Simple, Practical, and Fast Dynamic Truncation Kernel Multiplication

Lianke Qin(UCSB), Somdeb Sarkhel(Adobe), Zhao Song(Adobe), Danyang Zhuo(Duke)

Abstract

Computing the product of a kernel matrix and a vector is the most basic and important operation in high-performance machine learning and scientific computing. The speed for this calculation determines plays a critical role in the overall performance of machine learning training and inference. As dataset sizes rapidly increase, the dimension of the kernel matrix also increase accordingly, and this product computation is increasingly a performance bottleneck. In the meantime, our observation is that many popular kernel matrices are inherently sparse, due to natural data distributions. In this paper, we design an efficient data structure to approximate kernel matrix vector multiplication. Our data structure is a search tree which enables us to quickly extract those entries and calculate the multiplication results.

Motivation

Kernel method is an important class of machine learning techniques. It is widely used in classification, deep neural networks, and computer vision. When using the kernel method, we often need to compute the product of a kernel matrix and a vector. The speed for this computation determines the overall performance of the higher level machine learning and scientific computing tasks.

The dimension of the kernel matrix is the same as the size of the dataset. Modern dataset has increasing numbers of samples. This makes the calculation of a kernel matrix and a vector increasingly slow. In the meantime, our observation is many data distributions naturally lead to sparse kernel matrices. For example, when data are clustered around several locations, the entry in the kernel matrix is non-negligible if only if the corresponding data points are in the same cluster. This raises an important question:

Can we speed up the kernel matrix vector multiplication time for sparse kernel matrices?

Technique

Our approach uses an efficient search tree data structure design. The search tree enables us to quickly locate non-negligible entries in a large kernel matrix. During the multiplication, we simply omit negligible entries to speed up the matrix vector computation. This approach enables us to compute the kernel vector multiplication efficiently when the kernel entries are highly sparse. This truncated matrix vector multiplication is motivated by activation functions like ReLU which sets the output as zero if the corresponding input values do not surpass certain threshold.

We define the truncated matrix vector multiplication in Definition 1.

Definition 1 (Truncated Matrix Vector Multiplication) *Given $x_1, \cdots, x_n \subset \mathbb{R}^d$ and $y_1, \cdots, y_m \subset \mathbb{R}^d$. We define matrix $\mathsf{K} \in \mathbb{R}^{n \times m}$ as follows:*

$$\mathsf{K}_{i,j} := f(x_i, y_j).$$

For any query vector $v \in \mathbb{R}^m$, a truncation threshold τ and an index $i \in [m]$, the goal is to compute

$$\sum_{j=1}^{m} v_j \cdot \mathsf{K}_{i,j} \cdot \mathbf{1}_{\langle x_i, y_j \rangle \geq \tau}, \forall i \in [n]$$

As shown in Figure 1, we use a max binary search tree to preprocess all the entries of kernel matrix K based on the inner product $\langle x_i, y_j \rangle$. Given a vector $v \in \mathbb{R}^m$, a truncation threshold τ an index $i \in [m]$, the data structure can identify the indexes of kernel entries which surpass the trucation threshold in logarithmic time via a parallel search from tree root down to the leaves.

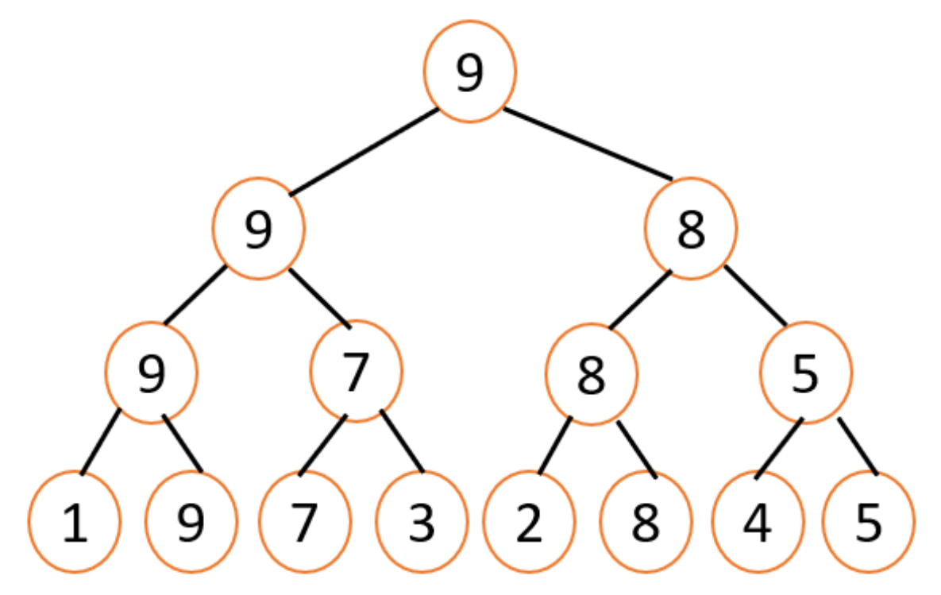

Fig. 1: Max binary tree search structure example

Our Results

We present our main theorem as follows:

Theorem 2 (Main Theorem) *Assume the time complexity of evaluating $f(x, y)$ is $\mathcal{T}_f$. There exists a data structure which uses $O(mn + d(m + n))$ spaces and supports the following operations:*

- Init$(y_1, y_2, \cdots, y_m \in \mathbb{R}^d, x_1, x_2, \cdots, x_n \in \mathbb{R}^d)$. *Given $y_1, y_2, \cdots, y_m \in \mathbb{R}^d$ and n data points $x_1, x_2, \cdots, x_n \in \mathbb{R}^d$, the time complexity of* Init *operation is $O(mn(d + \mathcal{T}_f))$.*
- Update$(z \in \mathbb{R}^d, j \in [m])$. *Given $z \in \mathbb{R}^d$ and an index $j \in [m]$, the* Update *operation runs in $O(n(d + \log(m) + \mathcal{T}_f))$ time.*
- Query$(i \in [n], \tau \in \mathbb{R})$. *Given an index $i \in [n]$ and a threshold $\tau \in \mathbb{R}$ as input and let K_i denote the number of entries of above τ in tree T_i, the* Query *operation runs in $O(K_i \log(m))$ time and output a set containing all y_j such that $\langle x_i, y_j \rangle \geq \tau$ in tree T_i.*
- Multiply$(v \in \mathbb{R}^m, \tau \in \mathbb{R})$, *Given a vector $v \in \mathbb{R}^d$ and a threshold $\tau \in \mathbb{R}$ as input, and let K_i denote the number of entries of above τ in each tree T_i,* Multiply *outputs the result of truncated matrix vector multiplication $\mathsf{K} \cdot v$ in $O(\sum_{i=1}^{n} K_i \log(m))$ time.*

Evaluation

We evaluate our algorithm with $n = 2048$, $d = 64$ and $m = 2048$ randomly generated data points $\{x_i\}_{i=1}^n$ and $\{y_j\}_{j=1}^m$, where $\{x_i\}_{i=1}^n$ are generated by 32 random clusters and each cluster contains 64 data points.

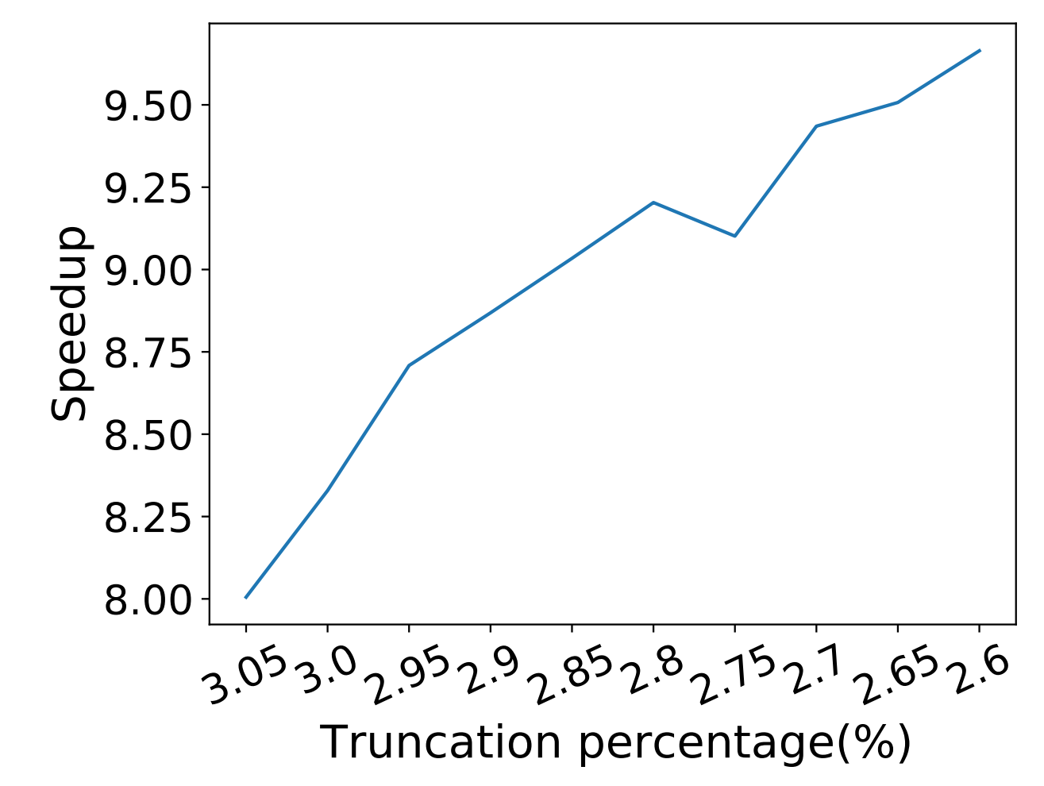

Fig. 2: Speedup under different truncation percentage.

