Journal of Machine Learning Research 1 (2018) 1-5 Submitted 12/17; Published xx/xx

ThunderSVM: A Fast SVM Library on GPUs and CPUs

Zeyi Wen[†] WENZY@COMP.NUS.EDU.SG
Jiashuai Shi[†‡] SHIJIASHUAI@GMAIL.COM
Bingsheng He[†] HEBS@COMP.NUS.EDU.SG
Qinbin Li[†] T0915610@U.NUS.EDU
Jian Chen[‡] ELLACHEN@SCUT.EDU.CN
[†] *School of Computing, National University of Singapore*
[‡] *School of Software Engineering, South China University of Technology*

Editor: Kevin Murphy and Bernhard Schölkopf

Abstract

Support Vector Machines (SVMs) are classic supervised learning models for classification, regression and distribution estimation. A survey conducted by Kaggle in 2017 shows that 26% of the data mining and machine learning practitioners are users of SVMs. However, SVM training and prediction are very expensive computationally for large and complex problems. This paper presents an efficient and open source SVM software toolkit called *ThunderSVM* which exploits the high-performance of Graphics Processing Units (GPUs) and multi-core CPUs. ThunderSVM supports all the functionalities—including classification (SVC), regression (SVR) and one-class SVMs—of LibSVM and uses identical command line options, such that existing LibSVM users can easily apply our toolkit. ThunderSVM can be used through multiple language interfaces including C/C++, Python, R and MATLAB. Our experimental results show that ThunderSVM is generally an order of magnitude faster than LibSVM while producing identical SVMs. In addition to the high efficiency, we design our convex optimization solver in a general way such that SVC, SVR, and one-class SVMs share the same solver for the ease of maintenance. Documentation, examples, and more about ThunderSVM are available at `https://github.com/zeyiwen/thundersvm`.

Keywords: SVMs, GPUs, multi-core CPUs, efficiency, multiple interfaces

1. Introduction

Support Vector Machines (SVMs) have been widely used in many applications including document classification (DOrazio et al., 2014), image classification (Pasolli et al., 2014), blood pressure estimation (Kachuee et al., 2015), disease detection (Bodnar and Salathé, 2013), and outlier detection (Roth, 2006). A survey conducted by Kaggle in 2017 shows that 26% of the data science practitioners use SVMs to solve their problems (Thomas, 2017). The open-source project LibSVM which supports classification (SVC), regression (SVR) and one-class SVMs has been widely used in many applications. LibSVM was developed in 2000 (Chang and Lin, 2011), and is maintained since then. Despite the advantages of SVMs, SVM training and prediction are very expensive for large and complex problems.

Graphics Processing Units (GPUs) have been used to accelerate the solutions of many real-world applications (Dittamo and Cisternino, 2008), due to the abundant computing

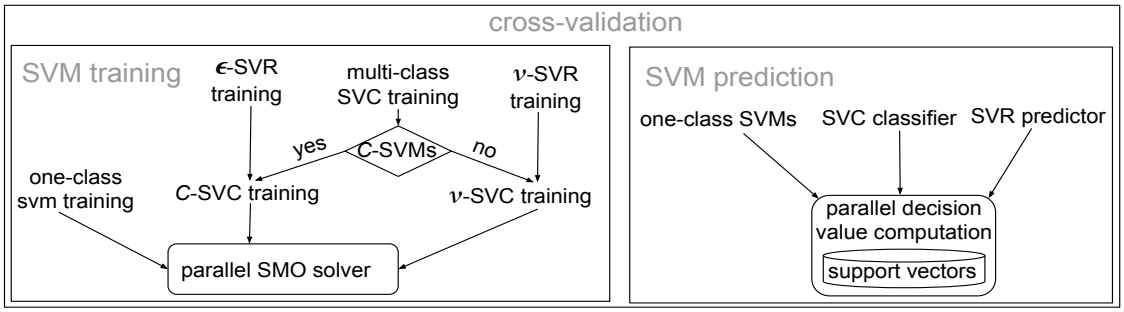

Figure 1: Overview of training and prediction

cores and high memory bandwidth of GPUs. In this paper, we introduce a toolkit named *ThunderSVM* which exploits GPUs and multi-core CPUs. The mission of our toolkit is to help users easily and efficiently apply SVMs to solve problems. ThunderSVM aims to reduce the time of practitioners spent on training SVMs and prediction using SVMs. ThunderSVM supports all the functionalities of LibSVM including SVC, SVR and one-class SVMs. It is worthy to point out that one way to train SVMs faster is to use kernel approximations. ThunderSVM aims to find an exact solution whereas using kernel approximations leads to an approximate solution. To lift the barrier of using ThunderSVM, we use the same command line input options as LibSVM, such that existing LibSVM users are able to easily switch to ThunderSVM. Moreover, ThunderSVM supports multiple interfaces such as C/C++, Python, R and MATLAB. ThunderSVM can run on machines running Linux, Windows or Macintosh operating systems with or without GPUs. We provide a detailed user guide which includes instructions for execution environment setup and examples of using ThunderSVM. To allow contributors to easily engage in ThunderSVM, we provide a detailed API description. Our experimental results show that ThunderSVM is generally a factor of 10x faster than LibSVM in all the functionalities. The full version of ThunderSVM, which is released under Apache License 2.0, can be found on GitHub at `https://github.com/zeyiwen/thundersvm`. ThunderSVM has attracted 656 stars and 67 forks at GitHub as of April 15, 2018.

