorithm.

By increasing the number of neighbors u, a user can smoothly trade-off between efficiency and accuracy. While using a lower number of neighbors can be more efficient, sometimes ignoring neighbors can lead to clusters in the high-dimensional space becoming disconnected, as attractive forces are computed only between neighbors when moving the low dimensional points. Section 4.4 explores the number of nearest neighbors which is necessary for the best quality approximations.

Tsnecuda uses a pair of methods for computing the approximate neighbors, and trades off dynamically between them based on the computation overhead. In the sections below, we discuss these two unique methods for computing approximate nearest neighbors.

3.1.1.1 Flat Inverted Index

In cases where we have very few points, or are operating in low dimension, tsnecuda defaults to a base approximate nearest neighbor structure: an inverted index based on locality sensitive hashing (LSH) with multi-probing [46]. In traditional LSH indexing, we use a family of hash function $\mathcal{H}$ - usually random projection [47] to determine a hash bucket to search for nearest neighbors. To use basic LSH, first we hash the full set of high-dimensional vectors into a set of buckets. We repeat this process for the full family of hash function $\mathcal{H}$, meaning a point is hashed to multiple buckets. At query time, for each query vector q, a candidate set of vectors $\mathcal{C}$ is generated by retrieving all of the buckets that q is hashed to. The candidate set is then searched for a set of nearest neighbors in a naive way. This method, however, can require $\mathcal{H}$ to be very large to achieve strong approximation results (causing slow queries).

In order to address this, Panigrahy [48] proposes “Entropy-based hashing” which uses the same hashing method, however alters the query procedure. Entropy-based hashing computes a probability for each bucket B that the query vector q lies in B, and then samples hash buckets which have a high probability of having the data. Unfortunately, performing the distance computation directly is cumbersome - so instead of directly computing the probabilities, Panigrahy [48] samples random points around q in the higher dimensional space, and use their hashed values to generate the candidate set of buckets. While this method can reduce the number of hash tables, it significantly increases query time - and also requires knowledge of the nearest neighbor distance, which can be difficult to compute in a dataset-specific way, and can be impossible to tune.

Tsnecuda uses LSH with Multi-Probing [46] (Figure 3.1, which is designed to address the issues with both of these methods (basic and Entropy-based). Multi-probe LSH uses a carefully derived probing sequence to check multiple buckets which are likely to contain the nearest neighbors of a query vector. It does so by choosing a “hash-perturbation vector” which can be added to the hash bucket to generate a neighboring bucket that can be searched. Because the search strategy depends on the hash function, and not the dataset, it can be precomputed to save time during the query phase. We refer interested readers to Lv, Josephson, Wang, *et al.* [46] for a full treatment of the method.

We use the GPU implementation of LSH with multi-probing provided by Johnson, Douze, and Jégou [45]. Clearly, an inverted index has significant benefits over vantage-point trees when it comes to implementation on GPU devices, as they have an embarrassingly parallel structure, and perform the same operation repeatedly while search the indices, allowing for a GPU warp to efficiently explore a hash bucket. While it can be efficient, LSH requires a large amount of memory bandwidth, and is often memory-limited (See Section 4.2 for further analysis). The exact GPU implementation details are described in Johnson, Douze, and Jégou [45].

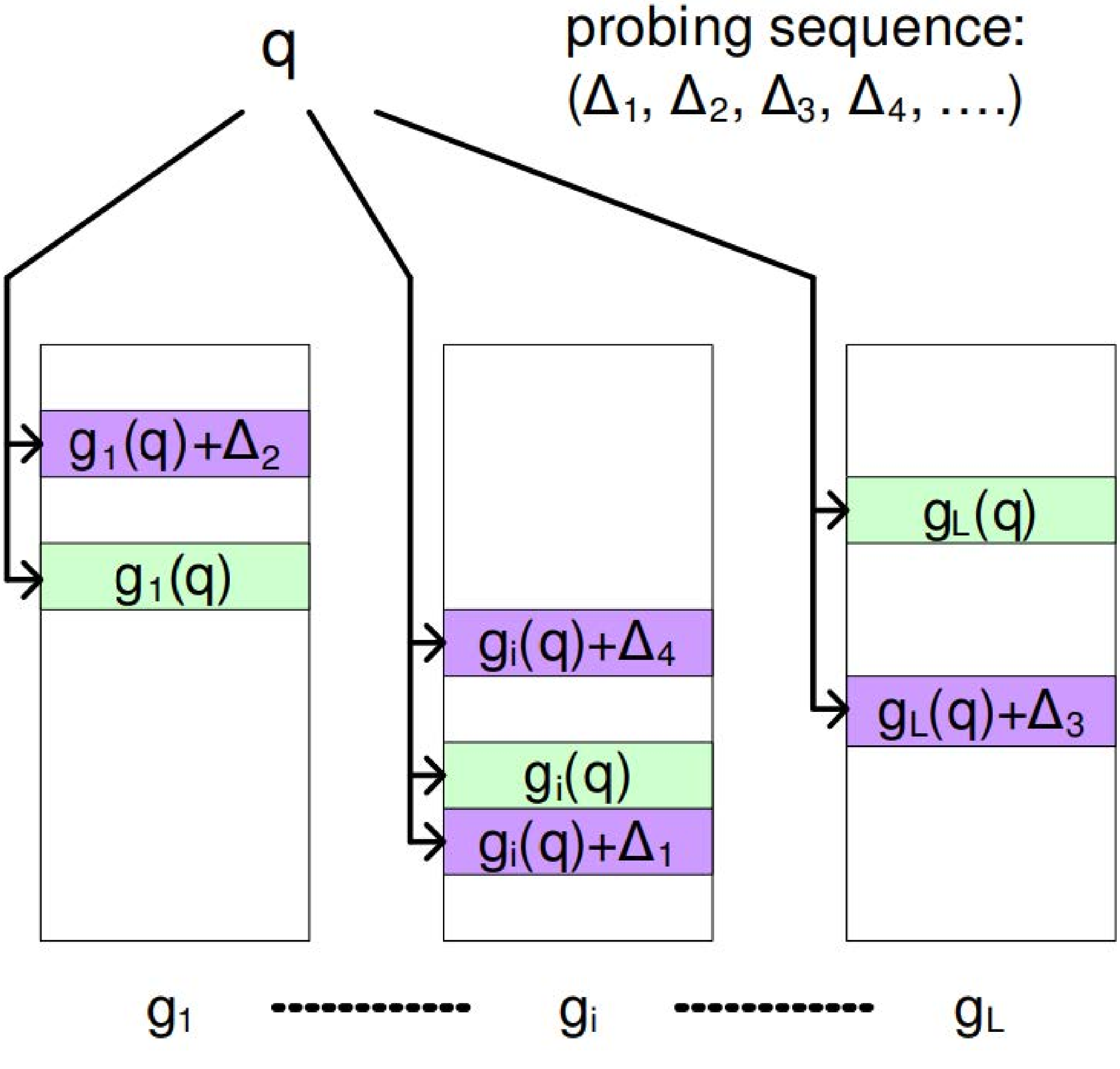


Figure 3.1: Multi-probe LSH uses a sequence of hash perturbation vectors to explore multiple buckets. $g_i(q)$ is the hash function for table $i \in \mathcal{H}$, and Δ_j is a hash-perturbation vector. Figure from [46].

There are several parameters to the above hashing method which need to be selected by the Tsnecuda algorithm. There are the size of $\mathcal{H}$, the number of buckets for each hash function r, and the number of buckets that should be probed for nearest neighbors w. Johnson, Douze, and Jégou [45] found that for many use cases, setting $|\mathcal{H}| = 1$ was sufficient. In Tsnecuda, we scale $r \in O(\sqrt{N})$, which, for a well distributed dataset, will generated inverted indices which are size $O(\sqrt{N})$. This is not, however, always the case - and datasets that are highly correlated with the hash function can cause significant slowdowns in the algorithm. Finally, in Tsnecuda we found that searching $w \approx 20$ cells was sufficient to find many of the nearest neighbors in most of our datasets. This number is somewhat arbitrary, and depends significantly on the dataset, so we expose this parameter to the user to tune for their particular experiment.

3.1.1.2 IVFADC

While using a flat index can be useful when there are fewer or low-dimensional vectors, for high dimensional or very large data, it is important to improve the efficiency of both memory storage and queries over a flat index. When the number of points or the dimension of those points exceeds a certain limit size, tsnecuda switches to using IVFADC as a search method. This allows tsnecuda to solve both large and small problems efficiently.

IVFADC uses the same multi-probing structure from the last section, however additionally adds product-quantization to the hash buckets, which compresses vectors into a short code (in the case of tsnecuda, we use 8-bit codes), so the real vector need not be stored in the index (only the short code representation). Unfortunately, linearly scanning a PQ inverted index can be slow as the size of the code-set grows.

In order to avoid this linearly growing cost, database vectors y are encoded using

$$y \approx q(y) = q_1(y + q_2(y - q_1(y)) \tag{3.1}$$

where q_1 and q_2 are quantizing functions. The q_1 function is a coarse quantizer, while the q_2 quantizer is a finer approximation encoding the residual value. We then rephrase the nearest neighbor problem

$$\mathcal{N}_x = k\text{-argmin}_{y \in N} ||x - y_i|| \tag{3.2}$$

as an approximate asymmetric distance problem, where $k-$argmin is an operation computing the k minimum values across the data. For a number τ of inverted lists to scan, the algorithm first computes

$$\mathcal{N}_x^{IVF} = \tau\text{-argmin}_{c \in \mathcal{C}} ||x - c|| \tag{3.3}$$

This gives a coarse grained approximation of the location of the point x in terms of "centroids" in $\mathcal{C}$. We then construct our nearest neighbors

$$\mathcal{N}_x \approx k\text{-argmin}_{y \in N | q_1(y) \in \mathcal{N}_x^{IVF}} ||x - q(y)|| \tag{3.4}$$

by storing the index as an inverted file, and grouping the vectors around the centroids, scanning the τ lists for k neighbors each. We can achieve a look-up by linearly scanning

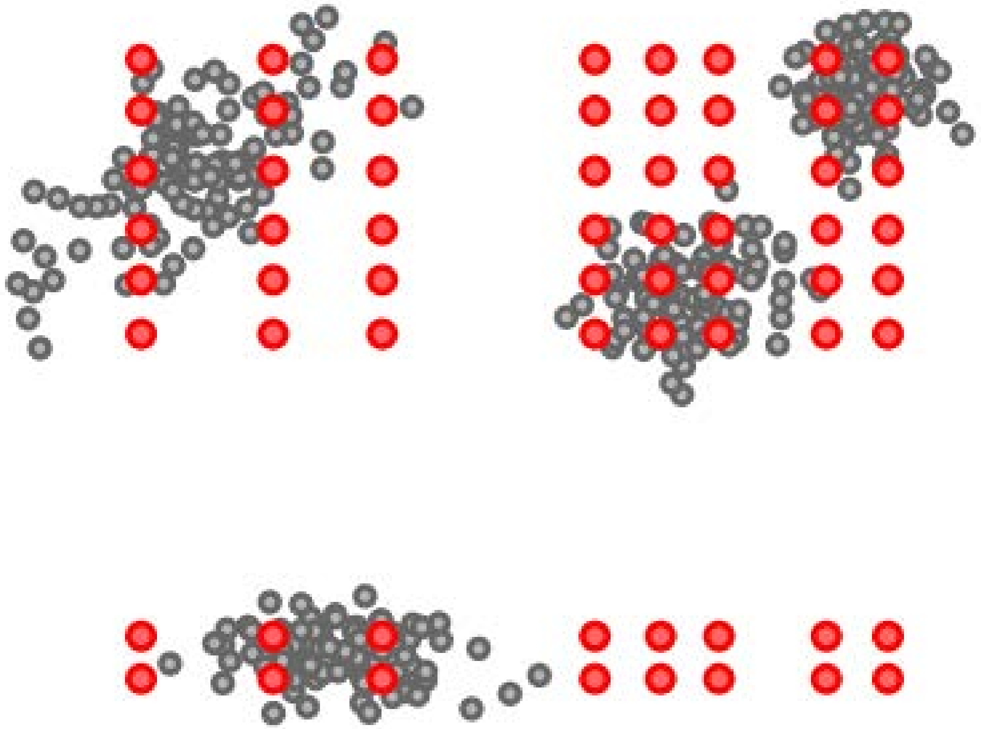


Figure 3.2: Example of the quantization structure (in 2-D) formed with IVFADC [49].

$O(\tau)$ inverted lists, making the approximate nearest neighbor algorithm $O(N)$ which gets more efficient for large N. For more details of this algorithm, we refer interested readers to Johnson, Douze, and Jégou [45]. Figure 3.2 gives an example of the quantization that we use: We first select a set of quantization vectors by building using k-means to generate locations in the higher-dimensional space. We then use a fine-grained quantizer (in this case, a grid), to fine-tune the quantization which can be used to compute approximate nearest neighbors.

In the case of tsnecuda, we assume that k-means is a good approximation to the quantization space. This may not, however, always be true (For example, a uniform line of points in the higher dimensional space). Because the goal of t-SNE is to focus on local neighborhoods, we hope that such neighborhoods will likely fall in similar k-means quantization bins, allowing for accurate computation of the neighbors with relatively few queries. It is worth noting, however, that this assumption, can cause our visualization to have odd artifacts if we do not search enough quantization bins.

