Exploring the computational feasibility limits of perplexity in t-SNE for scenarios of limited working memory

Dimitar Netzov[1]

Supervisor(s): Klaus Hildebrandt, Martin Skrodzki[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Dimitar Netzov
Final project course: CSE3000 Research Project
Thesis committee: Klaus Hildebrandt, Martin Skrodzki, Christoph Lofi

Abstract

Modern data analysis often involves working with large multidimensional datasets. Visualizing this kind of data helps leverage human intuition and pattern recognition to reveal hidden relationships. t-SNE is a widely used tool for creating such visualizations. Despite its popularity, it suffers drawbacks in the form of hard-to-tune parameters with no heuristic for guaranteed best results. Due to the size of the data researchers have to work with, the algorithm can often exceed the available memory and lead to slowdowns and crashes. This paper investigates the behaviour of memory usage with respect to the tunable parameter perplexity and the size of the data. It provides a reliable way for researchers to predict the memory consumption before running the algorithm for the popular openTSNE implementation of t-SNE. In addition, a modification to reduce the peak memory usage of the implementation is presented. Together, these contributions improve the reliability and efficiency of t-SNE pipelines in memory-constrained environments.

1 Introduction

The need to visualize high-dimensional data is shared across domains such as medicine, biology, and machine learning. For example [6]. A promising tool for this task is t-Distributed Stochastic Neighbor Embedding (t-SNE), a dimensionality reduction technique that maps data into a low-dimensional space for visualization purposes. While t-SNE is widely adopted due to its ability to preserve local structure in the data, it suffers from some computational limitations. In particular, the memory complexity of the basic t-SNE algorithm scales quadratically with the size of the dataset [5], which leads to scalability issues when working with large-scale data.

This problem is especially impactful when tuning the perplexity parameter, a key hyperparameter that affects the quality of the embedding. Perplexity has a direct impact on both computational cost and the final output [1]. Yet, selecting an appropriate perplexity is non-trivial and often involves repeated executions of the algorithm. For large datasets and constrained hardware environments, many of these executions may exceed memory limits, resulting in failed runs or unacceptable runtimes.

Existing literature has acknowledged this limitation. For example, Skrodzki et al. [3] propose to leverage a linear relationship between perplexity and data size to identify suitable perplexities on subsamples of the data. The approach helps determine a desirable perplexity to use on the entire dataset, but it does not guarantee that the chosen value will compute for said dataset on a given hardware. This paper addresses this issue by determining what the highest computationally feasible perplexity for a given dataset and hardware configuration is. The goal is to be able to preemptively rule out values that would exceed available working memory, thereby improving the efficiency and reliability of the t-SNE workflow in constrained environments.

The research focuses on the openTSNE[1] implementation of t-SNE. It investigates:

- Which stages of the algorithm are the most memory-intensive.
- How memory usage can be modeled theoretically, and whether sub-sampling techniques can approximate memory usage.

Empirical measurements during execution are used to verify these models.

By answering these questions, the study contributes both a practical tool for researchers using t-SNE in low-resource settings and a deeper understanding of the algorithm's memory behavior. The outcomes are relevant for applications involving large-scale data, where optimizing computational resources is crucial for achieving accurate visualizations.

2 The t-SNE algorithm

The idea behind t-SNE is to reconstruct the local neighbourhood of the points of the higher-dimensional space in the lower-dimensional space. To do this, it first needs to model said neighborhood. This is done using conditional probabilities. Each point determines the conditional probability of picking its neighbours if it were to pick them based on a Gaussian probability density around the point. In practice, to improve efficiency, only the k nearest neighbors are considered for each point, with k typically set to around 3 times the perplexity value. This sparsification sets all other conditional probabilities to zero, based on the assumption that their influence would be negligible. These conditional probabilities are then symmetrized into joint probabilities, forming the so-called affinity matrix, which captures pairwise similarities in the high-dimensional space.

Using the similarities of the high-dimensional points, the algorithm must now proceed to model a low-dimensional representation of the data that has a matching affinity matrix. In contrast to regular SNE, in t-SNE, a t-student distribution is used for the low-dimensional matrix. The cost function of the two matrices is the Kullback–Leibler divergence between the high and low dimensions. The optimisation step involves moving along the gradient, which can be modeled as each point exerting a force on every other point depending on the mismatch between their similarities. Multiple parameters exist to help improve the results of this stage, like introducing a momentum term and exaggerating the forces, but these do not affect the space complexity.