2. Overview and Design of ThunderSVM

Like LibSVM, ThunderSVM supports one-class SVMs, C-SVMs and ν-SVMs where C represents the regularization constant and ν represents the parameter controlling the training error. Both C-SVMs and ν-SVMs are used for classification and regression.

For ease of maintenance, we develop ThunderSVM in a modular manner. Figure 1 shows the overview of ThunderSVM which has many functionalities: one-class SVMs for distribution estimation, C-SVC and ν-SVC for SVM classification, and ϵ-SVR and ν-SVR for SVM regression. The training algorithms for those SVMs are built on top of a generic parallel SMO solver which is for solving quadratic optimization problems. Notably, the SVM training for regression (such as ϵ-SVR and ν-SVR) and the multi-class SVM training can be converted into the training of an SVM classifier. The prediction module is relatively simple, because the prediction is the same for one-class SVMs, C-SVMs and ν-SVMs. The

prediction is essentially computing predicted values based on support vectors. ThunderSVM also contains the cross-validation functionality which consists of training and prediction.

Although GPUs have been used for machine learning for a while, the design and implementation of an efficient GPU program is still a non-trivial task. We have carefully developed SVM training and prediction with algorithmic parallelization and memory efficiency optimizations. In SVM training and prediction, kernel value computation consumes significant amount of time (especially for data sets with a large number of training instances), because it requires accesses to the GPU global memory of high latency and performing massive computation. Next, we present key details of the parallel training and prediction.

2.1 Design of Parallel SVM Training Algorithms

We have developed a series of optimizations for the training. First, ThunderSVM computes a number of rows of the kernel matrix in a batch, reuses the rows that are stored in the GPU memory buffer, and solves multiple subproblems in that batch. Thus, ThunderSVM avoids performing a large number of small read/write operations to the high latency memory and reduces repeated kernel value computation. Moreover, we apply GPU shared memory to accelerate parallel reduction, and use the massive parallelism to update elements of arrays.

More specifically, for solving each subproblem in the SVM training, we use the SMO algorithm which consists of three key steps. Step (i): Find two extreme training instances which can potentially improve the currently trained SVM the most. Step (ii): Improve the two Lagrange multipliers of the two training instances. Step (iii): Update the optimality indicators of all the training instances. We parallelize Step (i) and (iii). Step (ii) is computationally inexpensive, and we simply execute it sequentially. In Step (i), our key idea is to apply the parallel reduction (Merrill, 2015) twice for finding the two extreme training instances. In the parallel reduction, we first load the whole array from the GPU global memory to shared memory in a coalescent way, and then reduce the array size by two at each iteration until only one element left. In Step (iii), we dedicate one thread to update one optimality indicator to use the massive parallelism mechanism of the GPU.

For solving the batch of subproblems, we propose techniques to exploit the properties of the batch of subproblems. First, to reduce access to the high latency memory, the kernel values needed for the batch are organized together and computed through matrix multiplication. As a result, we reduce a large number of small read/write operations to the high latency memory during kernel value computation. We use matrix operations from the high-performance library cuSparse (Nvidia, 2008) provided by NVIDIA. Second, to reduce repeated kernel value computation, we store the kernel values in a GPU buffer for efficient reuse during the optimization for the batch. Regarding the selection of the batch, we make use of the set of instances with the deepest gradients.

Training SVMs for regression (SVR) or for multi-class classification can be reduced to training SVMs for binary classification (SVC), as discussed in the previous study (Shevade et al., 2000). Two SVC training algorithms (C-SVC and ν-SVC) and training algorithm for one-class SVMs are essentially solving optimization problems using SMO. More details about the relationship of SVR and SVC, and SMO can be found in the Appendix. The key task in the SVM training is to parallelize SMO, and the insight has been discussed above. The parallelism principles are applicable to CPUs.

data set			elapsed time (sec)			speedup	
name	cardinality	dimension	ThunderSVM		LibSVM		
			gpu	cpu		gpu	cpu
mnist8m (svc)	8.1x10^6	784	7.1x10^3	3.2x10^4	8.1x10^5	114	39.9
e2006-tfidf (svr)	16087	150360	13.25	343.5	9161	691	25.3
webdata (osvm)	49749	300	4.66	16.5	1493	320	90.5

Table 1: Comparison between ThunderSVM with LibSVM

2.2 Design of the Parallel Prediction Algorithm

Although ThunderSVM supports several algorithms (such as one-class SVMs, classification and regression), their underlying prediction algorithm is identical: a function based on the support vectors and their Lagrange multipliers. The function is $v = \sum_{i=1}^{n} y_i \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_j) + b$ where $\boldsymbol{x}_j$ is the instance of interest for prediction; y_i and α_i are the label and Lagrange multiplier of the support vector $\boldsymbol{x}_i$, respectively; b is the bias of the SVM hyperplane; $K(\cdot, \cdot)$ is the kernel function. In ThunderSVM, we perform the prediction by evaluating the equation in parallel. First, we conduct a vector to matrix multiplication in parallel to obtain all the needed kernel values, where the vector is $\boldsymbol{x}_j$ and the matrix consists of all the support vectors. Then, the sum of the equation can be performed using a parallel reduction.