For our implementation, we choose $|C| = \sqrt{N}$, and train the vectors C using the k-means algorithm. Thus, q_1 is the id of the nearest centroid. q_2 is much more precise, and is selected using product quantization [44] which interprets the vector y as a set of quantized sub-vectors. The parameter τ, selected by the user, controls the accuracy of the KNN algorithm.

3.1.1.3 Approximate Nearest Neighbor Parallelism

In addition to being faster than using tree-based structures, inverted files allow for an efficient multi-GPU implementation of our algorithm. By replicating the centroid index across multiple GPU devices, we can achieve almost linear speedups by splitting the queries across the GPUs. In cases where there are too many points to fit in GPU memory, Inverted files can be used to balance the load across multiple devices, costing at most $n - gpu$ additional look-ups.

3.1.2 p_{ij} Symmetrization

While we have been able to successfully approximate $p_{j|i}$, we still need to efficiently compute p_{ij} according to Equation 2.4. A naive symmetrization algorithm using a sparse matrix-matrix multiplication as we explored in [36] and [39] can up to double the number of possible points that are considered in the computation of the attractive forces. While x_i may be an approximate nearest neighbor of x_j, x_j may not be a neighbor of x_i. The naive symmetrization will add a matrix entry in this case.

This thesis version of tsnecuda drops a full symmetrization to bring tsnecuda in line with the algorithm implementations in Raschka, Patterson, and Nolet [35] and Fu, Zhang, Cai, *et al.* [42]. Both of these works found that full symmetrization was not necessary for the effective generation of the t-SNE embedding. Instead of adding a new matrix entry, we ignore the inverse relationship entirely. Doing this not only improves the computational efficiency of the symmetrization algorithm which no longer needs to rely on a sparse matrix-matrix computation but also improves the efficiency of the later attractive force computations which no longer have to do sparse matrix reductions.

3.1.3 Computing the Attractive Forces

Once the k-Nearest Neighbors are computed, and we have computed the sparse matrix P which stores the nonzero values of p_{ij}, we can focus on actually computing the attractive forces during each iteration of the gradient descent algorithm.

In previous work, we decompose the attractive force computation into a series of sparse matrix computations [36]. However, we found in Chan, Rao, Huang, *et al.* [39] that performing the reduction directly using device atomic instructions is superior, especially for smaller numbers of points ($\leq 512,000$). In Chan, Rao, Huang, *et al.* [39] we computed $p_{ij}q_{ij}(y_i - y_j)$ and use `atomicAdd` to directly sum the result into the attractive force matrix. To achieve the $O(Nu)$ run-time, we represented P as a sparse matrix with a nonzero value at i, j iff j is a neighbor of i. The matrix Q containing the elements q_{ij} is never computed in its entirety; instead, the matrix $P \odot Q$[1] is computed directly by iterating over nonzero values of P.

By introducing the novel symmetrization described in Section 3.1.2, we can further simplify the computation of the attractive forces during the gradient descent. Because we have

[1] $\odot$ is the element-wise product

a dense matrix of values (each point has exactly u neighbors), we can write efficient CUDA kernels that loop through the attractive forces and store this information in a work-space. Then, we efficiently sum the columns in the work-space with dense matrix-vector multiplication. If memory efficiency is desired, tsnecuda can avoid using work-space memory here by instead using device atomics to directly write to the attractive force vector. We found on average that this was 5-7% slower, and decided to trade additional memory for computational efficiency.

This optimized algorithm is referred to as tsnecuda (optimized) in Section 4.2 while discussing the performance.

3.2 Repulsive Forces

We have already discussed several methods for approximating the repulsive forces in Chapter 2. Tsnecuda has implementations of both the Barnes-Hut algorithm [27] and the FIt-SNE algorithm [37], however, we have found FIt-SNE to perform better than Barnes-Hut in almost all scenarios. In addition, the FIt-SNE algorithm is far easier to parallelize, making it an excellent choice for tsnecuda's implementation. In this section, we discuss the implementational details of each of the algorithms, and some optimizations that were necessary to improve the performance of each algorithm on GPU.

The Barnes-Hut implementation of tsnecuda is based on an implementation of Barnes-Hut from Burtscher and Pingali [33] - which uses a quad-tree implementation on the GPU to compute the repulsive forces. While quad-trees are inefficient to use on GPUs, by limiting the depth of exploration and forcing all warps to explore to the same depth, the implementation can be made somewhat efficient. This algorithm is referred to as tsnecuda (BH T-SNE) in Section 4.2 while discussing the performance.

The FIt-SNE implementation of the algorithm is built on cuFFT, a fast FFT library for CUDA developed by NVIDIA. An overview of the algorithm is given in Section 2.2.2.2. We follow the Linderman, Rachh, Hoskins, *et al.* [37] implementation and use a 150x150 grid of equispaced interpolation points for the FFT. In Section 4.4 we explore the effect of altering the number of interpolation points. This algorithm is referred to as tsnecuda (FIt-SNE) in Section 4.2 while discussing the performance.

3.3 Memory Usage

While we have analyzed the computation time requirements of tsnecuda, we haven't closely discussed the memory requirements for the algorithm.

There are two phases that the algorithm passes through: the computation of the nearest neighbors, and the gradient descent iterations. The memory usage for the IVFADC method used is analyzed fully in Johnson, Douze, and Jégou [45], and depends on the number of points. The memory usage is $O(Nd)$ bytes where N is the number of points and d is the number of dimensions when using the flat inverted index. In practice, the constant is close to 18. When using IVFADC to decrease memory usage, the usage depends heavily on the encoding of the sub-vectors. In tsnecuda, we encode each vector with 16 bits - this reduces the memory used to $O(N)$. In practice, the constant is close to 30. See Johnson, Douze, and Jégou [45] for additional memory analysis.

In the second phase, for the repulsive forces we allocate memory for both the attractive and repulsive forces. For the repulsive forces, we allocate a work-space for the interpolated points, and the fast-FFT used to compute the matrix multiplication. When using q interpolation points on each dimension per interval, I interpolation intervals, and a lower-dimensional space of dimension 2, the allocated space is $O(Nq^2I)$ bytes[2]. With the options used in the paper, we use 644 bytes per point, with a 5.08MB overhead. To compute the attractive forces, we allocate a work-space for the device p_{ij} values which is $3Nu$ floating points values where u is the number of nearest neighbors. For performance with large numbers of points, the attractive force computation drops the additional $2Nu$ work-space which is added for efficient reduction. Finally, we need to allocate a number of vectors for accumulating data, totaling $11N$ floating points.

Overall, as implemented during the iteration the algorithm allocates a total of 1.07KB for each point, along with a 5.08MB overhead. With the dropped attractive force overhead from the work-space, we allocate 816 bytes per points. This is a clear area for optimization, which we discuss in Section 4.2.

3.4 Multi-GPU Computation

The thesis version of tsnecuda also introduces multi-GPU computation. An updated FAISS library allowed for multi-GPU use during the K-Nearest Neighbors computation, either by duplicating the search index across GPUs to improve performance or by splitting the search index across GPUs to increase search capacity. Tsnecuda dynamically trades off between the two modes as required, allowing nearest neighbors to be efficiently computed in a multi-GPU scenario.

In addition to parallelization of the nearest neighbors across GPUs, the introduction of the novel symmetrization tricks from Section 3.1.2 allow us to split the computation of the attractive forces across GPUs as well. By maintaining two copies of the lower-dimensional points array, and by splitting the matrix P along the point dimension, for N points and k GPUs we can compute N/k attractive forces on each device in parallel as the computations are independent. This parallelization strategy can have diminishing returns as the number of points increases, however, as the points array has to be copied between GPUs, which can be a limiting factor when not using specially accelerated hardware such as NVLINK.

[2] We refer interested readers to the source code at `http://github.com/cannylab/tsnecuda` for further information

3.5 Additional Algorithm Improvements

The optimized thesis version of tsnecuda introduces a number of additional minor changes from the optimization above from Chan, Rao, Huang, *et al.* [36] and Chan, Rao, Huang, *et al.* [39], which are summarized below:

- Support for multiple metrics in the higher dimensional space (Now supports inner product, and L_p (including L_{inf}) in addition to the L_2 metric)
- Memory optimizations (with a 25-50% savings in overall memory)
- Dynamic balancing of inverted files and IVFADC for nearest neighbor lookup
- Dynamic kernel parameter selection
- Dynamic selection of grid cells for FFT computation
- Real-Time visualization of t-SNE computation using ZMQ and Matplotlib
- Faster compilation times (compilation reduced by almost 80% over previous version)
- Support for analyzing the KL-Divergence in addition to the gradient norms
- Additional python examples and benchmarks

3.5.1 Algorithm

Algorithm 2 gives the full outline of the discussed sections. Our full implementation is publicly available, so we omit many of the code details.

Algorithm 2 General GPU Accelerated T-SNE Algorithm

```
Input: $N \times d-$dimensional array of data
Output: $N \times 2-$dimensional projection
FAISS Computation (approximate k-NN)
Use pairwise distances to compute sparse matrix of $P_{ij}$
for $i = 1$ to convergence do
    if using Barnes-Hut then
        R-Force Tree Building (build quad-tree for Barnes-Hut)
        R-Force Computation (use tree to compute approximate repulsive forces)
    else if using FIt-SNE then
        Compute polynomial interpolant coefficients
        Compute FFT of coefficients
        Compute terms $\phi(y_i)$
    end if
    Compute Attractive Forces via $p_{ij}q_{ij}(y_i - y_j)$ and sum across $j$
    Apply Forces (apply forces to points in lower dimensional space)
end for
return lower dimensional projection
```

Chapter 4

Performance Analysis and Results

In this section, we analyze the performance of tsnecuda (all three of the forms) against state of the art algorithms for t-SNE. In Section 4.1, we discuss the experimental environment and talk about how the experimental environment impacts the performance of tsnecuda. In Section 4.2, we look at the raw performance of the tsnecuda algorithm against the other state of the art algorithms across several common public datasets. In Section 4.3, we discuss the performance of the individual GPU kernels and identify room for improvements. Finally, in Section 4.4, we discuss the impact of the approximations that we have made on the quality of the embeddings.

4.1 Experimental Design

To evaluate tsnecuda, we present both qualitative and quantitative experiments in which tsnecuda is compared to other state of the art algorithms (Discussed in Chapter 2). In this section, we provide the experimental details and setup for the experiments we run in this work.

4.1.1 Computation Environment

We perform experiments given in this paper using a system with an Intel i7-5820K Processor clocked at 3.3Ghz, containing 6 physical cores (12 with hyperthreading) and 64GB of DDR4 RAM. The GPU in use on this system is the NVIDIA Titan-X Maxwell edition GPU, with 3072 CUDA cores clocked at 1.0 GHz and 12GB of GDDR5 memory. It supports a maximum memory speed of 7.0Gbps, with a maximum memory bandwidth of 336.5Gbps. The GM200 chip (Titan-X) has 3072Kb of L2 cache, and 24 Streaming Multiprocessors (6GPCs), with a theoretical peak performance of 6.12TFLOPs for single-precision floating-point operations. The CPU platform has a theoretical peak performance of 691.2GFlops, giving a peak-to-peak theoretical margin of 8.85x. Unlike in Chan, Rao, Huang, *et al.* [39], we now no longer optimize the CUDA grid sizes for our GPU, as they are selected dynamically by the tsnecuda algorithm.

The multi-GPU experiments performed in Section 4.2.4 are performed on a different testbed due to the i7-5820K only having support for 28 PCI lanes, whereas to address 8 GPUs at the minimum required 8x PCI lanes, we need at least 64 PCI lanes. For multi-GPU training, we use 8x NVIDIA Quatro RTX 8000 GPUs on VMs running on Google's Cloud Platform. These machines have Skylake CPUs with 64 vCores and 128GB of DRAM.

4.1.2 Datasets

We analyze the performance of tsnecuda using several datasets:

Simulated Data: It is important to be able to benchmark methods in a controlled experiment, so we construct simulated data consisting of equal-sized clusters of points sampled from four 50-dimensional Gaussian distributions with identity covariance. In these experiments, we can vary the size and dimensions of the data, and effectively examine the performance of different algorithms in a controlled environment.

MNIST: The MNIST dataset [50] is a classic computer vision dataset consisting of 60,000 training images, and 10,000 testing images depicting different handwritten digits (numerals 0-9). Each of these digits is black and white, with dimensions 28x28 constituting a 784-dimensional image space.

CIFAR: The CIFAR datasets [51] both consist of a 50,000 image subset from the larger tiny-image dataset. The datasets have images of 10 (Resp. 100) classes such as “ship”, “car”, “horse”, “frog” etc. The images from the CIFAR-10/100 datasets are full-color images with dimensions of 32x32x3, giving a 3072-dimensional data space to explore. In addition to examining CIFAR-10 in the RAW data format, we also explore CNN generated codes using LeNet [52]. These are 1024 dimensional embeddings of each image learned by the neural network.

4.1.3 Comparison Algorithms

We compare tsnecuda against seven different algorithms in our experiments. These algorithms were chosen as a representative sample of algorithms from the most commonly used (SkLearn) to the fastest overall (FIt-SNE). Pezzotti *et al.* [31] do not make their code public, so we could not compare against their code, while the code presented by Dimitriadis *et al.* [53] is targeted to neural spike sorting, and is not optimized for general t-SNE.

SkLearn: The SkLearn implementation [54] is the most popular implementation and is a python-based implementation of the algorithm. We use the Barnes-Hut approximation version of the algorithm. It computes exact nearest neighbors.