2.1 Improvements on the time and space complexities

The time and space complexities of the base t-SNE algorithm scale quadratically with N [5]. To improve this, modifications to both the affinity matrix calculation and the embedding process are available.

[1] https://opentsne.readthedocs.io/en/stable/

Starting with the affinities, many implementations resort to approximating the nearest neighbours instead of fully computing them. A popular approach is Approximate Nearest Neighbors Oh Yeah[2] (Annoy), which subdivides the high-dimensional space into separate regions of space based on the density, and generates a tree representation of the subdivision to use in locating neighbours.

In addition, for data sizes much larger than the perplexity, which is often the case in practice, the affinity matrix turns out to be sparse. This means that it can be stored in a data structure meant for sparse data, such as a Compressed Sparse Row (CSR) matrix. This brings the space complexity down from $O(N^2)$ to $O(P \cdot N)$.

2.2 OpenTSNE

OpenTSNE's implementation of t-SNE is the basis of this paper's analysis of the algorithm; as such, it is important to first examine it in some more detail. Most t-SNE implementations can be broadly divided into two parts: the affinity matrix calculation and the fitting of the low-dimensional data using the affinities. The openTSNE library adopts a similar modular structure, where the affinities are created and stored by an instance of the Affinities class, while the optimization is handled by the `TSNE` class.

For the affinity calculation, multiple implementations are offered by default. The `PerplexityBasedNN` class implements the nearest-neighbor search based on a user-specified perplexity value. This is the default method that adheres most closely to the original t-SNE formulation and is the focus of this paper. Each `Affinity` class also has the option to symmetrize the result, which is enabled by default, as this is the standard approach in t-SNE. In addition, different metrics are available to use for the KNN calculations, as well as various approaches to the KNN calculation itself, including approximation-based methods. The default approximation method in openTSNE is the Annoy approximation.

The optimisation step also has a handful of configurations. Besides the tunable parameters, there is also an important choice in the algorithm used to compute the forces between the low-dimensional points. Rather than computing exact gradients as in the original t-SNE, openTSNE exclusively uses approximate methods to speed up computation. It supports two acceleration techniques: the Barnes-Hut approximation using quad trees for datasets of moderate size, and FIt-SNE, an FFT-based interpolation method for larger datasets. These approximations significantly reduce runtime while preserving embedding quality. [4] [2].

3 Methodology

3.1 Exact formula

The typical approach when dealing with the memory complexity of an algorithm is to look for the big O. This allows for some broad generalisations about the algorithms' performance that are independent from implementation details. However, in the case of a t-SNE pipeline that operates on limited hardware, the big O is not specific enough, as it obscures coefficients that might matter in a non-asymptotic use case. For this reason, an exact formula for the memory usage is sought in this paper.

3.2 Mixed approach

There are two main approaches to assertaining the space complexity of an algorithm: theoretical and experimental. Theoretical approaches have the advantage of offering an insight into why and which parts of the algorithm behave the way they do. This is helpful when looking for ways to improve the performance. The disadvantage of this method is that it is specific to each implementation and requires a deeper understanding of the source code. In contrast, the analytical approach only relies on measurements, so the specifics of the code are less relevant. Since OpenTSNE is highly adjustable in the implementation specifics, the analytical approach may prove more convenient for further analysis.

There is currently no explicit equation for a t-SNE implementation in literature, so a theoretical analysis can provide valuable insight into the reasons behind the memory load of the algorithm. To address the downsides, an experimental approach was used, where the equation was estimated from measured memory load. It can be used to inform and validate theoretical findings, and in this paper, it serves as a test run for use as a primary method for estimating memory usage in other implementations of t-SNE.

3.3 General setup

The paper will be using the following openTSNE configuration: PerplexityBasedNN to compute affinities, Annoy approximation with Euclidean distance of nearest neighbors as the method to calculate affinities, and Flt-SNE as the embedding method. Memory in all experiments was measured as the Resident Set Size (RSS) of the process in Mb. RSS denotes the amount of memory a process currently has in physical RAM. Experiments were run on Windows 11. Libraries psutil and pympler were used to measure the size of the process and objects, respectively.