3. Experimental Studies

We compare the efficiency on training SVMs for classification, regression and one-class SVMs (denoted by "OSVM"). Three representative data sets are listed in Table 1. We conducted our experiments on a workstation running Linux with two Xeon E5-2640 v4 10 core CPUs, 256GB main memory and a Tesla P100 GPU of 12GB memory. ThunderSVM are implemented in CUDA-C and C++ with OpenMP. We used the Gaussian kernel. Three pairs of hyper-parameters (C, γ) for the data sets are (10, 0.125), (256, 0.125), and (64,7.8125) and are the same as the existing studies (Wen et al., 2014, 2017c). More experimental evaluation can be found in the Appendix. ThunderSVM when using GPUs is over 100 times faster than LibSVM. When running on CPUs, it is over 10 times faster than LibSVM. This demonstrates the effectiveness of our algorithmic design. For prediction, ThunderSVM is also 10-100 times faster than LibSVM (Wen et al., 2017b). We also varied the hyper-parameters C from 0.01 to 100 and γ from 0.03 to 10, and ThunderSVM is tens to hundreds of times faster than LibSVM in the data sets.

4. Conclusion

In this paper, we present our software tool called "ThunderSVM" which supports all the functionalities of LibSVM. For ease of usage, ThunderSVM uses identical input command line options as LibSVM, and supports Python, R and Matlab. Empirical results show that ThunderSVM is generally a factor of 100x faster than LibSVM in all the functionalities when GPUs are used. When running purely on CPUs, ThunderSVM is often 10 times faster than LibSVM. We hope this significant efficiency improvement would help practitioners in the community quickly solve their problems and enable SVMs to solve more complex problems.

Appendix A. Formal Definitions of Support Vector Machine Algorithms

In this section, we formally define various SVM based algorithms including classification, regression and one-class SVMs. Following the name convention in the research community (and also in LibSVM), we use C-SVC for SVM classification with C as the regularization constant; we use ν-SVC for SVM classification with ν to control training error. Similarly, we use C-SVR and ν-SVR for the two types of SVM regression.

A.1 Support Vector Machine Classification

In this subsection, we present a few types of SVM classification algorithms. For consistency, we adopt the same notation as LibSVM (Chang and Lin, 2011) when appropriate. We also use different notations from LibSVM in some places where we think the different notations tend to be more natural.

A.1.1 C-Support Vector Machine Classification (C-SVC)

Formally, an instance $\boldsymbol{x}_i$ is attached with an integer $y_i \in \{+1, -1\}$ as its label. A positive (negative) instance is an instance with the label of $+1$ (-1). Given a set $\mathcal{X}$ of n training instances, the goal of the SVM training is to find a hyperplane that separates the positive and the negative training instances in the feature space induced by the kernel function with the maximum margin and meanwhile, with the minimum misclassification error on the training instances. The SVM training is equivalent to solving the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{w},\, \boldsymbol{\xi},\, b}{\text{argmin}} \quad & \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 - \xi_i \\ & \xi_i \geq 0,\ \forall i \in \{1, ..., n\} \end{aligned} \tag{1}$$

where $\boldsymbol{w}$ is the normal vector of the hyperplane, C is the regularization constant, $\boldsymbol{\xi}$ is the slack variables to tolerant some training instances falling in the wrong side of the hyperplane, and b is the bias of the hyperplane. The form of the optimization Problem (1) is call the *primal form* of the SVM training.

To handle nonlinearly separable data, the above optimization problem is solved in the dual form shown below where mapping functions can be implicitly applied with lower computation cost.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} . \quad & \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\boldsymbol{\alpha}^T\boldsymbol{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \forall i \in \{1, ..., n\}, \\ & \sum_{i=1}^{n} y_i\alpha_i = 0 \end{aligned} \tag{2}$$

where α_i is the Lagrange multiplier and denotes the *weight* of $\boldsymbol{x}_i$; $\boldsymbol{Q}$ denotes an $n \times n$ matrix $[Q_{i,j}]$ and $Q_{i,j} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, and $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a kernel value computed from a kernel function; C is the same regularization constant as in Problem (1). The kernel values of all the training instances form a *kernel matrix* (Weinberger et al., 2004).

A.1.2 ν-Support Vector Machine Classification (ν-SVC)

The ν-SVMs for classification is proposed by Schölkopf et al. (2000). The hyper-parameter ν is introduced to control training errors and the number of support vectors. Formally, the optimization problem of training ν-SVC is shown below.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} . \quad & -\frac{1}{2}\boldsymbol{\alpha}^T \boldsymbol{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq 1, \forall i \in \{1, ..., n\}, \\ & \boldsymbol{e}^T\boldsymbol{\alpha} = \nu n, \quad \sum_{i=1}^{n} y_i \alpha_i = 0 \end{aligned} \tag{3}$$

where $\boldsymbol{e}$ is a vector of all one, and the other notations have the same meaning of those in C-SVC shown in Problem (2).

A.1.3 Multi-class Support Vector Machine Classification

Multi-class SVM classification is implemented via pairwise coupling (also used in LibSVM) which has shown superiority over other methods (Hsu and Lin, 2002). During training multi-class SVMs, many binary SVMs are trained using SMO. Given the training data set, the data set is first decomposed into multiple subsets of two classes. Then, we obtain a binary problem (s, t) that consists of all the instances of class s and t. There are $\frac{k(k-1)}{2}$ binary SVM classifiers in total, where k is the number of classes in the data set. After training the binary SVM classifiers, the predicted values of $\text{SVM}_{s,t}$ on the binary problem (s, t) are used for the final class label prediction (e.g., through majority voting).