BH-TSNE: BH-TSNE is the original implementation by Van Der Maaten *et al.* [27]. It does not have parallel computation support and relies on the Barnes-Hut approximation to compute the repulsive forces. It uses k-Nearest Neighbors (32) to reduce the computational load, however, it does not use approximate nearest neighbors.

MULTICORE-TSNE: Multi-Core T-SNE [34] is an algorithm that targets parallelization of the k-Nearest Neighbors step using CPU-based parallelization. We refer to this algorithm as MULTICORE-K where K is the number of cores used in the computation. It seems in practice that 4 is the optimal number of cores to use for the computation, with more cores providing little benefit to the computational speeds.

FIt-SNE: FIt-SNE [37] is a CPU-based implementation of the algorithm discussed in Section 2.2.2.2. It uses the ANNOY library to compute approximate nearest neighbors.

RAPIDS: RAPIDS [35] is an optimized GPU version of t-SNE based on the tsnecuda Barnes-Hut implementation [36]. RAPIDS uses much of the same code, however, claims to streamline the initialization and attractive force computation.

AtSNE: Anchor t-SNE [42] is a GPU accelerated t-SNE method which uses anchor points in the data to compute the repulsive forces.

GPGPU-TSNE: GPGPU-TSNE [40] is a GPU accelerated t-SNE method which uses tensor operations to compute the underlying operations of t-SNE (hence, allowing heavily optimized libraries such as Tensorflow [41] to compute the overall operations).

4.2 Algorithm Performance

In this section, we discuss the performance of tsnecuda compared to the algorithms given in Section 4.1.3. We explore the performance across several datasets (given in Section 4.1.2) and explore questions with multi-GPU training.

4.2.1 Synthetic Data

Figure 4.1 gives the overall performance on the synthetic dataset for all of the methods presented in this paper. We can see that in general tsnecuda outperforms all methods. It's interesting to note that tsnecuda outperforms RAPIDS, mostly because it uses the optimized FIt-SNE algorithm over the older Barnes-Hut t-SNE algorithm. The optimized tsnecuda (which is novel to this work), clearly outperforms the other tsnecuda implementations -

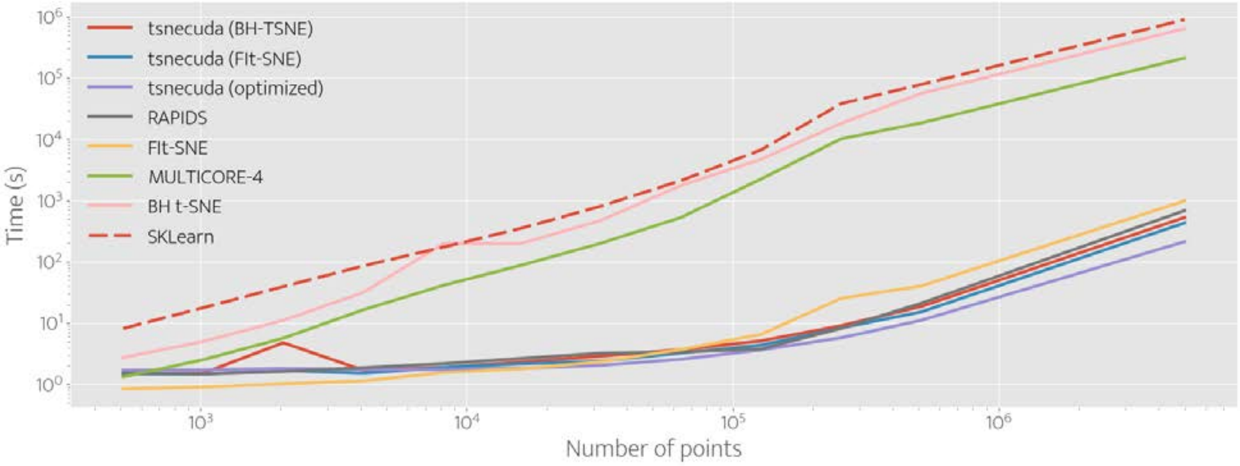


Figure 4.1: Performance on the synthetic dataset.

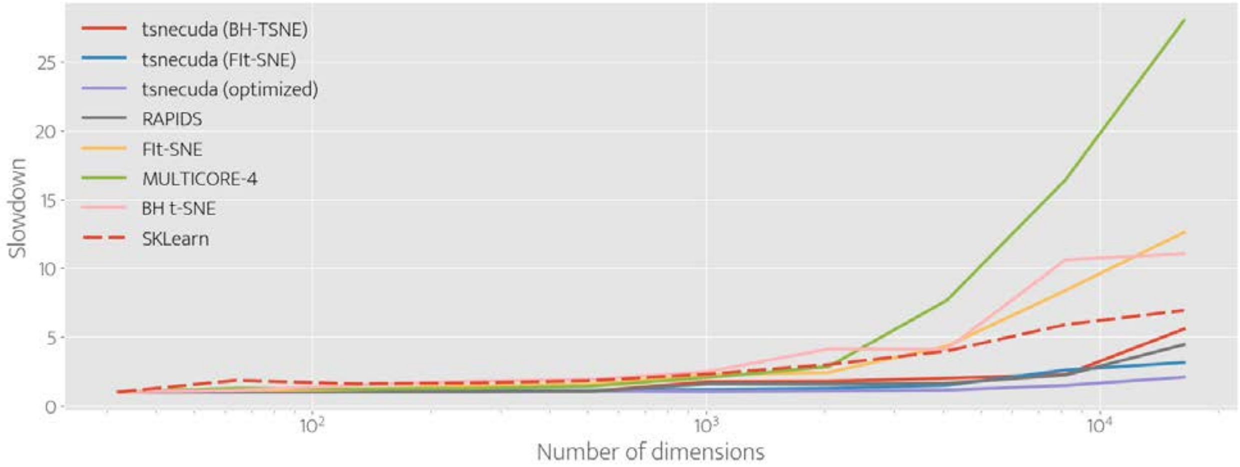


Figure 4.2: Performance on the synthetic dataset for differing numbers of dimension (5000 points, slowdown is computed as $\frac{\text{Time @ X dim}}{\text{Time @ 10 dim}}$).

primarily due to its streamlined optimization of the attractive forces - which make up a large majority of the computation time (See Figure 4.8).

Figure 4.2 demonstrates the performance of tsnecuda across multiple dimensions for 5000 points on the synthetic dataset. We can see that the optimized tsnecuda, which makes use of the ability to switch dynamically between basic inverted files, and IVFADC for nearest neighbor computation outperforms the other two implementations of tsnecuda as the dimensions grow. We also see that tsnecuda outperforms RAPIDS in higher dimensions, however, the performance difference is slight (due to RAPIDS's use of the same ANN libraries). We also outperform all of the CPU-based tools, which do not use an approximate nearest neighbor method, which can cause slowdowns even at very low numbers of points as the dimension grows (see MULTICORE-4, which uses KD trees for nearest neighbor computation).

4.2.2 MNIST

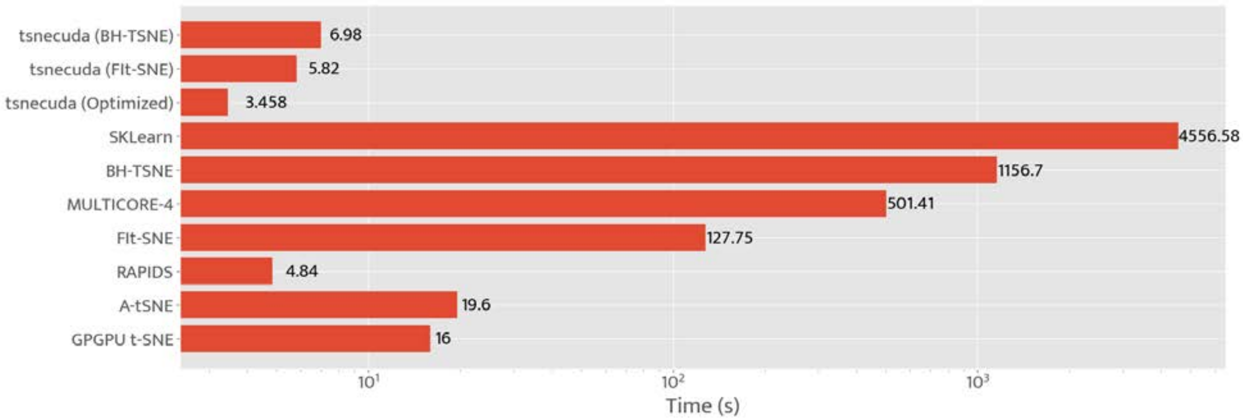


Figure 4.3: Performance on the MNIST dataset

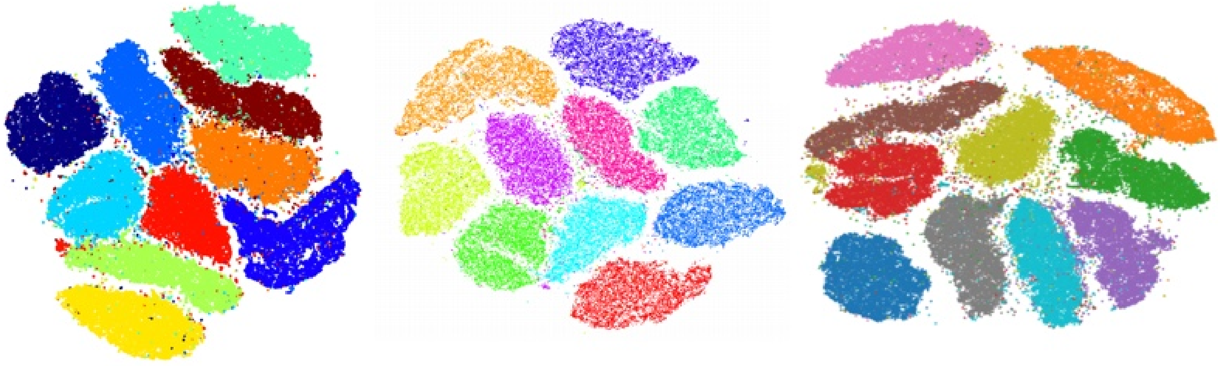


Figure 4.4: Qualitative examples of the MNIST embedding. Left: MULTICORE-4, Center: BH-TSNE, Right: tsnecuda. Tsnecuda has some additional artifact points (noise around the cluster edges), due primarily to the approximation in the nearest neighbors, however the larger structures are intact, including the two-lobe structure of the 3s (Dark blue cluster on the left, green cluster center, and red cluster on the right).

Figure 4.3 gives the performance of tsnecuda on the popular MNIST dataset. The computation here is the mean of 20 runs of the MNIST dataset. We can see that tsnecuda outperforms all of the state of the art embedding algorithms (including those that use GPUs or are a followup to the original tsnecuda paper).

While AtSNE and GPGPU-TSNE are slower, recall from Section 3.3 that one major issue that tsnecuda suffers from compared to these algorithms is the amount of GPU memory that is used. In the chase for performance, tsnecuda chooses to trade off memory for speed in almost all scenarios. While this means that we have relatively high performance, it means that tsnecuda can be limited when it comes to very large datasets ($\geq 30M$ points on a single GPU). For example, tsnecuda takes 3.6Gb of GPU VRAM to compute the MNIST embedding, while AtSNE requires only 750Mb. Because tsnecuda scales to large numbers of GPUs, this is less of an issue - however, we believe that it is interesting and important future work to continue to reduce the memory footprint of the algorithm while maintaining the performance.

In this vein, NVIDIA provides a unified memory structure which unifies CPU and GPU memory, however, we found in limited experimentation that the amount of time spent copying memory to the GPU on-demand was significant for the datasets that most users would explore (datasets below 10 million points in 1024 dimensions). Using unified memory in these cases was up to 60% slower than managing all memory on the GPU. Some limited optimization could be made. In the computation of the attractive forces, we use a work-space of size $O(Nu)$ and a reduction kernel to avoid the use of atomic access in the attractive force computation kernel. In practice, we found that this is quicker than the atomic reduction by almost 15%, however, when N and u are high, such a trade-off can be extremely expensive in terms of memory. Similarly, the FFT and Nearest Neighbor code also use large memory work-spaces which can be limiting for very large datasets. It remains future work for tsnecuda to expose these memory decisions to the user via the python wrapper, so users can decide which trade-offs they would prefer to make in terms of speed vs. GPU memory.

4.2.3 CIFAR

Figure 4.5 gives the performance of our algorithm on the LeNET [52] codes for the CIFAR-10 dataset. We can see that similar to the MNIST dataset, tsnecuda outperforms all available implementations while operating in this search space. Figure 4.6 demonstrates the quality of the embeddings generated using tsnecuda. In some cases, the clusters are more defined when using tsnecuda compared to other algorithms. We do however see the effect of using a small number of nearest neighbors can cause global clusters to be independent, such as in the brown cluster at the bottom of the figure. This is a clear downside to using approximate nearest neighbors, and a trade-off which can be explored by the user (See Section 4.4).

