Accelerating T-distributed Stochastic Neighbouring Embedding

Hongyi Liu
UM-SJTU Joint Institute
Shanghai Jiao Tong University
Shanghai, China
henryliu00@sjtu.edu.cn

Abstract

T-distributed Stochastic Neighbouring Embedding(t-SNE) is an effective dimensional reduction method for high dimensional data visualization. It is defined as an constraint optimization problem minimizing the Kullback-Leibler divergence between projected distribution and the original distribution of the data. This project will explore t-SNE's optimization procedure and large scaled t-SNE using various optimization algorithms. Alternative acceleration methods for naive gradient descent and block coordinate descent with multi-thread parallel computing will be used for small datasets and large dataset respectively. We will show that our method can accelerate the optimization procedure and reduce the computational time greatly on a personal computer.

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

1 Introduction

1.1 Background

Dimensional reduction has long been a popular topic for data mining and machine learning. Representing high-dimensional data in low dimensional space such as 2D or 3D space can help visualize the structure of data and help human observer to interpret information especially when the data are on a manifold. This paper will first give a short review about a series of dimensional reduction methods from the Stochastic Neighbor Embedding(SNE)[1] to its advanced version of t-distributed Stochastic Neighbor Embedding(t-SNE)[2] and analyse its computational cost. After the analyse, the paper will discuss existing solution to large scaled t-SNE. An implementation of the algorithms will also be presented in this project along with the report.

The SNE[1] belongs to a family of dimensional reduction algorithm that preserves locally structure. These include the Locally Linear Embedding and the Locality Preserving Projections. They focus on the preserving local structures such that similar data are still similar in the low dimensional space. This similarity is captured by a conditional Gaussian distribution. By defining a cost function that penalize the dissimilarity of closely located data between original space and projected space, we solve the optimization to learn the projection.

The naive SNE comes with a serious problem in practical use: the Crowding problem. Since the SNE focuses on perserving local structure, points that are distant from each other turned out to be mapped into similar places that overcrowd the lower dimensional space. T-SNE[2] moderate this problem by modify the cost function using a t-distribution that is heavy tailed and balance the penalization between local structure and global structure. T-SNE is now widely used in data visualization and dimensional reduction for its transferability on various datasets.

1.2 Target

T-SNE and its variants suffers from heavy computational burden on large dataset. Naive method has a computation cost of $O(n^2)$ where n is the size of the dataset. This makes the training time unendurable when data scales to size of millions. In this project we will explore the optimization process during t-SNE. We will first try alternative acceleration methods for naive gradient descent and discuss its efficiency. We will also show the block coordinate descent with multi-thread parallel computing can speed up the optimization procedure for moderately large datasets.

2 Problem Definition

2.1 T-SNE

T-SNE is defined as an constraint optimization problem minimizing the Kullback-Leibler divergence between projected distribution and the original distribution of the data, as we describe it according to Hinton et al. as below[2]. Consider the original data in a high dimensional space $\{X_i\} \in \mathbb{R}^{s_x}$ with n data points. We want to project $\{X_i\}$ in low dimensional space as $\{Y_i\} \in \mathbb{R}^{s_y}$ where for visualization tasks s_y is generally set to 2 or 3. We define the joint distribution associated with $\{X_i\}$ and $\{Y_i\}$ as p_{ij} and q_{ij} respectively. P_{ij} is defined as the following:

$$p_{j|i} = \frac{\exp(-||X_i - X_j||^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-||X_i - X_k||^2/2\sigma_i^2)} \tag{1}$$

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

We notice $p_{j|i}$ is a condition distribution defined at center X_i with distance given by Gaussian kernel associated with parameter σ_i. The joint distribution p_{ij} is define to ensure the symmetric property $p_{ij} = p_{ji}$, where $1/2n$ is the normalization term that gives $\sum_{i\neq j} p_{ij}$.