A.1.4 Support Vector Machines with Probabilistic Output

To obtain probabilistic output, the predicted values of the $\frac{k(k-1)}{2}$ binary SVM classifiers (described in Section A.1.3) on the training instances are used to train the sigmoid function.

$$P(y_i = 1|\boldsymbol{x}_i) = \frac{1}{1 + \exp(Av_i + B)} \tag{4}$$

where v_i is the decision value predicted by an SVM classifier, and the parameters A and B can be obtained by maximizing the following log likelihood.

$$\begin{aligned} &\max_{A,B} . \; F = \sum_{i=1}^{n} t_i \log(P(y_i = 1|\boldsymbol{x}_i)) - (1 - t_i)\log(1 - P(y_i = 1|\boldsymbol{x}_i)) \\ &\text{where } t_i = \begin{cases} \frac{N_+ + 1}{N_+ + 2} & \text{if } y_i = +1 \\ \frac{1}{N_- + 2} & \text{if } y_i = -1, \end{cases} \end{aligned} \tag{5}$$

N_+ denotes the number of positive training instances, and N_- denotes the number of negative training instances. Newton's method with backtracking is a commonly used approach to solve the above optimization problem (Lin et al., 2007) and is implemented in ThunderSVM. The aforementioned SVMs with probabilistic output is for multi-class SVMs. For binary SVMs with probabilistic output, the sigmoid function can be directly used for prediction.

A.2 Support Vector Machine Regression

In what follows, we discuss two types of SVM regression algorithms: ϵ-SVR and ν-SVR.

A.2.1 The ϵ-Support Vector Machine Regression (ϵ-SVR)

Similar to SVC, given a set $\mathcal{X}$ of training instances, $\mathcal{X} = \{\boldsymbol{x}|\boldsymbol{x}_i \in \mathbb{R}^d, i = 1, ..., n\}$, where n denotes the data set cardinality and d denotes the data dimensionality. In SVR, each training instance is associated with a target value $z \in \mathbb{R}$ instead of a label. The SVM regression is a function estimation process that finds an optimal function $g(\boldsymbol{x})$ which minimizes the difference of the target value of each training instance $\boldsymbol{x}_i$ and the function value $g(\boldsymbol{x}_i)$. Meanwhile, the function $g(\boldsymbol{x})$ is as smooth as possible for achieving high accuracy in predicting unseen data. The training process of the SVM regression is equivalent to solving the following quadratic programming problem.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^{2n} s_i\alpha_i - \frac{1}{2}\boldsymbol{\alpha}^{\boldsymbol{T}}\mathbf{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \forall i \in \{1, ..., 2n\}, \\ & \sum_{i=1}^{2n} y_i\alpha_i = 0 \end{aligned} \tag{6}$$

where $\boldsymbol{\alpha} = \langle \alpha_1, \alpha_2, ..., \alpha_{2n} \rangle$ is Lagrange multipliers for the training instances; the value α_i denotes the contribution of a training instance $\boldsymbol{x}_t$ to the estimated function, where $t = i$ if $i \leq n$, otherwise $t = i - n$; y_i is computed by

$$\begin{cases} y_i = +1, \ s_i = \varepsilon - z_i & \text{if } i \leq n \\ y_i = -1, \ s_i = \varepsilon + z_i & \text{if } n < i \leq 2n \end{cases}$$

where ε is the error tolerant parameter.

A.2.2 The ν-Support Vector Machine Regression (ν-SVR)

Similar to ν-SVC, the hyper-parameter ν is introduced to control the training errors and the number of support vectors. The optimization problem of ν-SVR is defined as follows.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & -\sum_{i=1}^{2n} z_i\alpha_i - \frac{1}{2}\boldsymbol{\alpha}^{\boldsymbol{T}}\mathbf{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C/n, \forall i \in \{1, ..., 2n\}, \\ & \sum_{i=1}^{n} \alpha_i - \sum_{i=n+1}^{2n} \alpha_i = 0 \\ & \sum_{i=1}^{2n} \alpha_i = C\nu \end{aligned} \tag{7}$$

where the notations have the same meaning as those in ϵ-SVR.

A.3 One-class Support Vector Machines (OSVMs)

OSVMs were first introduce by Schölkopf et al. (2001), and can be used for distribution estimation and outlier detection (Hodge and Austin, 2004). Formally, the optimization problem is defined below.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & -\frac{1}{2}\boldsymbol{\alpha}^{\boldsymbol{T}}\mathbf{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq 1, \forall i \in \{1, ..., n\}, \\ & \sum_{i=1}^{n} \alpha_i = \nu n \end{aligned} \tag{8}$$

The notations used in Problem (8) are the same as the previous optimization problems.