4 Results

4.1 Small parameter samples

The data set for theoretical validation and training the ML model was obtained on Windows 11 using the MNIST dataset. The datapoints were the peak RSS measured in the execution of the PerplexityBasedNN init method. Every combination of ten linearly distributed perplexities from 5 to 100 and ten linearly distributed sample rates from 0.01 to 1 where used to get 100 total datapoints. Each datapoint is the average of three runs, for a total of 300 runs.

4.2 Algorithm section analysis

To help ease the theoretical analysis, the algorithm was divided into five distinct sections: the Annoy estimation, the conditional matrix computation, the symmetrisation, the normalisation, and the fitting of the low-dimensional embedding. This division was based on separation in the source code itself, as well as the behaviour of the memory consumption

[2] https://github.com/spotify/annoy

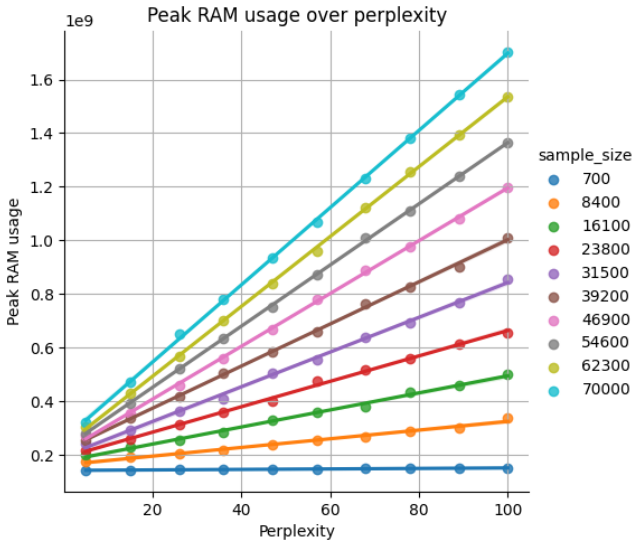

Figure 1: Peak memory usage of test samples with respect to the P parameter

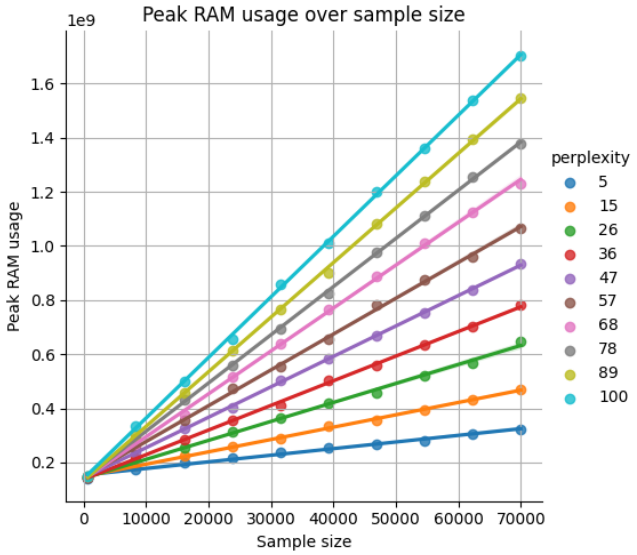

Figure 2: Peak memory usage of test samples with respect to the N parameter

(Fig. 3). To show the different sections of the openTSNE implementation, the library code was modified to return timestamps after each computation.

Initially, the sections gradually increase in memory consumption, peaking at the end of the symmetrisation step, and then go down in the next two, with memory demand increasing over time within the last section - fitting. In said last section, it can be seen that the memory usage increases over time/number of iterations. To avoid introducing another parameter into the equation, the last section of the algorithm will not be discussed further in this paper.

4.3 Theoretical analysis

The goal of the theoretical analysis is to produce an explicit formula for the peak memory usage of the program as a whole. In regular execution, subsequent sections inherit some data from the previous sections, resulting in increased memory usage. To get the maximum space used by a section one has to consider both the actively used data, and the leftover data from previous sections. In the case of openTSNE, the maximum usage corresponds to the symmetrization section, and so all sections before it have to be considered as well.