The projected joint distribution q_{ij} is defined by

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k\neq l}(1 + ||y_k - y_l||^2)^{-1}} \tag{3}$$

This is known as a Cauchy distribution or the Student t-distribution with degree one. The reason that Cauchy distribution gives better performance than Gaussian distribution is that Cauchy distribution is a heavy tailed distribution that moderate the crowding issue of naive SNE . Since we focus mainly on the optimization procedure, we will not go into the detail of machine learning advantages of t-SNE in our project.

2.2 Optimization of t-SNE

The optimization of t-SNE is defined as:

$$\min_{Y} \quad f(Y) = KL(P|Q) = \sum_{i} \sum_{j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{4}$$

using the definition of q_{ij} and p_{ij} stated above. This is often done by gradient descent and its variant. In this paper, we will test the optimization efficiency using different kinds of gradient descent algorithms. The gradient of $f(Y)$ is given below:

$$\frac{df}{dY} = 4 \sum_{j} (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||) \tag{5}$$

Besides all the hyper-parameter during the optimization, σ_i is of high importance. It controls the sparsity of the conditional Gaussian distribution and we need to vary its value to model dense or sparse regions accordingly. This is done by defining the perplexity associated with the Shannon entropy as below:

$$H(p_{j|i}) = -\sum_{j} p_{j|i} \log p_{j|i} \tag{6}$$

$$Perp(p_{j|i}) = \exp(H(p_{j|i})) \tag{7}$$

In our experiment we use binary search to set the perplexity around 30 for all points.

The early exaggeration is a trick to form natural clusters in t-SNE optimization. The p_{ij} is multiplied by a factor in the early stage, which emphasize on learning large probabilities that occurs mainly between natural clusters and form better cluster patterns in the projected space. We use early exaggeration to improve the visualization.

3 Method

To evaluate the effect of different optimization algorithm for t-SNE we test our result using three different gradient descent method: (1) vanilla gradient descent(GD) (2) Polyek momentum algorithm as originally proposed in the paper[2] and (3) Nesterov acceleration. To test the transferability of t-SNE on two different data sets: (1) an artificial dataset on a Swiss Roll manifold in three dimensional space and (2) the MNIST data set. The Swiss Roll dataset is illustrated in figure 1. The MNIST data set is preprocessed with principal component analysis(PCA)[3] to reduce the original 28^2 dimensional data into 50 to speed up the experiment.

We list the formula of our three optimization algorithms using following notations:$Y \in \mathbb{R}^n$ as the optimization variable, t as the iteration steps, α as the momentum, η as the learning rate. For the GD we have:

$$Y_{k+1} = Y_k - \eta \frac{df}{dY} \tag{8}$$

For the Polyek momentum algorithm we have:

$$Y_{k+1} = Y_k - \eta \frac{df}{dY} + \alpha(Y_k - Y_{k-1}) \tag{9}$$

For the Nesterov acceleration we have:

$$Z_{k+1} = Y_k - \eta \frac{df}{dY} \tag{10}$$

$$Y_{k+1} = Z_{k+1} + \alpha(Z_{k+1} - Z_k) \tag{11}$$

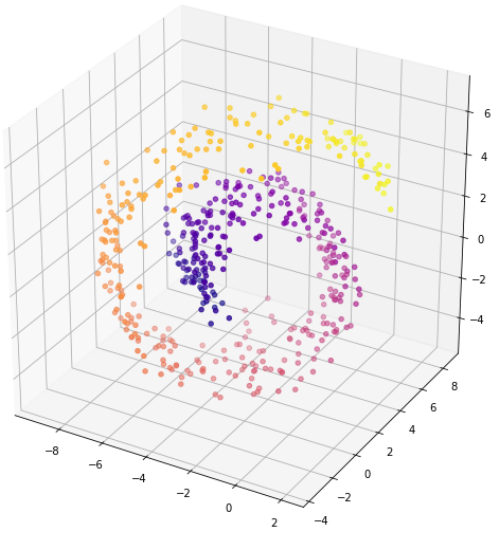


Figure 1: Artificial data on a Swiss Roll manifold in 3D space

We will compare the efficiency of these algorithms in the following section. During the optimization, we use early exaggeration in the early stage of optimization to get a better cluster patterns in the projected space. In our experiment we set $\eta = 100$ empirically, and $\alpha = 0.5$, early exaggeration magnification in to 4 and record the training progress with respect to time.