A.4 The Sequential Minimal Optimization Algorithm

All of the above optimization are convex and can be solved by various types of solvers. The goal of the training is to find a vector $\boldsymbol{\alpha}$ of Lagrange multipliers that maximizes the value of the objective function. Here, we describe a popular training algorithm, the Sequential Minimal Optimization (SMO) algorithm (Platt, 1998). SMO iteratively improves the vector until the optimal condition of the SVM is met. The optimal condition is reflected by an *optimality indicator vector* $\boldsymbol{f} = \langle f_1, f_2, ..., f_n \rangle$ where f_i is the optimality indicator for the i^{th} instance $\boldsymbol{x}_i$ and f_i can be obtained using the following equation: $f_i = \sum_{j=1}^{n} \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) - y_i$. Notably, the number of training instances in SMO for SVM regression is $2n$ due to the conversion from SVR to SVC. What we discuss here can be simply applied in SVR by changing n to $2n$. During the SVM training, the SMO algorithm has the following three steps in each iteration.

Step 1: Search two extreme training instances, denoted by $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, which have the maximum and minimum optimality indicators, respectively. It has been proven (Keerthi et al., 2001; Fan et al., 2005) that the indices of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by u and l respectively, can be computed by the following equations.

$$u = \underset{i}{\operatorname{argmin}}\{f_i | \boldsymbol{x}_i \in \mathcal{X}_{upper}\} \tag{9}$$

$$l = \underset{i}{\operatorname{argmax}}\{\frac{(f_u - f_i)^2}{\eta_i} | f_u < f_i, \boldsymbol{x}_i \in \mathcal{X}_{lower}\} \tag{10}$$

where

$$\begin{aligned} \mathcal{X}_{upper} &= \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3, \\ \mathcal{X}_{lower} &= \mathcal{X}_1 \cup \mathcal{X}_4 \cup \mathcal{X}_5; \end{aligned}$$

and

$$\begin{aligned} \mathcal{X}_1 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, 0 < \alpha_i < C\}, \\ \mathcal{X}_2 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = 0\}, \\ \mathcal{X}_3 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = C\}, \\ \mathcal{X}_4 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = C\}, \\ \mathcal{X}_5 &= \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = 0\}; \end{aligned}$$

and $\eta_i = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_i)$; f_u and f_l denote the optimality indicators

of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, respectively. This approach for selecting the two training instances is also known as "second order heuristics".

Step 2: Improve the Lagrange multipliers of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by α_u and α_l, by updating them using Equations (11) and (12).

$$\alpha_l' = \alpha_l + \frac{y_l(f_u - f_l)}{\eta} \tag{11}$$

$$\alpha_u' = \alpha_u + y_l y_u(\alpha_l - \alpha_l') \tag{12}$$

where $\eta = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_l, \boldsymbol{x}_l) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_l)$. To guarantee the update is valid, when α_u' or α_l' exceeds the domain of $[0, C]$, α_u' and α_l' are adjusted into the domain.

Step 3: Update the optimality indicators of all the instances. The optimality indicator f_i of the instance $\boldsymbol{x}_i$ is updated to f_i' using the following formula:

$$f_i' = f_i + (\alpha_u' - \alpha_u)y_u K(\boldsymbol{x}_u, \boldsymbol{x}_i) + \ (\alpha_l' - \alpha_l)y_l K(\boldsymbol{x}_l, \boldsymbol{x}_i) \tag{13}$$

SMO repeats the above steps until the optimal condition is met, i.e., $f_u \geq f_{max}$, where

$$f_{max} = max\{f_i | \boldsymbol{x}_i \in \mathcal{X}_{lower}\} \tag{14}$$

After the optimal condition is met, we obtain the $\boldsymbol{\alpha}$ values which corresponding to the optimal hyperplane and the SVM with these $\boldsymbol{\alpha}$ values is considered *trained*. Algorithm 1 summarizes the whole training process. In Algorithm 1, $\boldsymbol{\mathcal{K}}_u$ and $\boldsymbol{\mathcal{K}}_l$ correspond to the u^{th} and the l^{th} rows of the kernel matrix, respectively.

Algorithm 1: The SMO algorithm

Input: a training set $\mathcal{X}$ of n instances with labels $\boldsymbol{y}$
Output: a weight vector $\boldsymbol{\alpha}$

```
for i ← 1 to n do                                   /* initialize α and f */
    α_i ← 0, f_i ← −y_i
repeat
    search for f_u and u using Equation (9);
    compute kernel values K_u                       /* u^th row */
    search for f_l and l using Equation (10);
    compute kernel values K_l                       /* l^th row */
    update α_u and α_l using Equations (11) and (12);
    update f using Equation (13);
    search for f_max using Equation (14);
until f_u ≥ f_max
```

A.4.1 Using a larger working set in SMO

Instead of using a working set of size two, ThunderSVM uses a bigger working set and solve multiple subproblems of SMO in a batch. We precompute all the kernel values for the working set and store them in a GPU buffer which is a preallocated space on the GPU global memory. In each time we update the working set, q (where $q \geq 2$) instances in the

working set will be replaced with q new violating instances, such that (i) q rows of the kernel matrix can be computed at once to make efficient use of the GPU and reduce GPU memory accesses, and (ii) the kernel values in the GPU buffer can be reused (i.e., GPU buffer size is larger than q).

In the following, we first discuss our approach to select the q violating instances to refresh the working set. Then, we provide more details of computing kernel values for the q instances and store them to a GPU buffer.