Plotting the change of peak memory consumption over P (Fig. 1) and N (Fig. 2) reveals a strong linear relationship. This suggests that no significant contributions come from terms of higher power with respect to one of the two parameters. The slope of the dependency also changes with the other parameter, which hints at a $N \cdot P$ term in the final formula. Finally, one last observation is that the peak RSS tends to the same constant for low values of N, but not for low values of P, so a low to no contribution for the P term is expected. The constant component, C, is, of course, highly dependent on the code. This includes the code used to call the openTSNE methods and classes, making it harder, if not impossible, to predict with certainty. Thus, the expected theoretical formula for the space complexity is:

$$a \cdot P \cdot N + b \cdot N + C \tag{1}$$

Let's examine how memory evolves section by section during the program's execution:

Before any code is run, the data itself is passed in as an argument. The required space depends on the dataset, for MNIST that is $8 \cdot 50 \cdot N = 400N$.

To start off the execution, an Annoy index needs to be created in order to obtain the neighbours and distances matrices. In openTSNE, $k = 3 \cdot P$ neighbours are considered. The memory usage of this process peaks at the end of its creation and is equal to $4 \cdot k \cdot N$ for the neighbours and $8 \cdot k \cdot N$ for the distances. The constant factors are the datatype sizes, and are constant for openTSNE, regardless of input data. This gives a total of $36 \cdot P \cdot N$ bytes, in addition to the dataset from before.

What follows are the conditional probabilities. Like the previous section, the peak is at the end, and has k*N entries of 8 bytes each, so $24 \cdot P \cdot N$ bytes. The conditional probabilities are used with the neighbour indices to form a CSR matrix which has $24 * N * P$ bytes from the conditional, and $12 * N * P$ from the indices, equaling $36 \cdot N \cdot P$. With the previous section, that comes to $96 \cdot P \cdot N$.

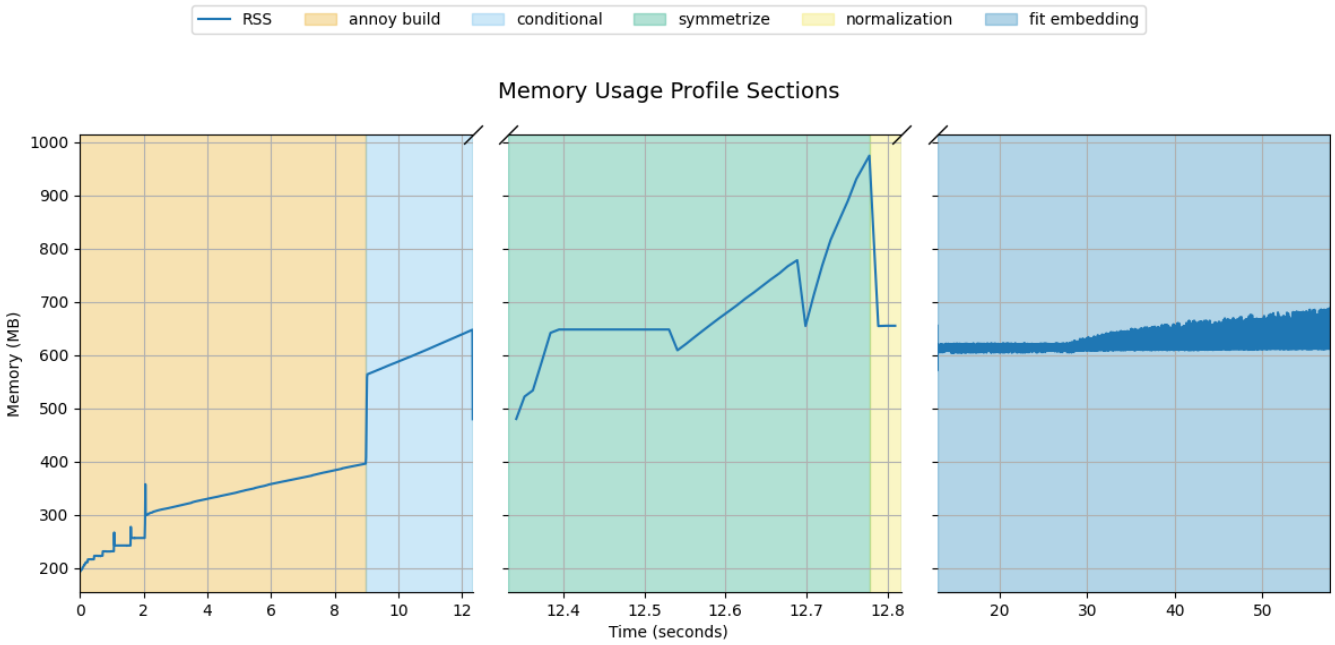

Figure 3: Memory usage profile of sections during execution of a full run with $P = 50$ and $N = 70000$. The middle part of the profile is zoomed in on the time scale for visibility.

```
# Symmetrize the probability matrix
if symmetrize:
    P = (P + P.T) / 2

if normalization == "pair-wise":
    P /= np.sum(P)
elif normalization == "point-wise":
    P = sp.diags(np.asarray(1 / P.sum(axis=1)).ravel()) @ P
```