4.2.4 Multi-GPU Performance

Currently, tsnecuda is the only GPU-based implementation which makes use of multiple GPUs for execution, which can allow tsnecuda to scale beyond what is possible for other GPU accelerated implementations. Figure 4.7 gives the performance of tsnecuda spread

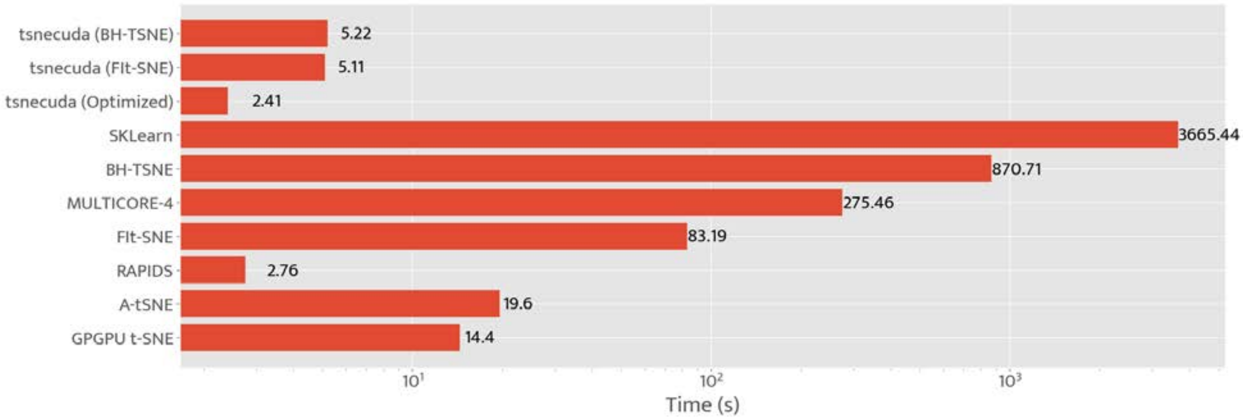


Figure 4.5: Performance on the LeNET [52] codes for the CIFAR-10 dataset.

across multiple GPUs. We can see that the speedup is slightly sublinear, with 8GPUs obtaining a speedup of 5.64 on 50M points vs. the single GPU. This is likely due to the large amount of communication that must occur with such a large number of points since at each step the entire point array must be synchronized. Because we do not make use of NVLINK between the GPUs, the transfer must be staged through the host machine (i.e. the CPU) making transfers limited to the PCIe bandwidth available. This problem is likely compounded by the fact that 8 GPUs can only operate on a single host with 4 PCI lanes per CPU, meaning that the speed of memory copies is reduced even further.

To make the above concrete, for 50M points and 2 GPUs, we expect a transfer of approximately 1.7Gb that has to be synchronized between the devices. PCI running at x16 has a peak transfer speed of 32GB/s, while DDR4 RAM at 2133 MhZ has a peak transfer speed of 17 GB/s. Because the data has to proxy through the host, we must perform four total transfers through RAM. This leads to an additional 0.42s per iteration, or a total of 420s over the course of 1000 iterations. This roughly aligns with our actual computation times of 2489s for 1 GPU, and 1673s for 2 GPUs. Linear scaling would suggest that the computation would take about 1244.5s. With the above approximations, the memory transfer explains about 98% of the additional computation time.

4.3 Individual Kernel Performance

Figure 4.8 gives an overview of the kernel times for different numbers of points (all on the synthetic 50-dimensional dataset). We can see that as the total number of points grows, the computation which is required by the attractive forces and the KNN computation begins to dominate the computation time. Both of these kernels are heavily bound by the memory access speed of the GPUs. When pre-computing the nearest neighbors for the computation of the matrix P, a single thread may be required to search across multiple disjoint inverted

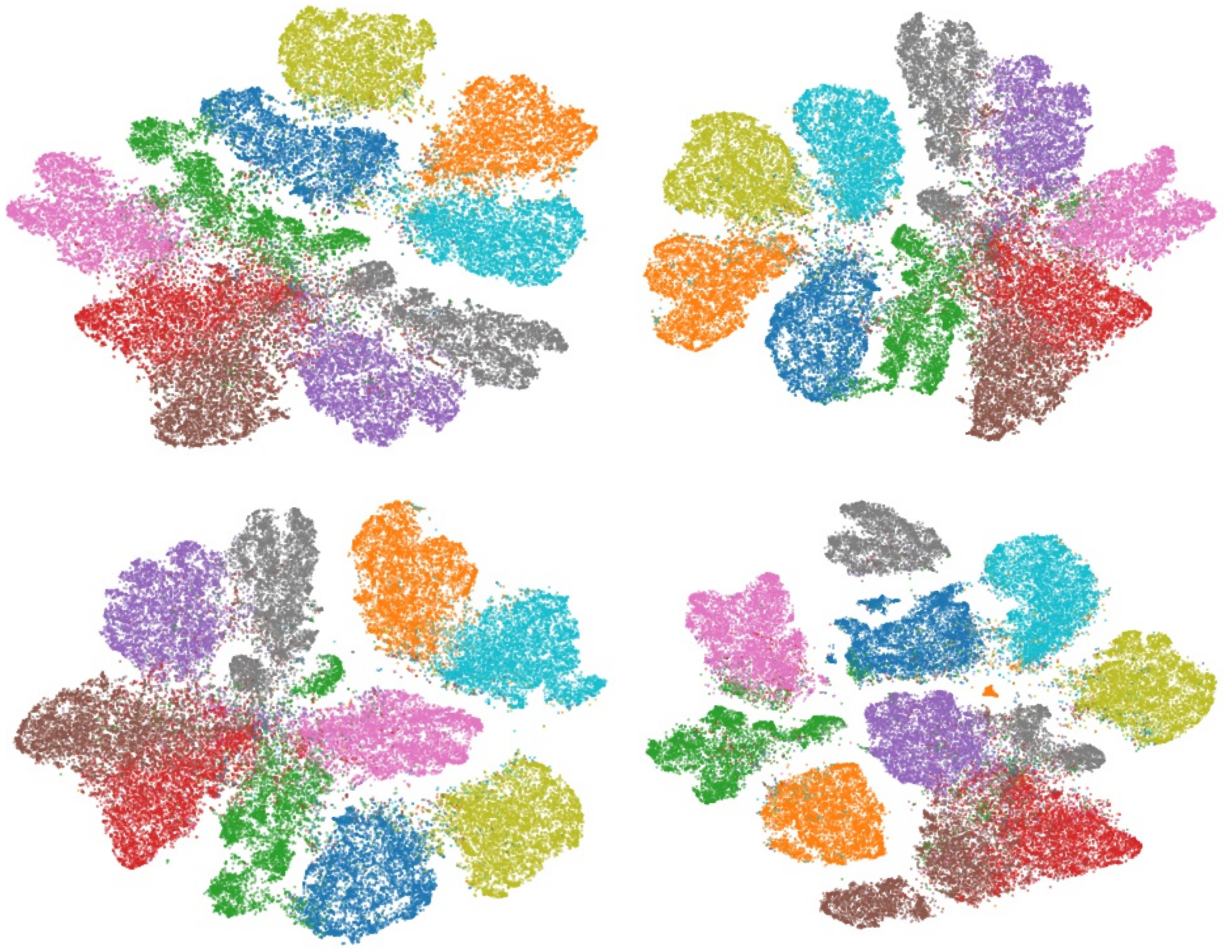


Figure 4.6: Qualitative examples of the LeNET [52] codes on the CIFAR-10 dataset. Top-Left: SKLearn, Top-Right: BH-TSNE, Bottom-Left: MULTICORE-4, Bottom-Right: tsnecuda. We can see that the tsnecuda structures are well separated, with little noise in-between the clusters. There are some small clusters which appear due to the nearest neighbors approximation (for example, the small orange cluster above and to the right of the purple cluster, and the green cluster directly attached to the pink cluster), however the algorithm still does a good job at separating the main bodies of points.

list elements. This means that each warp in the GPU kernel is performing a large number of relatively random reads across the entire point set - which can easily cause the GPU to function below its peak performance. It is interesting future work to explore spatially grouping nearest neighbor queries to improve access times, however the cost of grouping may outweigh the search time benefits. This can be seen from Figure 4.9, where we can see that the GPU is largely waiting on memory accesses during the nearest neighbor computation.

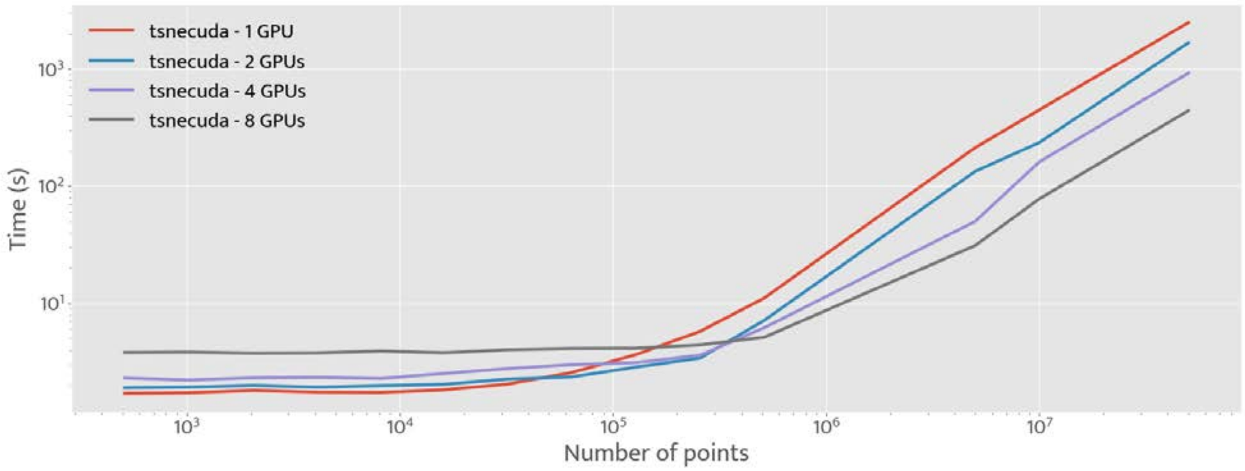
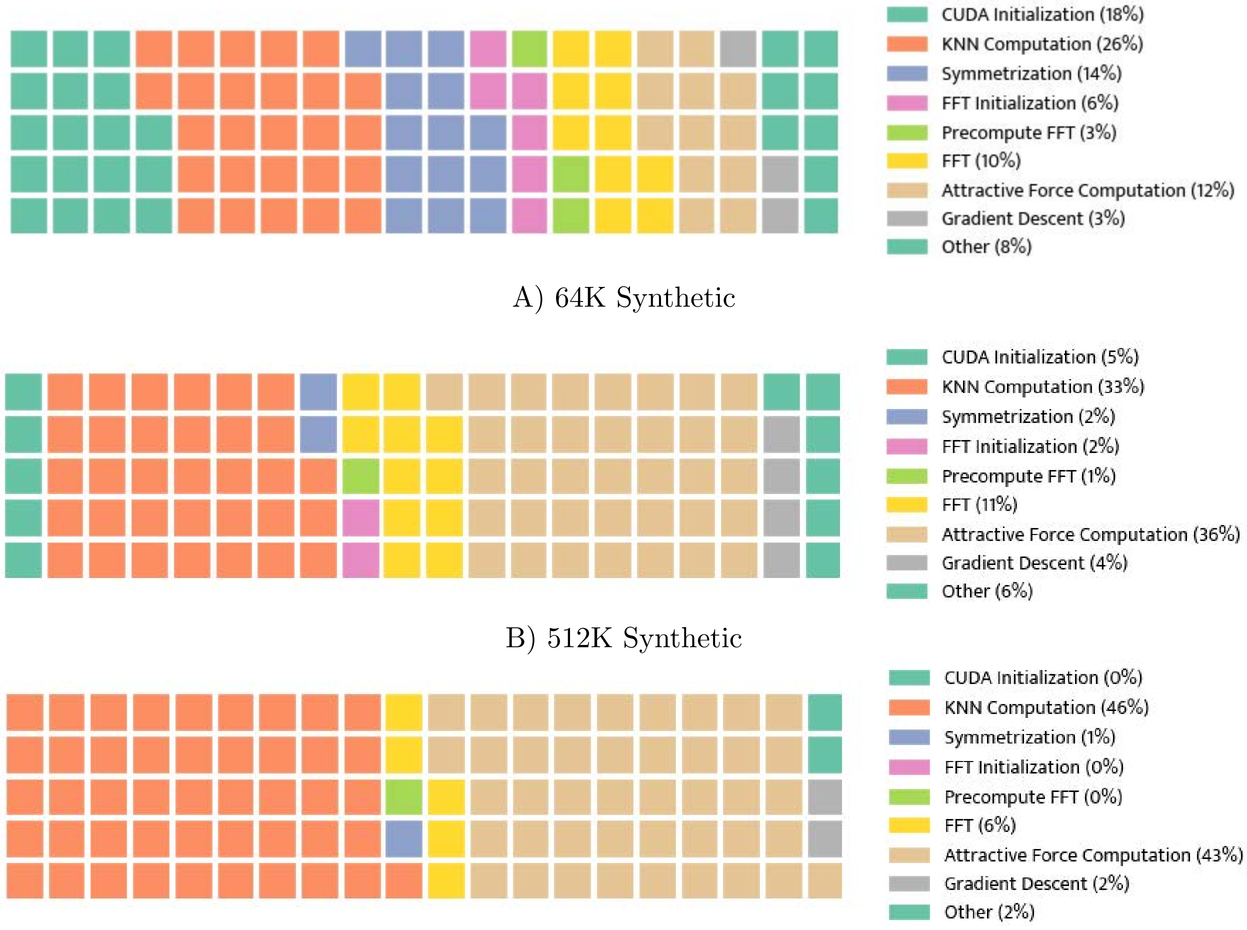


Figure 4.7: Performance on the synthetic dataset for different GPUs.