Selecting q violating instances: Our intuition for updating the working set is to choose q training instances that violate the optimality condition the most, such that the current SVM can be potentially improved the most (Joachims, 1998). Hence, we sort the optimality indicators ascendingly. Then, we choose the top $\frac{q}{2}$ training instances whose $y_i\alpha_i$ can be increased; and we choose the bottom $\frac{q}{2}$ training instances whose $y_i\alpha_i$ can be decreased. We consider $y_i\alpha_i$, because of the constraints $\sum y_i\alpha_i = 0$ and $0 \leq \alpha_i \leq C$.

Maintaining a GPU buffer for kernel values: Once the q violating instances are selected, we compute all the kernel values related to the q instances (i.e., q rows of the kernel matrix). Computing those kernel values is essentially matrix multiplication between the q instances and the rest of the training instances, because computing a kernel value can be viewed as a dot product of two vectors. Thus, the kernel value computation here can be efficiently carried out by the cuSPARSE library (Nvidia, 2008).

The kernel values computed here are stored in a GPU buffer on the GPU global memory, as the kernel values will be repeatedly used by SMO while improving the current SVM with the updated working set. Note that the SMO in our algorithm only considers the instances in our working set, which is different from the original SMO that needs to consider all the training instances in every iteration. We allocate a GPU memory buffer which can store $m \times q$ rows of the kernel matrix (i.e., allow m batches to be stored in the buffer). The first-in first-out batch replacement strategy is used when the buffer is full. Although other strategies may be more effective, we find first-in first-out simple and sufficiently effective in our algorithm.

Appendix B. Experimental Studies

In this section, we empirically study the efficiency of ThunderSVM in comparison with LibSVM. We compare the efficiency on training SVMs for classification, regression and one-class SVMs (denoted by "OSVM"). Regarding SVMs for classification and regression, we study both C-SVC, ν-SVC, ϵ-SVR and ν-SVR. The five representative data sets we used are listed in Table 2. We conducted our experiments on a workstation running Linux with two Xeon E5-2640 v4 10 core CPUs, 256GB main memory and an NVIDIA Tesla P100 GPU of 12GB memory. ThunderSVM are implemented in CUDA-C and C++ with OpenMP. We used Gaussian kernel and hyper-parameters C and γ for the kernel on each data set are the same as the existing studies (Wen et al., 2014, 2017c). LibSVM uses 12GB (which is the same as the GPU memory) of main memory for kernel value caching.

data set			elapsed time (sec)			speedup	
name	cardinality	dimension	ThunderSVM		LibSVM		
			GPU	CPU		GPU	CPU
MNIST8m (svc)	$8.1\text{x}10^6$	784	$7.1\text{x}10^3$	$3.2\text{x}10^4$	$8.1\text{x}10^5$	114	39.9
RCV1_test (svc)	677399	47236	621	20633	$8.3\text{x}10^5$	1337	40
Epsilon (svc)	400000	2000	1251	26042	1 week+	483+	23+
E2006-tfidf (svr)	16087	150360	13.25	343.5	9161	691	26.7
Webdata (osvm)	49749	300	4.66	16.5	1493	320	90.5

Table 2: Comparison between ThunderSVM with LibSVM

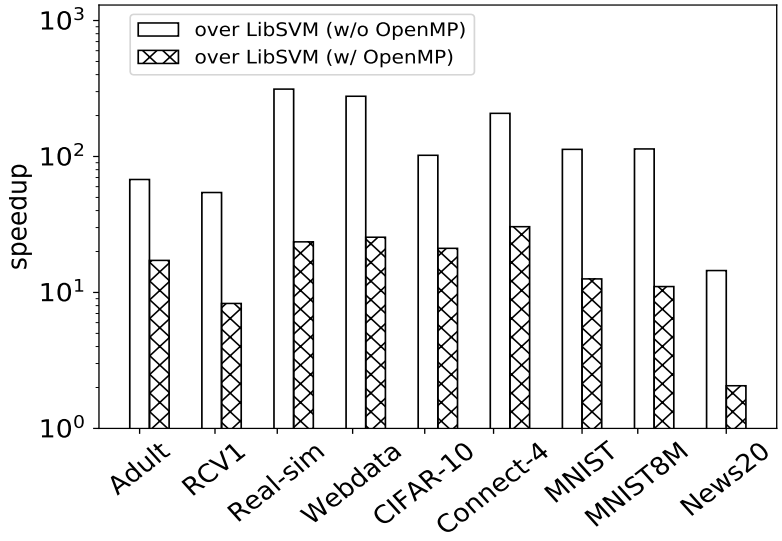

Figure 2: Training time speedup of ThunderSVM over LibSVM with/without OpenMP

B.1 Efficiency and Final SVM Comparison

We study the performance of ThunderSVM and LibSVM using more data sets and compare the final results produced by the two libraries. LibSVM with OpenMP uses 40 threads, which achieves the best performance. Some results shown here are from another work of ours (Wen et al., 2017a).

Efficiency comparision: Figure 2 shows the speedup on training of ThunderSVM over LibSVM. As we can see from the results, ThunderSVM consistently outperforms LibSVM without OpenMP by one to two orders of magnitude, LibSVM with OpenMP by 10x times. Figure 3 shows the results of speedup on prediction of ThunderSVM over LibSVM. As we can see from the figure, ThunderSVM consistently outperforms LibSVM without OpenMP by two orders of magnitude. When OpenMP is used for LibSVM, ThunderSVM still outperforms it by more than 10 times.