Next we test t-SNE on large scale data set using block coordinate descent. For t-SNE on 2D space for a data set of size n, we have $2n$ optimization parameters. For n equals to ten thousands we need several hours to finish reach convergence. The exact optimization method for t-SNE requires both time and memory space up to a complexity of $O(n^2)$. By using block coordinate descent, we introduce a scaling factor s, and update only n/s as batch size for each update. To update all the n optimization parameters for one time, the computational cost will scale to approximately $\frac{1}{s} = \frac{\text{batch size}}{n}$ as we use block coordinate descent. Due to the limitation of computational resources, we use a moderate large data size as $n = 10000$ to do our experiment. We will show that block coordinate descent greatly improves the time efficiency. The block coordinate descent is done such that in each iteration the full data points are separated to several batches and during one iteration all the optimization parameters are updated for one time. The batch size of block coordinate descent is set to 2000, 1000, 500 and 10000 as the baseline full batch gradient descent respectively. We will use the Polyek gradient descent since it outperformed other two algorithms in earlier experiments described below. Except for the batch size, all other hyper-parameters are set to be identical.

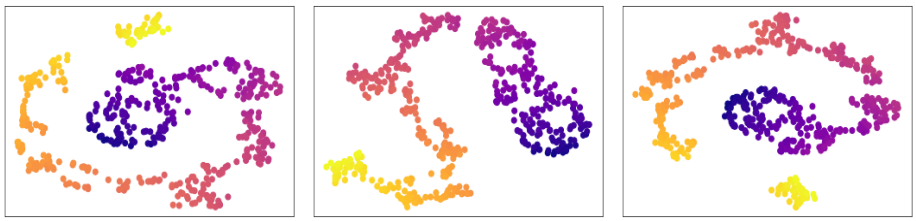
The block coordinate descent scheme mentioned above can fit in parallel computing directly. For every stage we separate the total data into s blocks , open s threads and use multi-threading to update the data for each block. We wait for all s threads to finish before we start the next iteration of update to synchronize the optimization procedure at each step. We do our experiment on a personal laptop with a 4 cores 8 threads CPU. Other settings are set to be identical with previous experiment.

4 Experiment Result

4.1 Swiss Roll

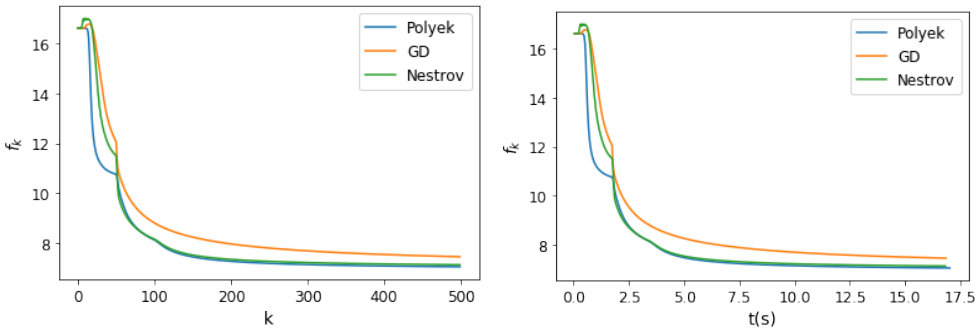
The projected result of the Swiss Roll data set is shown in figure 2. We see all three algorithms successfully projected the roll into a band structure in the two dimensional space, however the unroll direction is slightly different.