Figure 4: Problem code snippet in the S&N steps, boosting memory usage

In the next step, due to every step of symmetrisation being done in one line (fig. 4), three additional matrices are created and exist simultaneously in memory, one for the transpose, addition, and division. This means that the absolute peak for the entire program achieved in this section corresponds to $3 \cdot 36 \cdot N \cdot P = 108 \cdot N \cdot P$, or total peak memory of $204N + 400N + C$.

4.4 Analytical comparison

To test the viability of estimating the space complexity from experimental measurements, a training set of datapoints first had to be computed. That was done using the peak memory usage of runs with different combinations of N and P. Both parameters were linearly varied from a small to a medium value to obtain the pool for the combinations. The idea is for the datapoints to be fast to compute, so larger values of N and P would defeat the purpose. Terms of N^2 and P were added, expected to trend to zero, to verify the theoretical results.

To derive the estimated formula, machine learning (ML) was used in conjunction with the obtained data points. A regression model is suitable for this task since the formula is expected to be polynomial. Lasso was chosen as the specific model due to the imposed limit of positive coefficients, which is a real constraint on the coefficients of the actual formula. After fitting, new predictions are made by feeding the input parameters, which can then be used for validation on unseen data. The actual formula can also be obtained from the model, and that can help validate, adjust the theoretical model, specifically when dealing with the value of C.

After training the ML model, the resulting formula is

$$208 \cdot N \cdot P + 1285 \cdot N + 126000000 \tag{2}$$

This is quite similar to the theoretical result in the first term, but differs significantly in the second. The reprojection on the training set is remarkably close (Fig.5), and the mean absolute percentage error (MAPE) is 2.3%, which points to an accurate model. Using the analytical estimate, we can obtain an idea of what value C might take. With this value, the theoretical model can be adjusted and tested against the training set. The results show lower Relative Error, especially for cases of low P and high N. This result is consistent with an inaccurate N term, meaning the analytical result reflects reality more closely for the training set.

4.5 Verification on large parameters

To verify the theoretical and analytical results, the relative error between predictions and measured values was used. This has the advantage of showing whether the expected result is over- or under-estimated, and being more informative about the error in C for different parameters. Additionally, the training set for the ML model was used to validate the theoretical

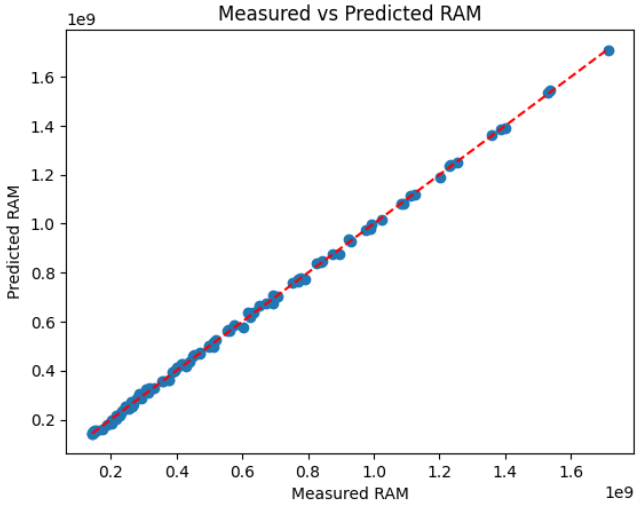

Figure 5: Predicted vs actual for the ML model on the training data, with the identity line