Final SVM comparison: ThunderSVM and LibSVM produce identical SVMs, because ThunderSVM can be viewed as a highly parallelized version of LibSVM. We have measured the training error to confirm if ThunderSVM produces almost identical results as LibSVM, and the results are shown in Table 3. Notably, the difference of the trained SVMs

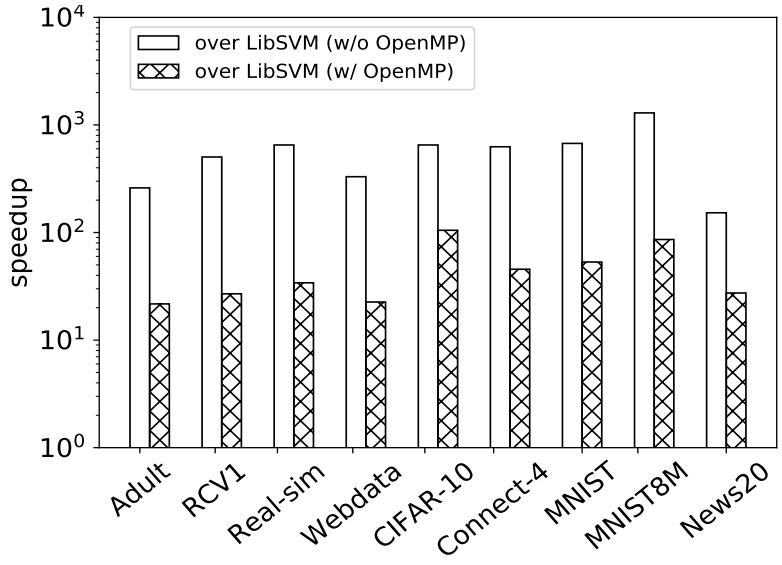

Figure 3: Prediction speedup of ThunderSVM over LibSVM with/without OpenMP

Table 3: Final classifier comparison between LibSVM and ThunderSVM

Data set	bias		training error	
	LibSVM	ThunderSVM	LibSVM	ThunderSVM
Adult	-0.510	-0.510	4.4%	4.4%
RCV1	-0.512	-0.512	0.11%	0.11%
Real-sim	-1.061	-1.061	0.27%	0.27%
Webdata	-0.936	-0.947	1.92%	1.92%
CIFAR-10	0.0245	0.0245	0.35%	0.35%
Connect-4	0.233	0.233	4.39%	4.39%
MNIST	0.360	0.360	0%	0%
MNIST8M	-7.339	-7.339	0%	0%
News20	-0.0016	-0.0016	2.23%	2.23%

by ThunderSVM and LibSVM is due to the floating point precision of a parallel program and sequential program; the difference is almost negligible in terms of the prediction accuracy of the trained model. As we can see from the results, the training errors are identical, which implies that ThunderSVM and LibSVM produce the same (at least highly similar) SVMs. To further confirm the SVMs trained by ThunderSVM and LibSVM are the same, we also compare the bias of the trained SVMs, and the results are shown in Column "bias" of Table 3. Note that we used the bias of the last binary SVM for the multi-class problems. As we can see from the results, the biases of SVMs trained by ThunderSVM are almost the same to those of LibSVM. Note that existing studies in machine learning commonly compare the difference of $||\vec{w}||$ of two algorithms. However, it is impossible for kernelized SVMs, because $||\vec{w}||$ is in the data space induced by the kernel function.

Appendix C. Related Work

We categorize the most relevant related work into two categories: the studies dedicated to training SVMs using CPUs and the studies dedicated to training SVMs using GPUs.

Training SVMs using CPUs: Platt (1998) proposed the SMO algorithm which is simple and efficient, and hence SMO is used in LibSVM, WEKA (Hall et al., 2009) and Catanzaro's algorithm (2008). Other studies in training linear SVMs, such as Joachims' algorithm using cutting plane (2009) and "Pegasos" (Shalev-Shwartz et al., 2011), cannot apply nonlinear kernels. Training SVMs in distributed environment, such as MapReduce SVMs (Catak and Balaban, 2012) and MPI SVMs (Cao et al., 2006), is inefficient due to the iterative nature of the SVM training and costly network communication. Another related study is binary SVMs with probabilistic output proposed by Platt (1999).

Training SVMs using GPUs: Catanzaro et al. (2008) first introduced GPUs for training binary SVMs. Wen et al. (2014) proposed GPU based binary SVM cross-validation by precomputing the whole kernel matrix which is stored in high-speed storage (for example SSDs). A recent study extended the algorithm for SVM regression problems (Wen et al., 2017c). Athanasopoulos et al. (2011) used GPUs to purely accelerate the kernel matrix computation in the SVM training. These studies are for training binary SVMs and cannot handle large data sets, because the size of kernel matrix is quadratic in the number of instances. For example, processing the MNIST8M data set with those algorithms needs 256TB of storage which is unacceptable for GPUs. Herrero-Lopez et al. (2010) used one-against-all method to solve multi-class problems on GPUs. However, they represented the training instances in dense format for the ease of implementation and better memory alignment. The dense representation makes the above algorithms difficult to handle large but sparse data sets. Another study (Vaněk et al., 2017) compares different GPU SVM implements and provides some benchmark results, and proposes the OHD-SVM algorithm. However, the work only focuses on binary SVMs and no multi-class SVMs or probabilistic SVMs are presented. Cotter et al. (2011) represented training instances in sparse format, that is CSR format (Buluç et al., 2009), and proposed a clustering technique to make use of the data sparseness. We also use CSR format to represent the training data for handling large but sparse data sets. We call Cotter et al.'s algorithm GTSVM. GTSVM supports both binary and multi-class SVMs, but does not support multi-class probability estimation due to their mathematical modelling.