The Polyek method gives the best result with an unbroken band while the other two gives a broken cluster at the yellow end of the roll. We further notice that three algorithms give a similar cost



(a) GD (b) Polyek (c) Nesterov

Figure 2: T-SNE projected result of artificial Swiss Roll data



(a) $f_k vs. k$ (b) $f_k vs. t$

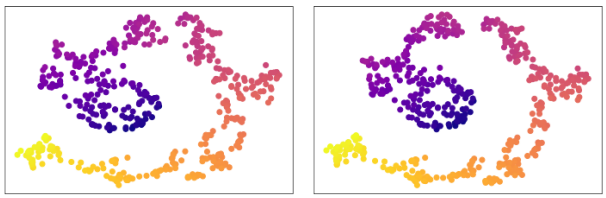
Figure 3: Time efficiency comparison for different optimization algorithms. (a) Loss function value respect to iterations. (b) Loss function value respect to Time.

function value at the end of the optimization procedure as shown in figure 3. Time costs for all three algorithms are almost the same as expected. The turning point at 50 epoch is due to early exaggeration used in the initial phase. We see that in the first phase Polyek has a better optimization speed than the Nesterov while the GD is the slowest. In the second phase the Nesterov quickly catch up with the Polyek while GD remains the slowest. We conclude that the early exaggeration turning point progress contribute the the final band of forming cluster patterns when comparing figure 3 to figure 2 and notice that only the Polyek algorithm appears to converge at iteration 50.

If we want to form better cluster patterns we should adjust the early exaggeration hyper-parameters to get closer to convergence at the early stage. We support this conclusion by setting early exaggeration to 100 iterations and plot the projected result of GD and Nesterov method in figure 4. We notice this alleviated the broken band problem comparing to 2.

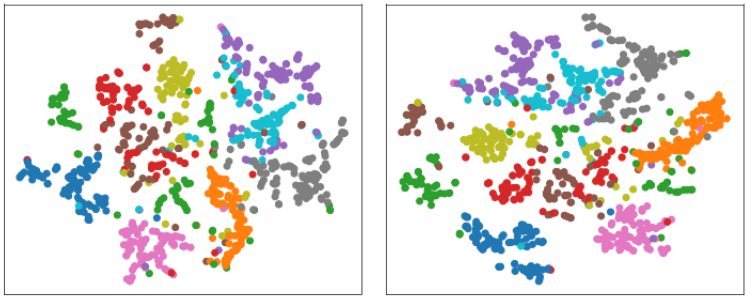
4.2 MNIST

Second experiment is done on the MNIST data set. We first use principle component analysis to reduce the dimension to 50, which approximately explains 60 percent of the total variance. We take

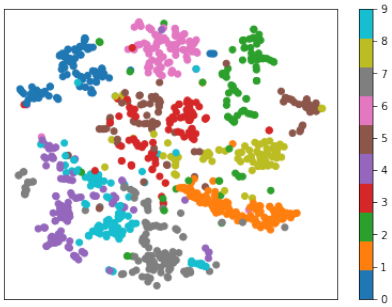


(a) GD (b) Nesterov

Figure 4: T-SNE projected result of artificial Swiss Roll data with longer early exaggeration phase