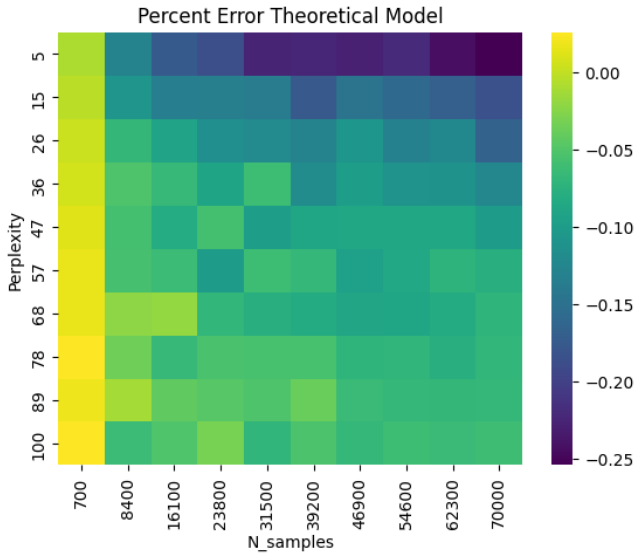

Figure 6: Percentage error of theoretical predictions over the low-spec datapoints, after adjusting for the constant term

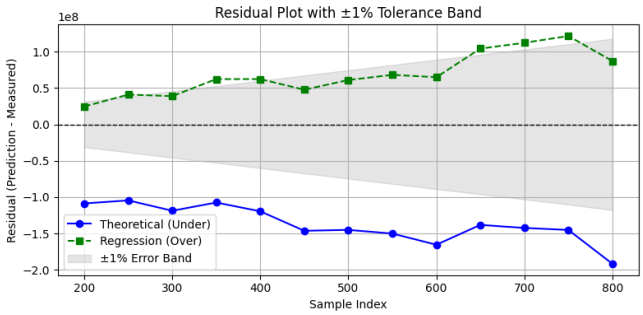

Figure 7: Residual error in bytes of theoretical vs ML model predictions over the high-spec datapoints, with 1% error margin

```
P += P.T
P /= 2
```

Figure 8: Modified symmetrization section

results and to show that the formula indeed follows the expected pattern for the training set.

The theoretical and regression models were tested on higher perplexities on the entire MNIST dataset, with 13 values between 200 and 800 being used. The results show that the regression model tends to stay in the 1% error margin and consistently overestimates, while the theoretical model performs slightly worse and tends to underestimate.

4.6 Improvement of peak memory usage

Let's circle back to the supposed inefficiencies in the symmetrization section (Fig. 4). If doing all the operations in one line causes multiple copies of the CSR matrix to be stored in memory, then would separating the operation into multiple steps help reduce memory usage? To test this, a copy of the class was created that had the line in question split into multiple lines as in Fig. 8. The two versions were then both ran in a setup analogous to the samples obtained in 4.1.

The resulting MAPE per parameter grid (Fig. **??**) shows a clear reduction in peak memory usage for higher values of N and P. No exact quantification for the improvement based on the input parameters is made here, but it still shows that the line separation is effective in achieving the desired performance improvement.

5 Responsible Research

High on the list of priorities for this research was to make the results reproducible. The code base has self-contained dependency data to ensure all tool versions are recorded for future researchers. The main tools and environments were mentioned in the paper and are further documented in the code. Care was taken to seed every function that required random input, as well as documenting the seed.

6 Discussion

This paper set out to explore the computational limits of t-SNE with respect to working memory, focusing particularly

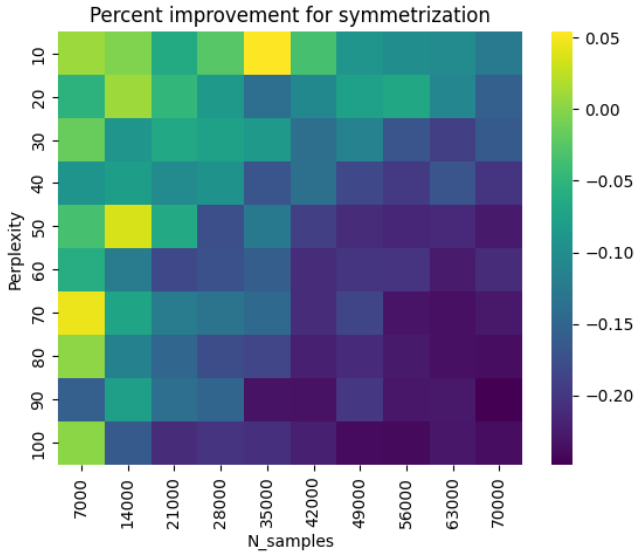

Figure 9: Heatmap of MAPEs per parameter for the improved symmetrization section over the library section. Smaller is better.

on the relationship between perplexity and memory usage. Although feasibility experiments that directly show run failures were not completed in time, the study still provides a step toward making such predictions.