This same issue occurs during the computation of the attractive forces. Because the attractive forces require a random look-up of the nearest neighbors, the GPU spends a majority of its time waiting on memory reads, as opposed to performing computations (See Figure 4.9). The attractive force computation can be parallelized across devices for the point set, however, the memory copies between devices can be limiting when dealing with fewer than 20 million points.

We can also see that over time the FFT computation does not grow significantly in cost. The reason for this is due to the underlying implementation flexibility of the FFT computation grid. As we scale the number of points, tsnecuda (Like FIt-SNE [37]) does not scale the number of interpolation points on the FFT grid, meaning that the computation required to compute the repulsive forces grows slowly with respect to N. By scaling the number of interpolation points, we can create much better global representations of the dataset, however, this will cause an additional slowdown in the repulsive forces kernel. Future work needs to expose this behavior to the users in the python wrapper, to allow them to make this trade-off effectively.

Figure 4.9 gives a breakdown of the reasons that kernels stall in the tsnecuda implementation, and can help us to identify room for future improvement. As discussed in the previous paragraphs, we can see those memory dependencies contribute to the vast majority of the stalls. This is heavily due to the random read behavior of the attractive forces kernel discussed above. It also suggests that by parallelizing the computation further, and by more clearly defining memory accesses, we could improve the performance of the compute on the GPU.



A) 64K Synthetic

B) 512K Synthetic

C) 5M Synthetic

Figure 4.8: Individual kernel times for the synthetic datasets. Each square represents 1% of the total computation time (The X and Y axes have no meaning). We can see that over time, the nearest neighbor and attractive force computations begin to dominate the computation-time distribution of the algorithm.

4.4 Understanding the Effects of Approximation

4.4.1 Nearest Neighbors

In all of our experiments above, we use the standard 32 approximate nearest neighbors for our computations as most current algorithms use this number of approximations. While this is the industry standard, we actually find that by increasing the number of neighbors, we are seeing markedly different behavior. Figure 4.10 shows the final embedding for the MNIST dataset for differing numbers of neighbors (holding all other variables constant).

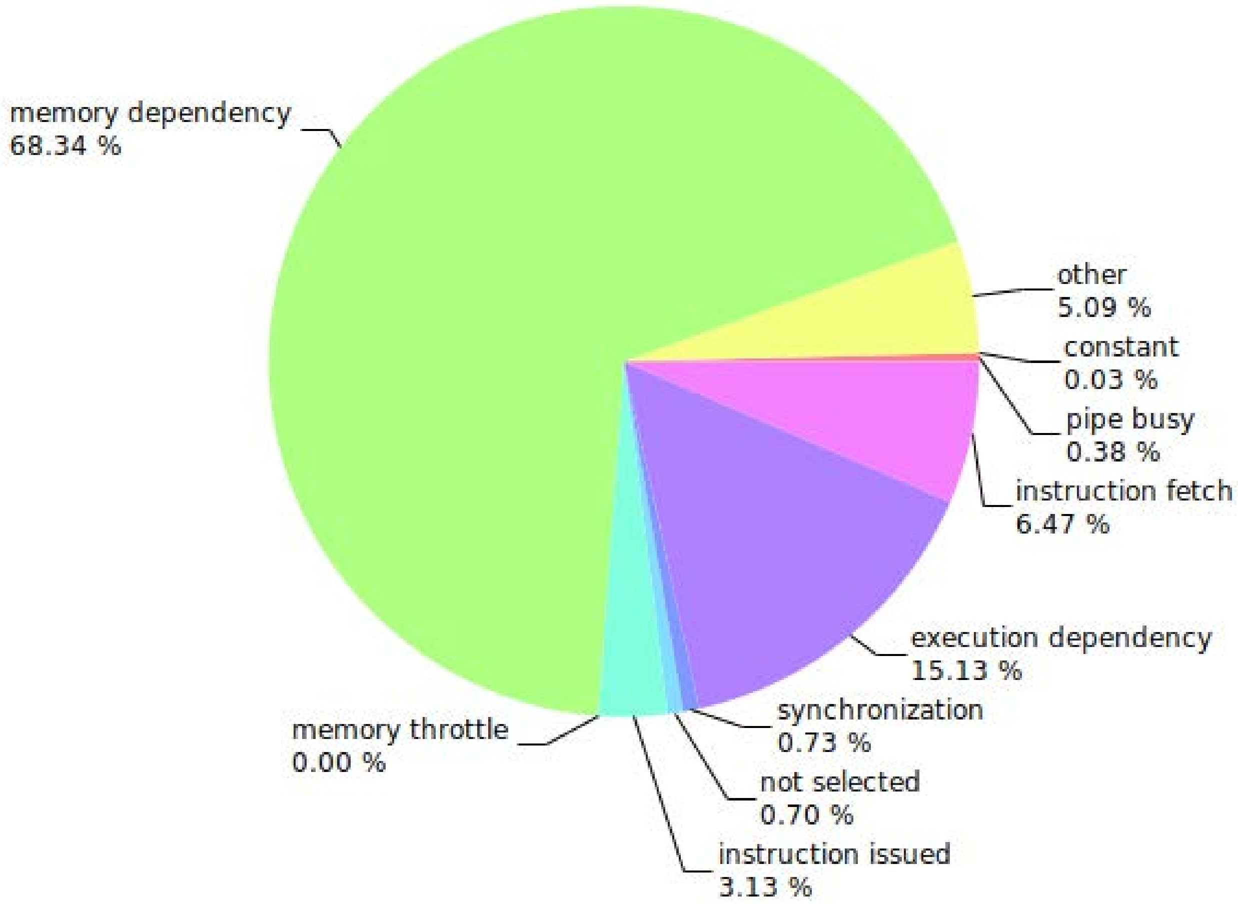


Figure 4.9: Reasons for kernel stalls in the tsnecuda implementation.

We can see that for 4 neighbors, the embedding quality breaks down, and the local clusters dominate the scene. Global clusters aren't well-formed at all for 4 neighbors. When we move to eight to sixteen neighbors, we can see the formation of local clusters, however, the clusters are somewhat separated. There are several distinct clusters for each number, and they are relatively disjoint. Moving to 32 and 64 neighbors, we see the distinct shapes from the MNIST dataset t-SNE embedding that we are familiar with. Using 512 neighbors seems indistinguishable from using the full pairwise labels (See Figure 4.4). Thus, there is some evidence that we are using too few neighbors in standard algorithms to achieve the best quality approximations. Tsnecuda, thus, can help by allowing researchers to perform higher-fidelity approximations in a reduced time.

We can see from 4.11 that the computation time increases with additional nearest neighbors, growing linearly with the number of neighbors that we are working with. This tracks with the attractive force computation being the largest share of the computation time for

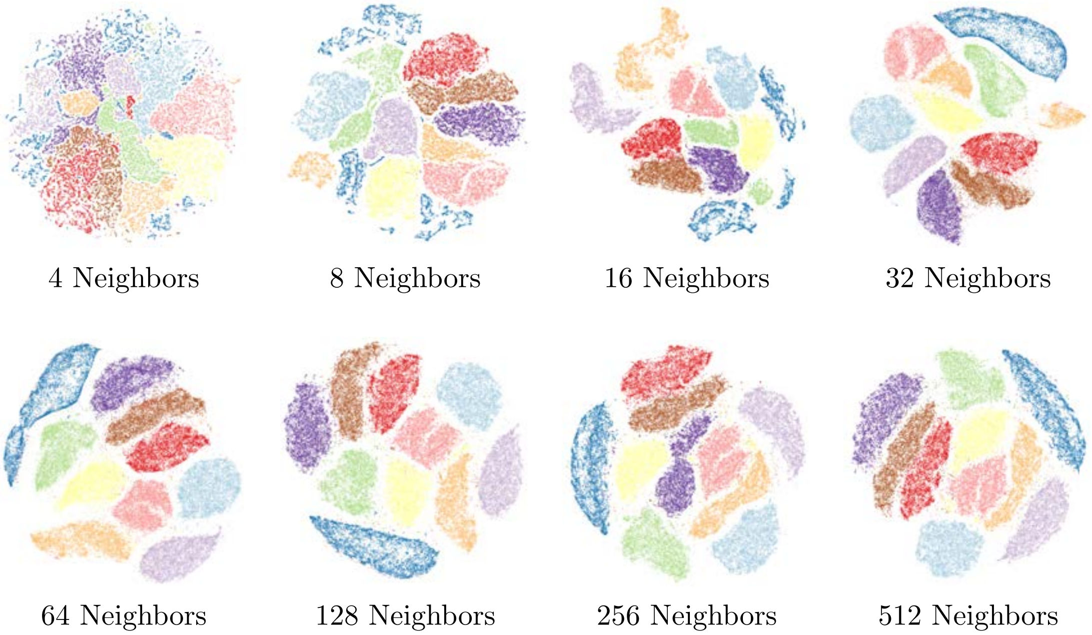


Figure 4.10: The effect of number of nearest neighbors on the computed MNIST embeddings using the IVFADC approximate nearest neighbor algorithm.

the embedding (See Figure 4.8), as it is an $O(uN)$ operation, where u is the number of neighbors.

4.4.2 Product Quantization vs. Flat Index

It's also interesting to explore how the choice of product quantization algorithm affects the embedding structures. Figure 4.12 gives two MNIST embeddings using different nearest-neighbor strategies. While the IVFADC embedding was computed faster, the structures are less compelling and do not demonstrate as accurately the computing structure. tsnecuda trades off between the two nearest neighbor methods as the number of points grows larger, to maintain performance while still generating high-quality embeddings.

4.4.3 Number of Interpolation Points

In our experiments, we follow [37], and choose an interpolation grid of 150x150 equispaced points in the lower-dimensional space for the repulsive force approximation. Figure 4.13 shows the effect of using different sized interpolation grids. Clearly, using interpolation grids that are too small causes artifacts to appear in the data. Using higher fidelity interpolation grids allows the data to more closely mirror the true distributions, but can cause cluster artifacts to become more apparent.

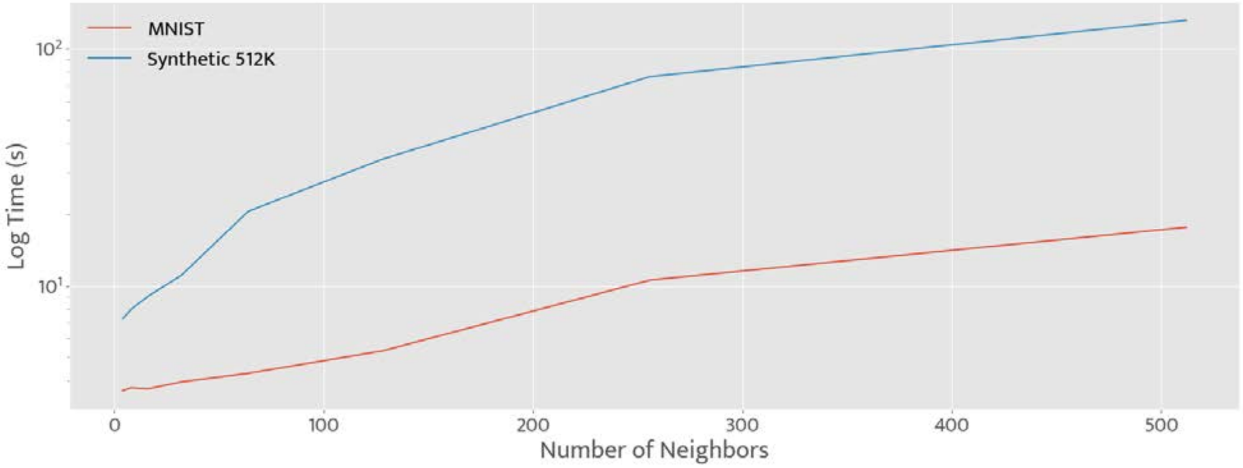
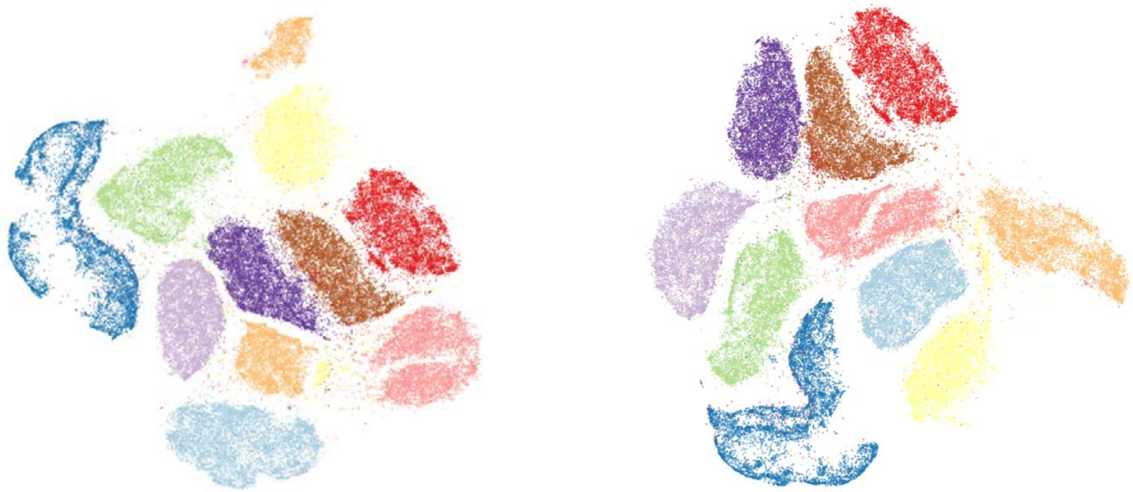
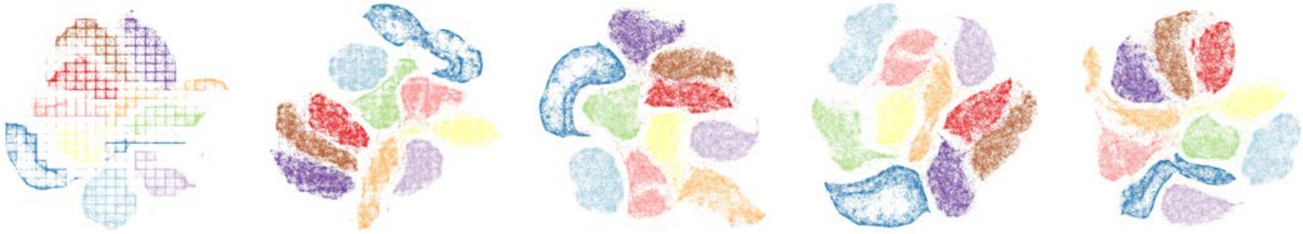


Figure 4.11: Embedding time for different numbers of neighbors in MNIST and the 512K Synthetic Dataset.