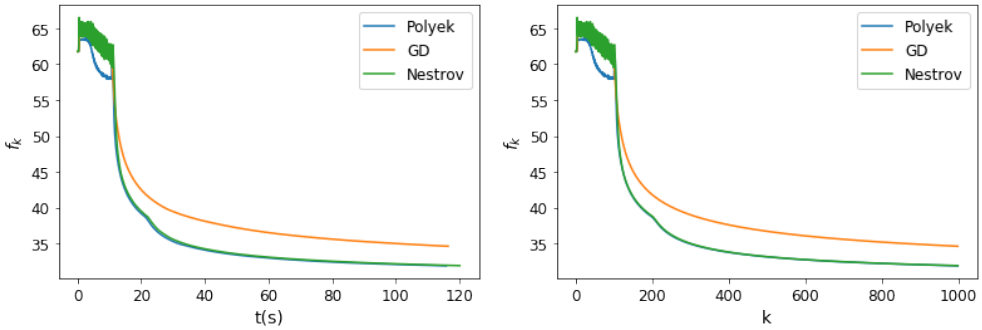


(a) GD (b) Polyek



(c) Nesterov

Figure 5: T-SNE projected result on MNIST data set with 1000 points



(a) $f_k vs. k$ (b) $f_k vs. t$

Figure 6: Time efficiency comparison for different optimization algorithms on MNIST data set with 1000 points. (a) Loss function value respect to iterations. (b) Loss function value respect to Time.

batch size	time spent after 1000 iterations(s)	KL-divergence after 1000 iterations
10000	7809	3.71
2000	2719	2.80
1000	1632	2.76
500	1191	3.95

Table 1: The time spent and KL-divergence on MNIST data set with 10000 points using different batch size for block coordinate descent after 1000 iterations

only 1000 samples from the set as an toy example. The result in figure 5 shows that all three methods gives fairly similar result. Different labels separated to form clusters although some clusters are twisted with each other.

The training progress plot shows a similar result as in the Swiss Roll case: a two phase pattern where Polyek converge fastest in the first phase and the cost of Polyek and Nesterov decays equally fast in the second phase. We conclude that from our experiment the Polyek momentum algorithm has the best performance for these two t-SNE experiments.

4.3 Large Scale MNIST

4.3.1 Block Coordinate Descent

We scale our experiments to large scale MNIST data set with 10000 data points. As shown in table 1, the original full batch gradient descent requires around two hours. As we gradually decrease the batch size, we see the time for 1000 full iterations gradually decrease and scales approximately $\frac{n}{\text{batch size}}$ of the full gradient descent computational cost as expected. The best performance is reached when batch size is 1000.

This can also be seen from the projected result in figure 7. When batch size equals to 500, a large proportion of points are scattered around the space although some of them formed cluster patterns. When batch size gradually increases to 2000 the points are clustered better with less scattered points. However when we use full batch gradient descent the clusters begin to twisted with each other and the border become unrecognizable. We explain this noticing the behaviour of optimization procedure during the early exaggeration phase.

As can see from figure 8(a), three of the block coordinate descent method has an increase in the loss function in the early exaggeration procedure as high variance are introduced during the block coordinate descent procedure. When batch size decreases, the variance increases and loss function values are raised higher. However we see during the early exaggeration phase (100 iterations) to the first 400 iterations the loss almost stayed the same for full batch method. We surmise that the random initial value for a large scale data set may be a local well such that some variance is needed to reach the outside of the local well to gain better performance.

From figure 8 (b) we see the optimization procedure with respect to time for the first 20 minutes. Although smaller batch size gives more updates in a unit time, batch size of 500 is outperformed by other two larger batch size for it introduced too much variance in the early exaggeration phase such that the optimization starting points has a quite large loss value that slows the whole procedure down. Comparing the time efficiency when batch size equals to 1000 and 2000 respectively, we see there is a trade off between computational cost and performance. We conclude that the best configuration in this experiment is when batch size equals to 2000.

4.3.2 Multi-threading Paralleled Block Coordinate Descent

For multi-threading paralleled block coordinate descent, we mainly focus on the time efficiency since the t-SNE result is almost identical to figure 7. In figure 9 we see that when we set thread number to be larger that 5, we can achieve the best convergence speed in the early stage. However when thread number are larger or equal to 5, the convergence speed in three cases are almost the same, and they only diverge in later stage of the optimization procedure. This is because in early stage the dominating factor of convergence speed is how many effective updates we can do in equal time. Since the experiment is done on a 4 core 8 threads computer, increasing threads number will not make optimizations faster even in ideal case. In later stage, smaller threads(scaling factor) has gives