References

Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. GPU acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, 2011.

Todd Bodnar and Marcel Salathé. Validating models for disease detection using twitter. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 699–702. ACM, 2013.

Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Annual Symposium on Parallelism in Algorithms and Architectures*, pages 233–244. ACM, 2009.

Li Juan Cao, S Sathiya Keerthi, Chong Jin Ong, Jian Qiu Zhang, Uvaraj Periyathamby, Xiu Ju Fu, and HP Lee. Parallel sequential minimal optimization for the training of support vector machines. *IEEE Trans. Neural Networks*, 17(4):1039–1049, 2006.

F Ozgur Catak and M Erdal Balaban. CloudSVM: training an SVM classifier in cloud computing systems. In *Joint International Conference on Pervasive Computing and the Networked World*, pages 57–68. Springer, 2012.

Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111. ACM, 2008.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

Andrew Cotter, Nathan Srebro, and Joseph Keshet. A gpu-tailored approach for training kernelized svms. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 805–813. ACM, 2011.

Cristian Dittamo and Antonio Cisternino. GPU White paper, 2008.

Vito DOrazio, Steven T Landis, Glenn Palmer, and Philip Schrodt. Separating the wheat from the chaff: applications of automated document classification using Support Vector Machines. *Political Analysis*, 22(2):224–242, 2014.

Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training Support Vector Machines. *The Journal of Machine Learning Research*, 6:1889–1918, 2005.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

Sergio Herrero-Lopez, John R Williams, and Abel Sanchez. Parallel multiclass classification using svms on gpus. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 2–11. ACM, 2010.

Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass Support Vector Machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, 2002.

Thorsten Joachims. Making large-scale svm learning practical. Technical report, Technical report, SFB 475: Komplexitätsreduktion in Multivariaten Datenstrukturen, Universität Dortmund, 1998.

Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural svms. *Machine Learning*, 77(1):27–59, 2009.

Mohamad Kachuee, Mohammad Mahdi Kiani, Hoda Mohammadzade, and Mahdi Shabany. Cuff-less high-accuracy calibration-free blood pressure estimation using pulse transit time. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1006–1009. IEEE, 2015.

S. Sathiya Keerthi, Shirish Krishnaj Shevade, Chiranjib Bhattacharyya, and Karuturi Radha Krishna Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, 2001.

Hsuan-Tien Lin, Chih-Jen Lin, and Ruby C Weng. A note on platt's probabilistic outputs for Support Vector Machines. *Machine Learning*, 68(3):267–276, 2007.

Duane Merrill. CUB v1. 5.3: CUDA Unbound, a library of warp-wide, block-wide, and device-wide GPU parallel primitives, 2015.

CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

Edoardo Pasolli, Farid Melgani, Devis Tuia, Fabio Pacifici, and William J Emery. SVM active learning approach for image classification using spatial information. *IEEE Transactions on Geoscience and Remote Sensing*, 52(4):2217–2233, 2014.

John Platt. Sequential Minimal Optimization: A fast algorithm for training Support Vector Machines, 1998.

John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

Volker Roth. Kernel fisher discriminants for outlier detection. *Neural Computation*, 18(4): 942–960, 2006.

Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. New support vector algorithms. *Neural Computation*, 12(5):1207–1245, 2000.

Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming*, 127(1):3–30, 2011.

Shirish Krishnaj Shevade, S Sathiya Keerthi, Chiranjib Bhattacharyya, and Karaturi Radha Krishna Murthy. Improvements to the SMO algorithm for SVM regression. *IEEE Transactions on Neural Networks*, 11(5):1188–1193, 2000.

Amber Thomas. Kaggle 2017 survey results: `https://www.kaggle.com/amberthomas/kaggle-2017-survey-results`, 2017.

Jan Vaněk, Josef Michálek, and Josef Psutka. A GPU-architecture optimized hierarchical decomposition algorithm for Support Vector Machine training. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3330–3343, 2017.

Kilian Q Weinberger, Fei Sha, and Lawrence K Saul. Learning a kernel matrix for nonlinear dimensionality reduction. In *International Conference on Machine Learning*, page 106. ACM, 2004.

Zeyi Wen, Rui Zhang, Kotagiri Ramamohanarao, Jianzhong Qi, and Kerry Taylor. MASCOT: Fast and highly scalable SVM cross-validation using GPUs and SSDs. In *IEEE International Conference on Data Mining (ICDM)*, pages 580–589, 2014.

Zeyi Wen, Jiashuai Shi, Bingsheng He, Jian Chen, and Yawen Chen. Efficient multi-class probabilistic SVMs on GPUs, 2017a.

Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, and Jian Chen. Supplementary material of ThunderSVM: `https://github.com/zeyiwen/thundersvm/blob/master/thundersvm-full.pdf`, 2017b.

Zeyi Wen, Rui Zhang, Kotagiri Ramamohanarao, and Li Yang. Scalable and fast SVM regression using modern hardware. *World Wide Web*, pages 1–27, 2017c.