A) Inverted index (Computed in 2.09s on 2 GPUs) B) IVFADC (Computed in 1.71s on 2 GPUs)

Figure 4.12: The effect of using different indexing strategies on embedding quality.



A) 25x25 Grid B) 50x50 Grid C) 75x75 Grid D) 150x150 Grid E) 375x375 Grid

Figure 4.13: The effect of using different size interpolation grids for the Lagrange polynomial in FIt-SNE.

Chapter 5

Applications of GPU Accelerated T-SNE

In this section, we discuss several applications of tsnecuda and their implications for modern data. In Section 5.1, we discuss interactive training and guidance of models with real-time t-SNE, and how real-time t-SNE can be used to explore the training process. In Section 5.2, we talk about how tsnecuda is used to infer relationships in word-embeddings spaces such as GloVe [55], and how we can explore large text datasets in NLP. Finally, in Section 5.3, we analyze the difficulty of several publicly available image datasets and examine the insights we can make from running t-SNE at the raw-pixel scale.

5.1 Real-Time Interactive Training

As t-SNE-CUDA can generate visualizations of the high-dimensional features of deep machine learning models faster than the per-epoch training time, we can use it to build interactive visualization of these features. These visualizations help users to understand, diagnose, and improve the training process of deep neural networks.

5.1.1 Visualizing Neural Network Codes During Training

Effectively visualizing and interpreting the training process of deep neural networks is often difficult due to a large number of samples and network weights. With tsnecuda, we can generate a visualization of the network activation for all data points in the training set during the training process. By examining the individual clusters of the codes over time, we can explore the misclassified samples, and provide a human-interpretable visualization of the training process.

Central to the idea of interactive-visualization is the ability to react to, and explore, data in near real-time. Tsnecuda is the first implementation of t-SNE which can generate a visualization faster than a training epoch of the Inception-V3 on the entire ImageNet [2]

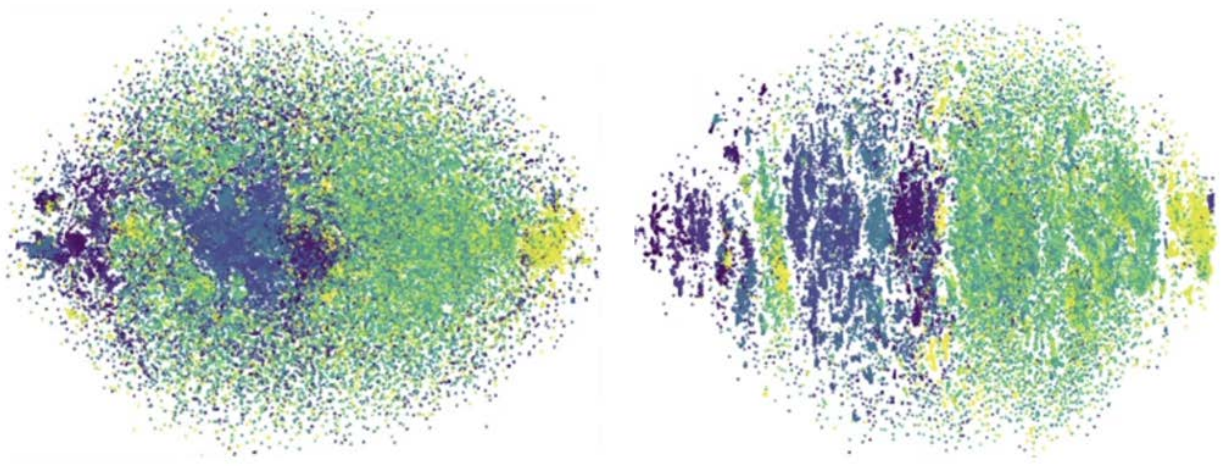


Figure 5.1: Inception-V3 [56] codes on the ImageNet Validation set [2] during different stages of training. Left: Codes at Epoch 2, Right: Codes at Epoch 20. Each was computed in under 5 min.

training set. With such efficiency, we can generate continuous visualization by initializing the t-SNE computation of the current epoch with the visualization from the previous epoch. Figure 5.1 shows the evolution of the clusters across epochs. We can see that over time, the codes are coalescing similar classes, which leads to better test-time performance.

5.1.2 Data Point Importance and t-SNE

Intuitively, it makes sense that points that lie close to the decision boundary (those that are very close together in high-dimensional space) are those which would be difficult to classify for a neural network. This suggests that the boundary points of the clusters in the lower-dimensional space should have more impact on the training process, and thus, should be the target of machine learning algorithms.

This interpretation of relative difficulty is, however, based only on intuition. In order to validate this, we computed the importance scores for each point using the algorithm given by Katharopoulos and Fleuret [57]. These importance scores, when used to weight samples during training, are shown to improve convergence speed and test error. In Figure 5.2 we visualize the importance scores of the LeNet [52] codes of each CIFAR-10 [51] data point in the t-SNE plot at various epochs during training. In the visualization, the colors of the data points represent the classes they belong to, and higher transparency represents lower importance during training. We observe that at epoch two the data-points at the boundaries of the clusters, and data points in the center where the t-SNE plot does not demonstrate clear clusters of colors, are considered to be most important. Similarly, at epoch five, the data points at the boundaries of the clusters are more important for training. This agrees with

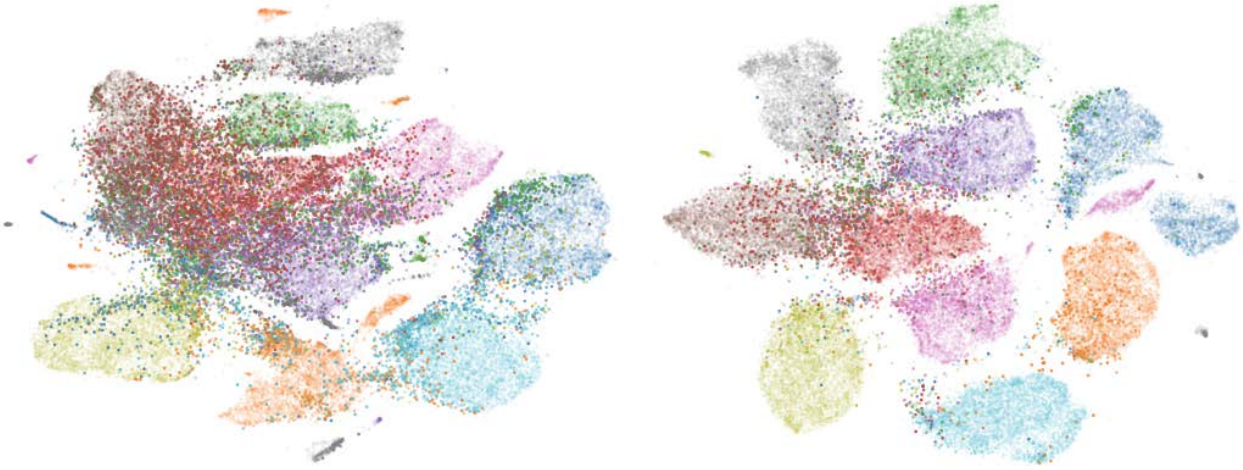


Figure 5.2: LeNet [52] code on the CIFAR-10 [51] training set with importance score represented as transparency of the points. Higher transparency represents lower importance of the data-point to the training process. Left: Codes at Epoch 3, Right: Codes at Epoch 5.

our intuition and demonstrates t-SNE plots' high interpretability and interaction potential for user-steered training of machine learning models.

5.1.3 Selective Training of Data Points

The ability for users to visually identify important data points on generated t-SNE plots demonstrates the potential for t-SNE driven interactions during the training process. To investigate this, we implemented an interactive training tool to explore the possibility of accelerating the training process of networks by allowing users to select the important data points based on the t-SNE clusters. We then examined the impact of a weighted training scheme which weighted selected samples more on the models' performance.

We performed our evaluation using the LeNet [52] model on the CIFAR-10 [51] dataset. During training, we generated the t-SNE plot, and then visually selected points on the boundary to weight higher during the training of the next epoch. With selective training, the network successfully reached the same training loss with 10% fewer training steps and reached a lower training loss upon convergence. We believe these preliminary results, presented in Chan, Rao, Huang, *et al.* [36], show that human-in-the-loop training using interactive real-time visualization and targeting techniques can be an interesting and promising area of research.

5.2 Exploring t-SNE for NLP Problems

In this section, we explore several applications of t-SNE to NLP problems.

5.2.1 Visualizing GloVe Vectors

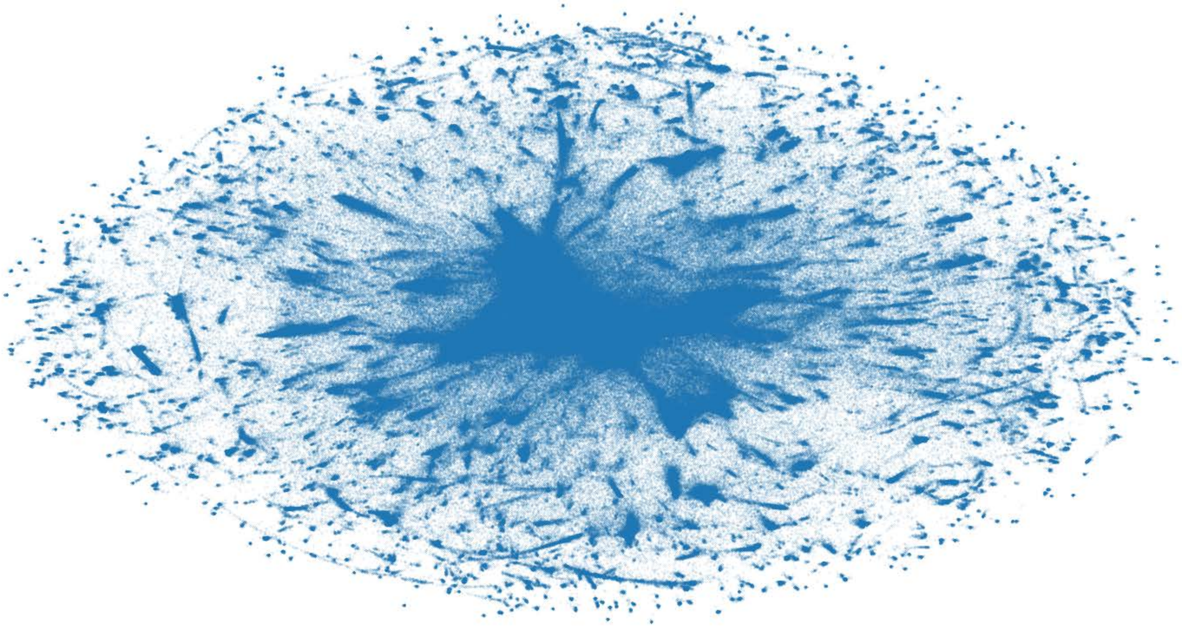


Figure 5.3: GloVe [55] vectors embedded with tsnecuda (Computed in 63.18s with tsnecuda).

The GloVe embedding [55] is a natural language dataset with a vocabulary of over 2.2M words, each embedded in 300-dimensional space. GloVe is a word-similarity embedding trained on 840B tokens found around the internet.

Figure 5.3 shows a coarse plot of the t-SNE that we computed across the *entire* GloVe vocabulary. Our GloVe embedding was computed in 63.18s with the updated version of tsnecuda with 2 GPUs, while the former version of tsnecuda took over 500s to compute the same embedding. As far as we know, this is the first time that the entire 2.2M dataset has been visualized[1]. While there are nice clusters of textually similar data (such as french words, dates, and times), semantic clusters seem less prevalent in the embedding space, and clusters appear to be heavily influenced by hamming distance, and semantic similarity is not always the dominating factor.