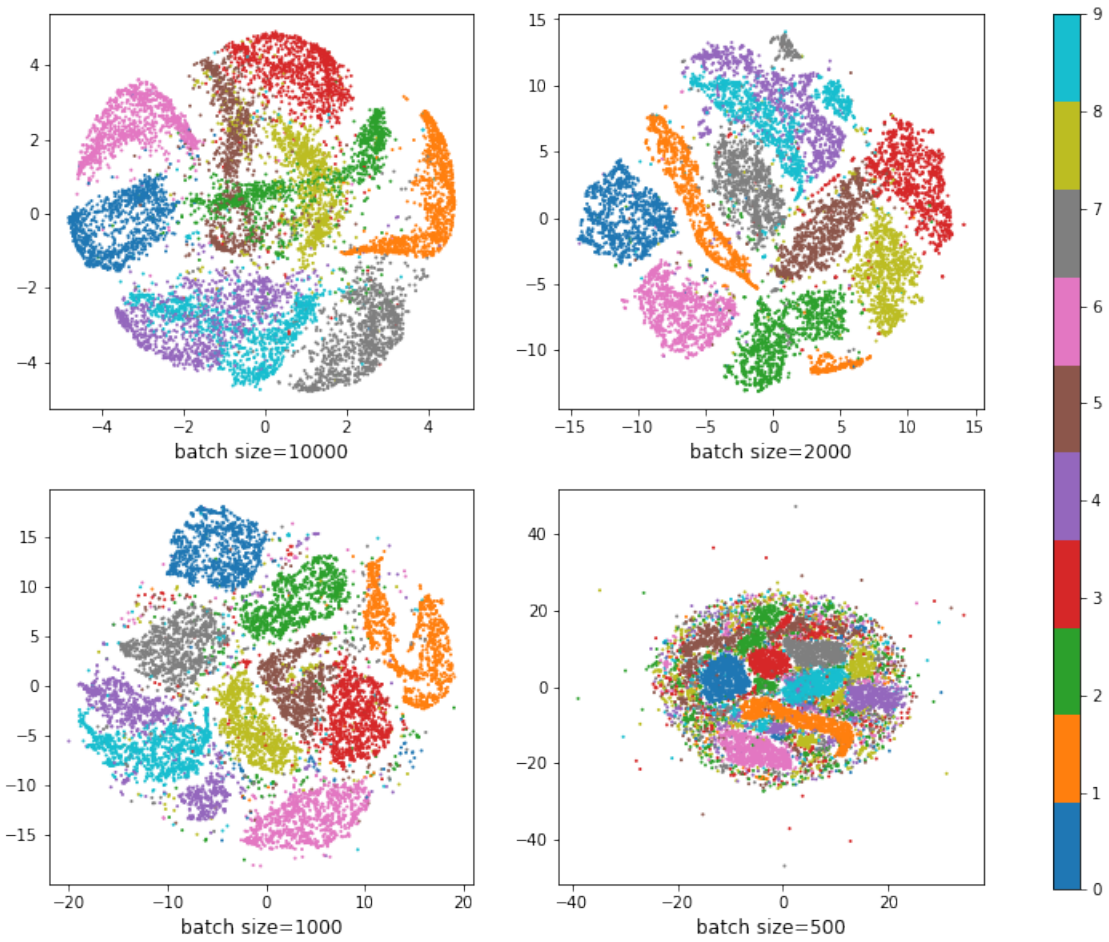
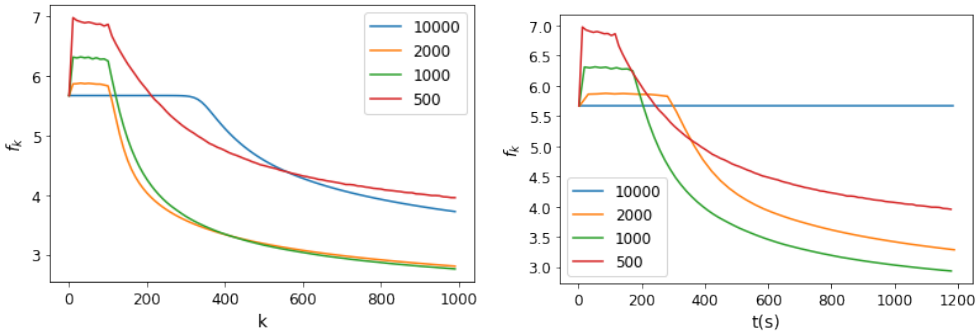