A hybrid approach combining theoretical modeling with empirical analysis was used to estimate memory requirements. The theoretical model was derived from an analysis of openTSNE's algorithmic structure and data flows, particularly the influence of perplexity on nearest-neighbor sparsity and the symmetrization step. This model was further refined by comparison to an analytically derived regression formula trained on a carefully constructed set of small-scale experiments.

The regression model closely matches empirical observations on held-out data, with a mean absolute percentage error of 2.3% on the testing set and within 1% for larger parameters. In addition, it consistently overestimates memory usage, making it a safer choice for conservative runtime decisions. In contrast, the theoretical model, while grounded in algorithm internals, tends to underestimate memory requirements, especially for high-perplexity configurations. This divergence underscores the importance of empirical correction terms (captured in the constant C and the underestimated linear N term), likely stemming from overlooked implementation-level behaviors, such as the overhead introduced by the Annoy index.

Even without feasibility failure data, these results lay a foundation for building practical tools. For example, given a hardware RAM limit and a dataset of known size, the regression model can already be used to predict the maximum perplexity that can be run without encountering slowdowns due to exceeding memory constraints. Such functionality could significantly streamline the t-SNE workflow, especially in constrained computing environments like laptops or VMs.

Moreover, the methodology proposed here: using small-scale analytical runs to train a predictive model, could generalize to other implementations of the t-SNE algorithm. The analytical path allows for safe extrapolation into regions of the parameter space where direct experimentation is computationally prohibitive.

The analytical results also helped strengthen and validate theoretical findings, leading to the discovery of an alternative implementation of a critical section of the code. The alternative successfully brings down the peak memory usage of the algorithm.

7 Conclusions and Future Work

This study presented an investigation into the memory complexity of the t-SNE algorithm as implemented in openTSNE, with a focus on how perplexity influences peak memory usage. By combining theoretical derivation with machine learning-based regression modeling, the work produced accurate, interpretable formulas for memory prediction that align well with empirical measurements.

Even in the absence of direct feasibility test results, the analytical framework proves valuable. It enables a priori estimation of whether a given t-SNE configuration (in terms of N and P) is likely to be computationally feasible on a given machine. This allows researchers to avoid failed runs due to memory overloads and makes hyperparameter tuning more efficient.

Future work can extend this approach by exploring the effect swap usage has on the various sections of the algorithm, and validating the predictions against actual out-of-memory failures in a broader range of hardware environments. This would confirm whether the conservative estimates provided by the regression model hold in practice. Additionally, investigating other hyperparameters such as learning rate, exaggeration, and different affinity metrics, and exploring their memory impact could further enhance the accuracy of the model.

Ultimately, this research contributes towards a fully automated tool for perplexity selection under memory constraints, improving the accessibility and reliability of t-SNE as a visualization tool in real-world settings.

References

[1] Anna C Belkina, Christopher O Ciccolella, Rina Anno, Richard Halpert, Josef Spidlen, and Jennifer E Snyder-Cappione. Automated optimized parameters for t-distributed stochastic neighbor embedding improve visualization and analysis of large datasets. *Nat. Commun.*, 10(1):5415, November 2019.

[2] George C. Linderman, Manas Rachh, Jeremy G. Hoskins, Stefan Steinerberger, and Yuval Kluger. Fast interpolation-based t-sne for improved visualization of single-cell rna-seq data. *Nature Methods*, 16(3):243–245, February 2019.

[3] Martin Skrodzki, Nicolas F. Chaves de Plaza, Thomas Höllt, Elmar Eisemann, and Klaus Hildebrandt. Navigating perplexity: A linear relationship with the data set size in t-sne embeddings, 2024.

[4] Laurens van der Maaten. Barnes-hut-sne, 2013.

[5] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[6] Theodoros P Zanos, Harold A Silverman, Todd Levy, Tea Tsaava, Emily Battinelli, Peter W Lorraine, Jeffrey M Ashe, Sangeeta S Chavan, Kevin J Tracey, and Chad E Bouton. Identification of cytokine-specific sensory neural signals by decoding murine vagus nerve activity. *Proc. Natl. Acad. Sci. U. S. A.*, 115(21):E4843–E4852, May 2018.