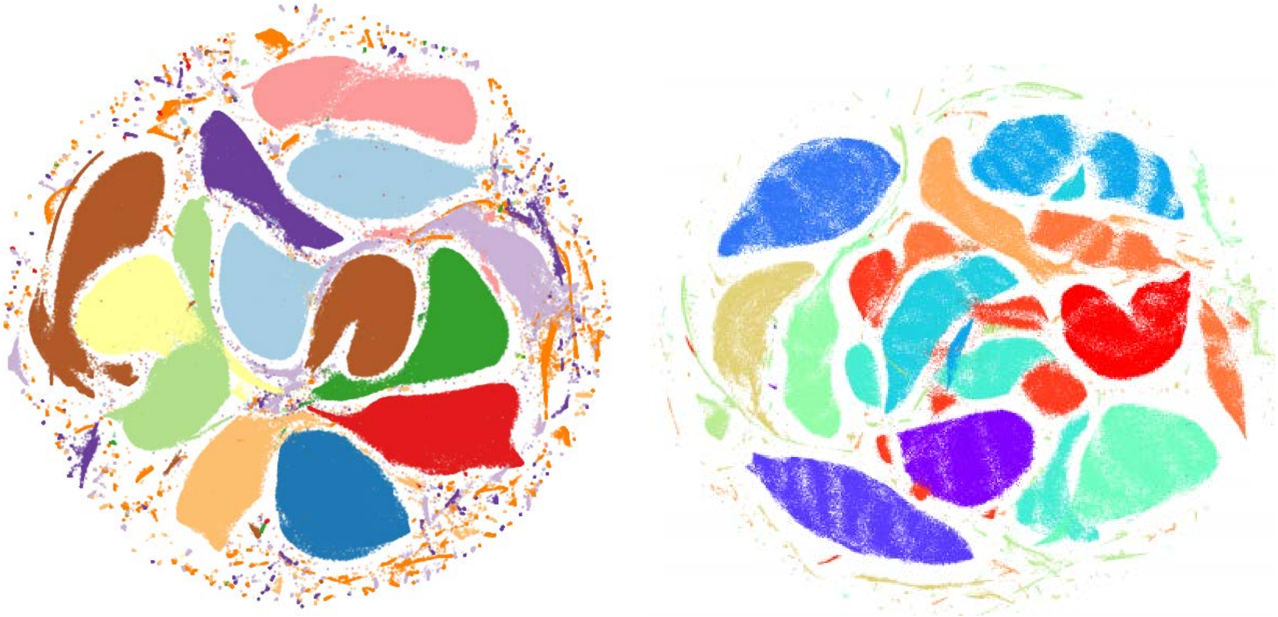
For example, we notice a cluster of dates in the embedding of the form “day-month-year” (Eg. 8-11-2001), as well as a cluster of similar dates of the form “day/month/year” (Eg. 8/11/2018). We confirm that the string 7-11-2001 is closer to 8-11-2001 than the string 8/11/2018, which suggests that GloVe has not captured the date-semantics, and is relying on the hamming distance to fill the gap. Similarly, while ”Singer” and ”Vocalist” are close

[1] This embedding also appears in our previously published works: [36] and [39], so technically, this figure is not novel.

together, "Singer" and "Stinger" are closer in GloVe space than "Singer" and "Alto", a similar term. Thus, by computing the embedding, we were able to draw some insight into the organization of the GloVe space, which was not simple or easy to see from the vector list alone.

5.2.2 Exploring DBPedia

DBPedia [58] consists of 560,000 Wikipedia articles with 14 categories. Each article is represented by a 100-D vector generated by FastText [59]. Figure 5.4 shows a comparison between the AtSNE embedding DBPedia and the tsnecuda embedding of DBPedia. While being almost 14 times faster than AtSNE, tsnecuda can generate sufficiently high-quality embeddings which to be useful for exploratory research purposes.



A) tsnecuda embedding (Computed in 13.32s)

B) AtSNE embedding (Computed in 180s)

Figure 5.4: tsnecuda vs. AtSNE when embedding DBPedia.

5.3 Exploring the Difficulty of Image Datasets

While it is natural to expect that the CIFAR-10 dataset is much harder than MNIST due to its dimensionality, this reasoning lies more in intuition than it does in experimentation. The improved efficiency of t-SNE-CUDA allows us to perform pixel-level experimentation in datasets such as CIFAR-10 [51], whereas previously only embedding-level experiments were possible. This allows us to gain additional insight into the reason for CIFAR-10 being a much harder classification problem than MNIST. Since CIFAR-10 is composed of 32x32x3 images, at a pixel level CIFAR has 50K images at 3072 dimensions. Figure 5.5 shows a t-SNE embedding of the raw pixels CIFAR-10 dataset. We can see immediately from this experiment why classification is much easier on the MNIST dataset. As shown by Figure 4.4, MNIST has a very clear nearest neighbor structure under the L2 metric in pixel space. In Figure 5.5, we see that CIFAR does not have the same structure - images that are close in pixel space are likely of many different classes.

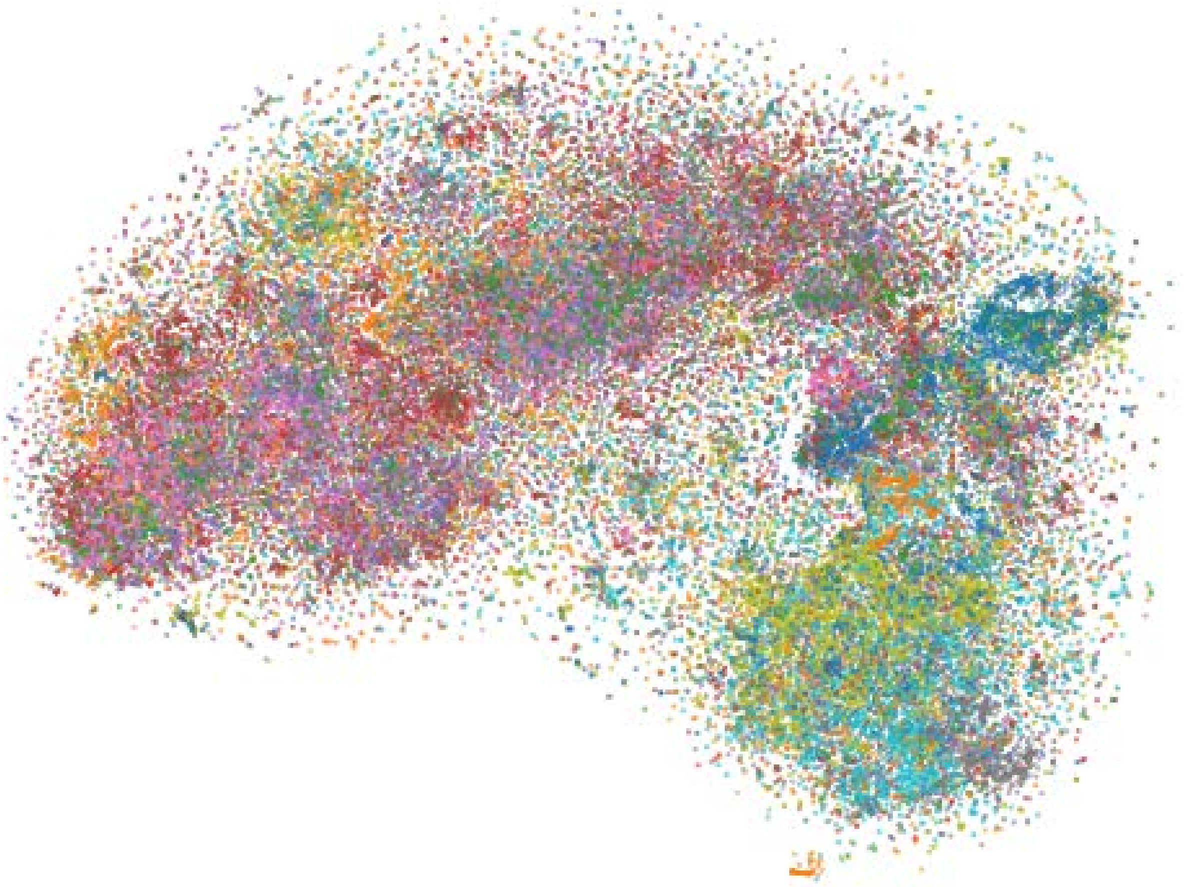


Figure 5.5: Raw pixel-space embedding of CIFAR-10 computed using tsnecuda.

While Figure 5.5 shows that there is some local pixel structure in the dataset, the pixel structure is not as well defined as in the MNIST dataset. Thus, we cannot expect a simple nearest neighbor in the euclidean space to perform well in classification, and we need a non-linear embedding to properly structure the space. Figure 4.6 shows that our non-linear embeddings provide a better L2 structure for our code, making the nearest neighbor classifier in the code-space more efficient (and validating the power of transforming the data with a neural network).

5.3.0.1 ImageNet

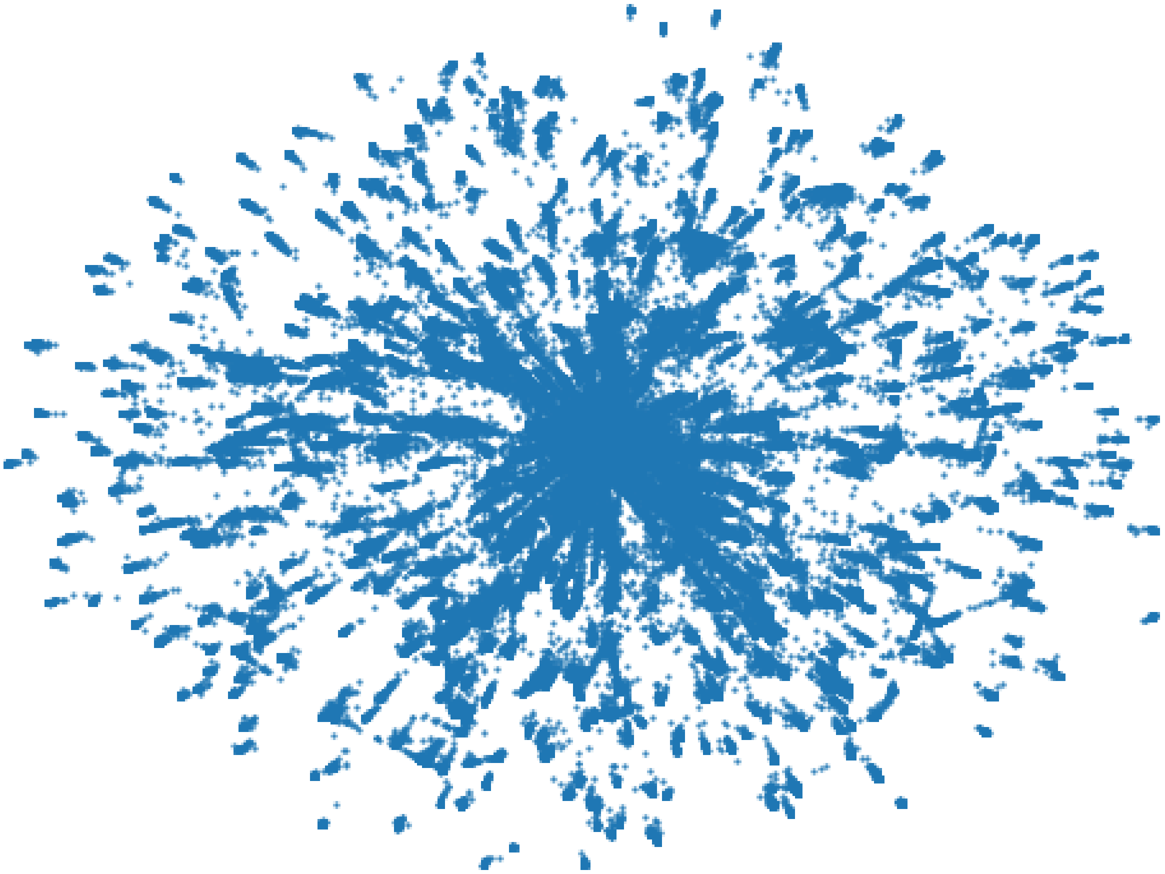


Figure 5.6: Raw pixel-space embedding of VGG Imagenet codes computed using tsnecuda (Computed in 308s).