Figure 7: T-SNE projected result on MNIST data set with 10000 points using different batch size for block coordinate descent after 1000 iterations



(a) $f_k vs. k$ (b) $f_k vs. t$

Figure 8: Time efficiency comparison on MNIST data set with 10000 points using different batch size for block coordinate descent. (a) Loss function value respect to iterations. Each k iteration means all the coordinates are updated for k times. (b) Loss function value respect to time. Only first 20 minutes are plotted, which correspond to the total running time when batch size equals to 500.

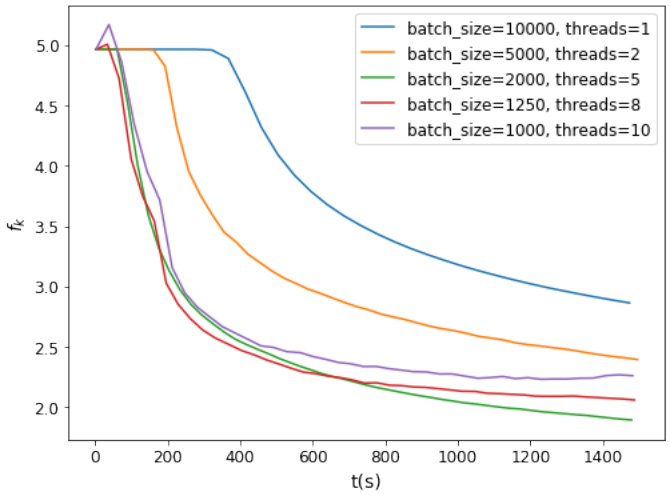


Figure 9: Time efficiency of loss function v.s. time on MNIST data set with 10000 points using different batch size and threads number for multi-threading paralleled block coordinate descent

better result with smaller cost. This is because smaller scaling factor result in less variance during optimization and gives better result.

5 Conclusion & Discussion

T-SNE is an effective dimensional reduction method for high dimensional data visualization. However it suffers from its heavy computational costs for large scale data. In our experiment we show that Polyek acceleration gives the best performance among all the acceleration of naive gradient descents. Since t-SNE turns a large scale data into a high dimensional optimization problem, it is suitable to use block coordinate descent to accelerate. Our experiment shows that the block coordinate descent can greatly reduce the computational time for moderately large data size. It can also be further accelerated by multi-threading paralleled computing.

This project has discussed different algorithms that can accelerate t-SNE optimization procedure, however non of these algorithms can reduce the computational complexity. Although in practical they reduce the computational time to an acceptable level, they still scales $O(n^2)$ when the computational cost increases. Further studies are needed to modify the optimization procedure if we wish to also decrease the complexity.

References

[1] G. E. Hinton and S. T. Roweis, "Stochastic neighbor embedding," in *Advances in neural information processing systems*, 2003, pp. 857–864.

[2] 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.

[3] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

A Appendix: Codes Documentation

The source codes in Swiss Roll, MNIST, large scale MNIST are programmed in Python. The source code for multi-threading paralleled block coordinate descent are programmed in C++ since Python does not support multi-threading programming on computational intensive algorithms. The python source code has only one file named "tSNE.py" that runs Swiss roll and MNIST experiment in its main function. The C++ implementation contains the header "tsne.h", source file "tsne.cpp" and main source file "main.cpp". The complied file "main" would run multi-threading block coordinate descent on linux system. Four related data set are also contained.