We find that Multiply achieves $8.33\times$ speedup compared with truncating each kernel entry with threshold sequentially. This speedup comes from efficiently locating the kernel indexes in logarithmic time for truncated kernel multiplication.

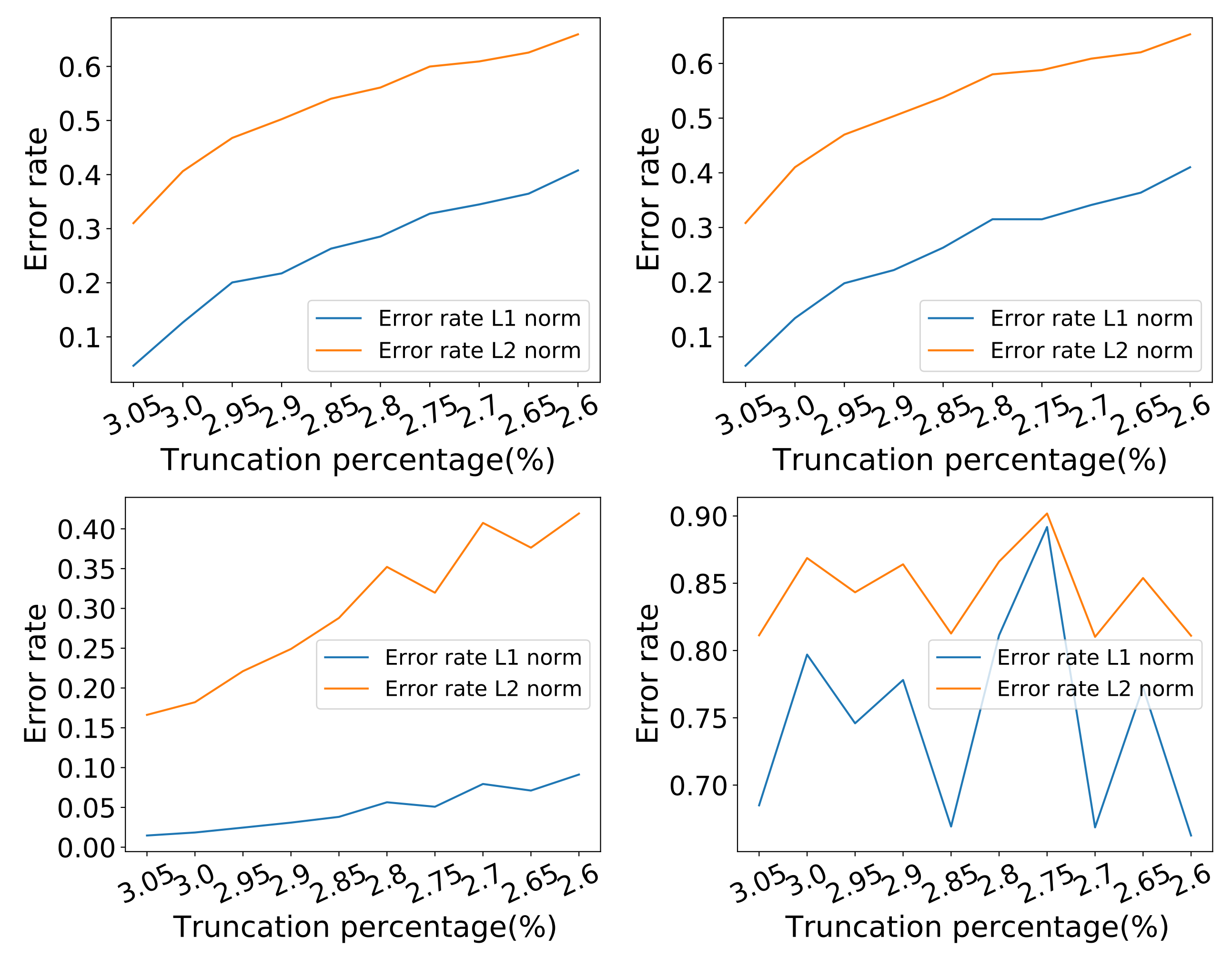

Fig. 3: L1 and L2 norm error rates under different truncation degrees. (1) Gaussian. (2) TStudent. (3) Polynomial. (4) Tanh.

For Gaussian, T-student and kernels, our Multiply can achieve $< 10\%$ L1 norm error rate and $< 40\%$ L2 norm error rate when the truncation percentage is larger than 3.0%. For Polynomial kernel, our Multiply can achieve $< 2\%$ L1 norm error rate and $< 20\%$ L2 norm error rate when the truncation percentage is larger than 3.0%. When the truncation percentage drops below 3.0%, the L1 and L2 error rates increase sharply as the truncation percentage decreases. This is because Multiply leverages less kernel entries to compute the multiplication which yields to lower accuracy.

For Tanh kernel, our Multiply yields to bad multiplication accuracy compared to the non-truncated matrix-vector multiplication.