The ImageNet ILSVRC15 dataset [60] is a large-scale image dataset that is particularly popular in computer vision research. ILSVRC15 is composed of 1.2M 224x224x3 full-color images. It remains an interesting challenge to explore the ways that different neural networks construct embedding spaces of ILSVRC15. While some previous work has explored codes on the ILSVRC15 validation set [61] - such explorations do not provide a full picture of the embedding space of such a large dataset. Figure 5.6 shows the embedding of the VGG19

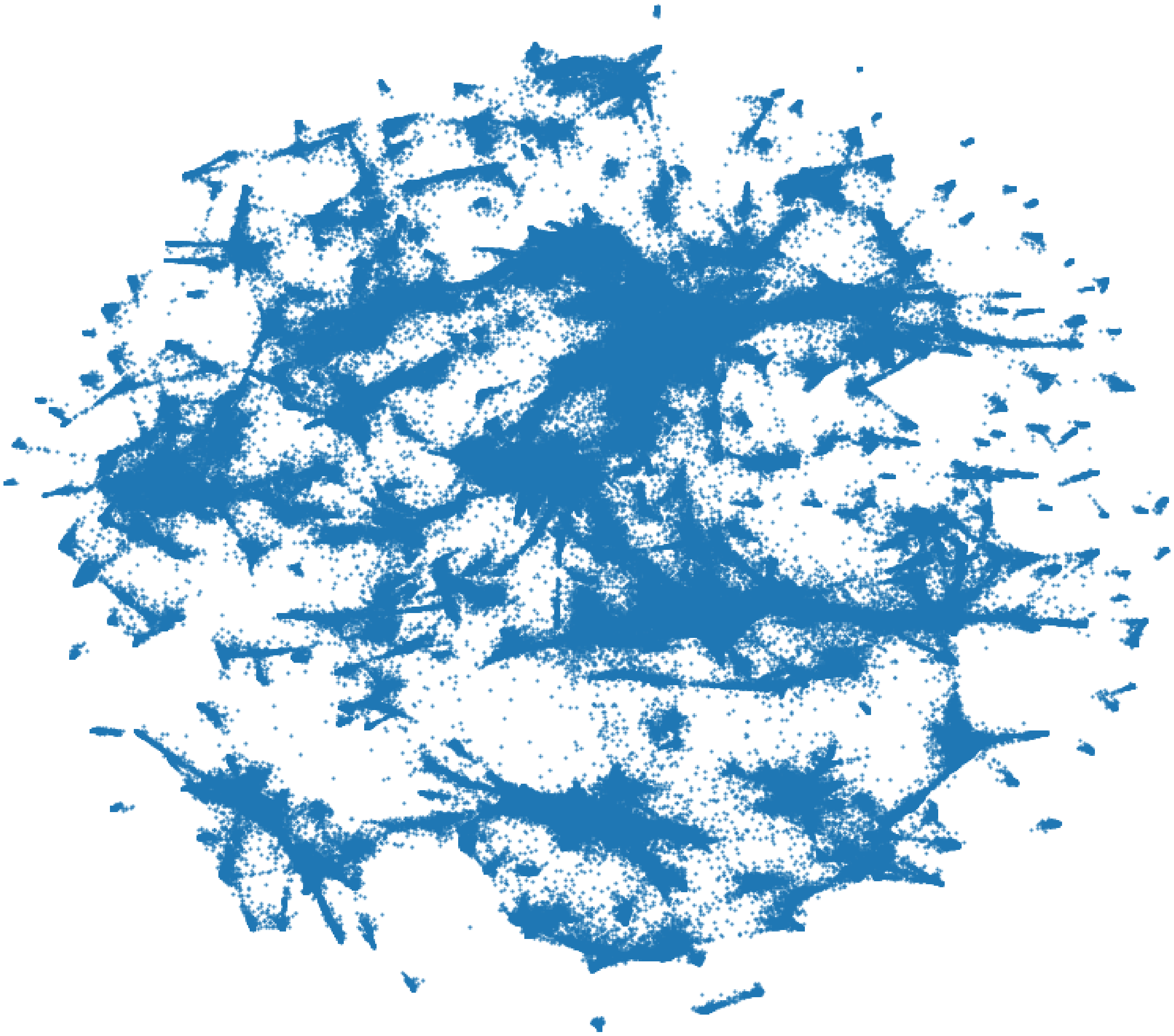


Figure 5.7: Raw pixel-space embedding of ResNet Imagenet codes computed using tsnecuda (Computed in 112s).

[62] 4096 dimensional codes for the entire ILSVRC15 dataset, while Figure 5.7 shows the embeddings using ResNet-200 [63].

An interesting aside is that there are many small, tight clusters in the VGG embedding - each corresponding to a different class. In the ResNet embedding, on the other hand, larger clusters are connected by intermediary data points. We find in general that these interconnected clusters correspond to coarser classifications such as "animals" or "machines." Such more general relationships are not as common in the $L2$ embedding of the VGG codes. These wispy connections suggest that the ResNet embedding space may be more continuous than the VGG embedding space, with points having more inter-class neighbors, while VGG separates classes more discretely. We can, thus, begin to use the information provided by t-SNE-CUDA to help explore some of the local patterns present in large data/embedding spaces.

Chapter 6

Conclusion and Future Work

In this work, we reintroduce an optimized version of tsnecuda, a GPU-accelerated t-Distributed Stochastic Neighbor Embedding algorithm. While tsnecuda continues to perform among the state of the art algorithms for t-SNE, we believe there are a large number of additional areas of exploration as we move into the future.

The most important area of improvement to the algorithm would be in the amount of memory that tsnecuda consumes during execution. In Section 4.2, we discuss how AtSNE and GPGPU t-SNE can significantly outperform tsnecuda in memory usage (AtSNE uses less than 20% of the memory that tsnecuda does). While we have put significant effort into reducing the memory footprint of the optimized algorithm presented in this work, there is still a ways to go to achieve the same memory footprint as these memory-optimized algorithms.

Another clear area of performance improvement could come from coalescing memory accesses in the attractive forces computation kernel. Because this kernel is largely limited by the performance of the reduction step, which causes a large number of random reads across GPU memory, it can be difficult to optimize. It is possible, however, that by computing a K-means clustering and storing local points on each GPU, we would be able to efficiently tile the memory across multiple GPUs, leading to significant performance increases.

This work represents the belief that researchers should have access to fast implementations of trivially parallelizable algorithms. It is thus exciting and interesting future work to bring forth GPU implementations of other embedding algorithms, most notably UMAP [24], which has recently begun to overtake t-SNE in popularity, primarily due to its efficiency in visualization on machines that do not have GPUs. These algorithms are often embarrassingly parallel, and we can save researchers hundreds of hours of waiting by implementing such algorithms in efficient ways.

To conclude, in this thesis we have both introduced tsnecuda, and we have demonstrated how the t-SNE algorithm can be optimized by using product-quantization to approximate the nearest neighbors of higher-dimensional data points and the FIt-SNE method to approximate gradient computation of t-SNE repulsive forces. With these optimizations, we achieved over 10x speedup over state-of-the-art t-SNE implementations and over 650x over the popular SkLearn library. This speedup enables us to explore previously intractable problems - both in the context of vision and NLP problems.

t-SNE-CUDA is publicly available at `https://github.com/CannyLab/tsne-cuda`.

Bibliography

[1] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne", *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database", in *CVPR09*, 2009.

[3] J. Li, X. Chen, E. Hovy, and D. Jurafsky, "Visualizing and understanding neural models in nlp", *arXiv preprint arXiv:1506.01066*, 2015.

[4] R. K. Samala, H.-P. Chan, L. M. Hadjiiski, M. A. Helvie, K. H. Cha, and C. D. Richter, "Multi-task transfer learning deep convolutional neural network: Application to computer-aided diagnosis of breast cancer on mammograms", *Physics in Medicine & Biology*, vol. 62, no. 23, p. 8894, 2017.

[5] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning", in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3357–3364. DOI: 10.1109/ICRA.2017.7989381.

[6] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks", in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015, pp. 4489–4497. DOI: 10.1109/ICCV.2015.510.

[7] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition", in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14, Beijing, China: JMLR.org, 2014, pp. I-647–I-655. [Online]. Available: http://dl.acm.org/citation.cfm?id=3044805.3044879.

[8] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. A. Ranzato, and T. Mikolov, "Devise: A deep visual-semantic embedding model", in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2013, pp. 2121–2129. [Online]. Available: http://papers.nips.cc/paper/5204-devise-a-deep-visual-semantic-embedding-model.pdf.

[9] H. Izadinia and P. Garrigues, "Viser: Visual self-regularization", Feb. 2018.

[10] D. Sacha, M. Sedlmair, L. Zhang, J. A. Lee, J. Peltonen, D. Weiskopf, S. C. North, and D. A. Keim, "What you see is what you can change: Human-centered machine learning by interactive visualization", *Neurocomputing*, vol. 268, pp. 164–175, 2017.

[11] N. Pezzotti, T. Höllt, J. Van Gemert, B. P. Lelieveldt, E. Eisemann, and A. Vilanova, "Deepeyes: Progressive visual analytics for designing deep neural networks", *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 98–108, 2018.

[12] T. Mühlbacher, H. Piringer, S. Gratzl, M. Sedlmair, and M. Streit, "Opening the black box: Strategies for increased user involvement in existing algorithm implementations", *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 1643–1652, 2014.

[13] H. Zou, T. Hastie, and R. Tibshirani, "Sparse principal component analysis", *Journal of Computational and Graphical Statistics*, vol. 15, no. 2, pp. 265–286, 2006.

[14] V. Rokhlin, A. Szlam, and M. Tygert, "A randomized algorithm for principal component analysis", *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 1100–1124, 2010.

[15] M. A. Cox and T. F. Cox, "Multidimensional scaling", in *Handbook of Data Visualization*, Springer, 2008, pp. 315–347.

[16] J. W. Sammon, "A nonlinear mapping for data structure analysis", *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 401–409, 1969.

[17] P. Demartines and J. Hérault, "Curvilinear component analysis: A self-organizing neural network for nonlinear mapping of data sets", *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 148–154, 1997.

[18] G. E. Hinton and S. T. Roweis, "Stochastic neighbor embedding", in *Advances in Neural Information Processing Systems*, 2003, pp. 857–864.

[19] J. B. Tenenbaum, V. De Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction", *science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[20] K. Q. Weinberger and L. K. Saul, "An introduction to nonlinear dimensionality reduction by maximum variance unfolding", in *AAAI*, vol. 6, 2006, pp. 1683–1686.

[21] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding", *science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[22] M. Belkin and P. Niyogi, "Laplacian eigenmaps and spectral techniques for embedding and clustering", in *Advances in Neural Information Processing Systems*, 2002, pp. 585–591.

[23] K. Larsen and J. Nelson, "Optimality of the Johnson-Lindenstrauss lemma", in *58th Annual IEEE Symposium on Foundations of Computer Science*, 2017, pp. 633–638.

[24] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction", *arXiv preprint arXiv:1802.03426*, 2018.

[25] S. Arora, W. Hu, and P. K. Kothari, “An analysis of the t-sne algorithm for data visualization”, *arXiv preprint arXiv:1803.01768*, 2018.

[26] S. Kullback and R. A. Leibler, “On information and sufficiency”, *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[27] L. Van Der Maaten, “Accelerating t-sne using tree-based algorithms.”, *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3221–3245, 2014.

[28] P. N. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces”, 1993.

[29] G. Dimitriadis, *Spikesorting_tsne*, `https://github.com/georgedimitriadis/spikesorting_tsne`, 2018.

[30] R. Shah and S. Silwal, “Using dimensionality reduction to optimize t-sne”, *arXiv preprint arXiv:1912.01098*, 2019.

[31] N. Pezzotti, B. P. Lelieveldt, L. van der Maaten, T. Höllt, E. Eisemann, and A. Vilanova, “Approximated and user steerable tsne for progressive visual analytics”, *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 7, pp. 1739–1752, 2017.

[32] M. Muja and D. G. Lowe, “Fast approximate nearest neighbors with automatic algorithm configuration”, *VISAPP*, vol. 2, no. 331-340, p. 2, 2009.

[33] M. Burtscher and K. Pingali, *An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm*. 2011, pp. 75–92. [Online]. Available: `http://iss.ices.utexas.edu/Publications/Papers/burtscher11.pdf`.

[34] D. Ulyanov, *Multicore-tsne*, `https://github.com/DmitryUlyanov/Multicore-TSNE`, 2016.

[35] S. Raschka, J. Patterson, and C. Nolet, “Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence”, *arXiv preprint arXiv:2002.04803*, 2020.

[36] D. M. Chan, R. Rao, F. Huang, and J. F. Canny, “T-sne-cuda: Gpu-accelerated t-sne and its applications to modern data”, *arXiv preprint arXiv:1807.11824*, 2018.

[37] G. C. Linderman, M. Rachh, J. G. Hoskins, S. Steinerberger, and Y. Kluger, “Efficient algorithms for t-distributed stochastic neighborhood embedding”, *arXiv preprint arXiv:1712.09005*, 2017.

[38] G. Linderman, Jan. 2018. [Online]. Available: `https://gauss.math.yale.edu/~gcl22/blog/numerics/low-rank/t-sne/2018/01/11/low-rank-kernels.html`.

[39] D. M. Chan, R. Rao, F. Huang, and J. F. Canny, “Gpu accelerated t-distributed stochastic neighbor embedding”, *Journal of Parallel and Distributed Computing*, vol. 131, pp. 1–13, 2019.

[40] N. Pezzotti, J. Thijssen, A. Mordvintsev, T. Höllt, B. Van Lew, B. P. Lelieveldt, E. Eisemann, and A. Vilanova, "Gpgpu linear complexity t-sne optimization", *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 1172–1181, 2019.

[41] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning", in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[42] C. Fu, Y. Zhang, D. Cai, and X. Ren, "Atsne: Efficient and robust visualization on gpu through hierarchical optimization", in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 176–186.

[43] E. Bernhardsson, "Annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk", *GitHub https://github. com/spotify/annoy*, 2017.

[44] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[45] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus", *IEEE Transactions on Big Data*, 2019.

[46] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search", in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 950–961.

[47] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in *Proceedings of the thiry-fourth Annual ACM Symposium on Theory of Computing*, 2002, pp. 380–388.

[48] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions", *arXiv preprint cs/0510019*, 2005.

[49] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 2321–2328.

[50] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database", *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[51] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", 2009.

[52] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[53] G. Dimitriadis, *T-sne-bhcuda*, `https://github.com/georgedimitriadis/t_sne_bhcuda`, 2016.

[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[55] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation", in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[56] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[57] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling", in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, 2018.

[58] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data", in *The Semantic Web*, Springer, 2007, pp. 722–735.

[59] E. Grave, P. Bojanowski, P. Gupta, A. Joulin, and T. Mikolov, "Learning word vectors for 157 languages", *arXiv preprint arXiv:1802.06893*, 2018.

[60] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge", *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[61] A. Karpathy, *T-sne visualization of cnn codes*, `https://cs.stanford.edu/people/karpathy/cnnembed/`.

[62] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[63] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.