Training Support Vector Machines on Multiprocessors and GPUs

鄧偉祥
10567212

林翰緯
0456808

ABSTRACT

We apply sequential minimal optimization (SMO) algorithm, which is one popular algorithm for training support vector machine (SVM), on MNIST database with multiprocessors and GPU. We partition the entire training dataset into smaller subsets and then use multiple cores (MPI) or threads (CUDA) to deal with each of the partitioned subsets. It shows our results are identical with the sequential version either using MPI or CUDA. The efficiency on 2 or 4 processors are about 0.7 and the speedup on GPU is 55.12.

1 INTRODUCTION

Support Vector Machine (SVM) is one of the best supervised learning algorithm which is used to analyze data for classification, regression. It has been used in real world application massively, such as handwritten digit recognition, object recognition, face detection and text categorization, etc. SVM is essentially a quadratic programming (QP) problem which can be stated as follows: Given training vector $x_i \in R^n, i = 1, \dots, l$ in two classes, and an indicator vector $\mathrm{y} \in R^l$ such that $y_i \in \{1, -1\}$, we want to solve the following optimization problem. (Here we only focus on C-SVC (C-Support Vector Classification), to see other types of SVMs, please refer to [5])

$$\min_{w,b,\xi} \frac{1}{2} w^T w + \mathrm{C} \sum_{i=1}^{l} \xi_i$$
$$\text{subject to } y_i(w^T \Phi(x_i) + b) \geq 1 - \xi_i$$
$$\xi_i \geq 0, \mathrm{i} = 1, \dots, l,$$

where $\Phi(x_i)$ maps x_i into a higher-dimensional space and C > 0 is the regularization parameter. We can use math skill turning this problem into dual problem (see appendix A for details):

$$\max_{\alpha} \mathrm{W}(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} y^{(i)} y^{(j)} \alpha_i \alpha_j K(x_i, x_j)$$
$$\text{subject to } \alpha_i \geq 0, i = 1, \dots, l$$
$$\sum_{i=1}^{l} \alpha_i \, y^{(i)} = 0$$

where $K(x_i, x_j) \equiv \Phi(x_i)^T \Phi(x_j)$ is the kernel function. In our implementation we only use Gaussian kernel $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$ (The brief illustration of $K(x_i, x_j)$ is in appendix B). If we could solve α_i for each training vector x_i, then the decision function is:

$$\mathrm{sgn}(\sum_{i=1}^{l} y_i \, \alpha_i \, K(x_i, x) + b)$$

If the output of this function for test data *x* is positive, then the predicted *y* = +1, if *x* is negative, then *y* = -1. We can use this function applying on a dataset so called test set other than training dataset to see whether the training result can predict unknown data correctly.

To solve the dual problem, many efforts have been done to reduce the training time. The most effective one is modified SMO algorithm proposed by [3]. It was based on Platt's SMO algorithm which updates one pair α_i and α_j, while holding all the other $\alpha_k's$ ($k \neq i, j$) fixed. Then tries to make the biggest progress towards the global maximum. And do it iteratively until converged.

Although modified SMO algorithm is fast, facing a binary task composed 100,000 points with hundreds of dimensions can often take on the order of hours of serial execution, that's why we think parallel computing will help.

2 SERIAL SMO ALGORITHM

Sequential Minimal Optimization (SMO) algorithm was first suggested by Platt [2]. It follows Osuna's theorem that decomposes the overall QP problem into QP sub-problems to ensure convergence. Unlike the previous methods, SMO chooses two Lagrange multipliers α_i to jointly optimize, finds the optimal values for these multipliers, and updates the SVM to reflect the new optimal values at every iterative step. The advantage of SMO is that numerical QP optimization can be entirely avoided, solving two Lagrange multipliers can be done analytically. After that, Keerthi et al. modified SMO to speed up the execution time of the original SMO. The details of modified SMO can be found in [3]. Here we only briefly overview the modified SMO. First we categorize the all indexes of training set as I_0, I_1, I_2, I_3, I_4 subsets, each of these denotation are $I_0 = \{i: y_i = 1, 0 < \alpha_i < C\} \cup \{i: y_i = -1, 0 < \alpha_i < C\}$, $I_1 = \{i: y_i = 1,\ \alpha_i = 0\}$, $I_2 = \{i: y_i = -1,\ \alpha_i = C\}$, $I_3 = \{i: y_i = 1,\ \alpha_i = C\}$, $I_4 = \{i: y_i = -1,\ \alpha_i = 0\}$. The error function f_i is defined as $\sum_{j=1}^{l} \alpha_j y_j K(x_j, x_i) - y_i$, then we define b_{up} as $\min\{f_i: i \in I_0 \cup I_1 \cup I_2\}$, I_{up}=$\arg\min_i f_i$, b_{low} as $\max\{f_i: i \in I_0 \cup I_3 \cup I_4\}$, I_{low}=$\arg\max_i f_i$. Earlier we mentioned that SMO updates two Lagrange multipliers at one step which will be selected to associate with b_{up} and b_{low} in modified SMO. The update functions of two Lagrange multipliers are defined as follows:

$$\alpha_{I_{\text{up}}}^{new} = \alpha_{I_{\text{up}}}^{old} - \frac{y_{I_{\text{up}}}(f_{I_{\text{low}}}^{old} - f_{I_{\text{up}}}^{old})}{\eta}$$

$$\alpha_{I_{\text{low}}}^{new} = \alpha_{I_{\text{low}}}^{old} + y_{I_{\text{up}}} y_{I_{\text{low}}} (\alpha_{I_{\text{up}}}^{old} - \alpha_{I_{\text{up}}}^{new})$$

where $\eta = 2K\left(x_{I_{\text{low}}}, x_{I_{\text{up}}}\right) - K\left(x_{I_{\text{low}}}, x_{I_{\text{low}}}\right) - K\left(x_{I_{\text{up}}}, x_{I_{\text{up}}}\right)$, and $\alpha_{I_{\text{up}}}^{new}$ and $\alpha_{I_{\text{low}}}^{new}$ need to clipped to $[0, C]$. There are still two variables, *Dual* and *DualityGap* which are used for checking the convergence of the program. To see the definitions of these two variables, please refer to [4]. A simple description of modified SMO can be summarized as table 1.

Table 1: Sequential SMO

Sequential SMO Algorithm:
Initial $\alpha_i = 0, f_i = -y_i, Dual = 0, i = 1, \dots, l$ Calculate $b_{\text{up}}, I_{\text{up}}, b_{\text{low}}, I_{\text{low}}, DualityGap$
Until $DualityGap \le \tau\lvert Dual\rvert$ (1) Update $\alpha_{I_{\text{up}}}, \alpha_{I_{\text{low}}}$ (2) Update $f_i, i = 1, \dots, l$ (3) Calculate $b_{\text{up}}, I_{\text{up}}, b_{\text{low}}, I_{\text{low}}, DualityGap, Dual$ Repeat

Our implementation of sequential SMO algorithm is following the pseudo-code mentioned in appendix A in [4]. The format of input file is the same as famous SVM library libsvm [5]. The output file will contain the needed information such as α_i, support vector x_i(when $\alpha_i \neq 0$) for latter testing the accuracy of prediction of new dataset. To know how to run our code please refer to appendix C.

3 PROPOSED SOLUTIONS

The complexity of updating f_i at one step is $\Theta(l \times n)$, where l is the number of training vector x_i and n is the dimension of x_i, which implies that this part will be very time consuming. The experiment result supports our assumption that we run our sequential code on SS-lab gpu server and find out updating error function f_i takes about 99% of whole execution time.

Fortunately, f_i can be regarded as a l-dimension vector and updating elements of $f_i$is independent with each other. We managed to use multiple processors of CPU and threads of GPU to parallel compute f_i at each iterative step respectively. In the following two sub-sections the strategies using MPI and CUDA will be described.

3.1 MPI

To reduce the time of calculating f_i , we need to assign jobs to each processor equally. So in our strategy, root processor will take this job. At the beginning of the program, the root processor takes charge of the calculation of all the parameters needed by calculating f_i and then broadcast them to each processor. Every processor including root one will calculate their part of f_i . After all processors are done, the root processor needs to collect and merge the results and then decides if the new round is needed. If yes, the whole new round will start again. If no, the root processor will handle the last job and the others processors will terminate. This loop could run thousands of times depending on the training data and parameters.

3.2 CUDA

Although updating f_i can be parallel computed, we come up with another approach to solve this problem at first. Since kernel function $K(x_j, x_i)$ is a $l \times l$ matrix whose element is calculated from two $n \times 1$ vectors. This is very similar with matrix multiplication so that we can calculate it in advance of running iterative steps. Then this matrix multiplication could be accelerated with GPU and be stored in memory for later use.

To explain why $K(x_j, x_i)$ is a $l \times l$ matrix and its elements are related with two $n \times 1$ vectors, we have to illustrate what is kernel function $K(x_j, x_i)$. In introduction we mentioned $K(x_i, x_j) \equiv \Phi(x_i)^T \Phi(x_j)$ where $\Phi(x)$ denotes the feature mapping which maps from the attributes to the features. Although given $\Phi(x)$, we could easily compute $K(x_j, x_i)$ by finding $\Phi(x_i)$ and $\Phi(x_j)$ and taking their inner product. But often $\Phi(x)$ itself is very expensive to calculate (perhaps because it is an extremely high dimensional vector). The usually way is to calculate $K(x_j, x_i)$ directly without having to explicitly find vectors $\Phi(x)$.

In our case we use Gaussian kernel $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$ and $x_i \in R^n, i = 1, \dots, l$. That means $K(x_i, x_j)$ for certain i, j, is a number related with the distance of two vectors $x_i, x_j \in R^n$. Therefore we can assign threads in GPU for every elements in $K(x_i, x_j)$, and every threads are in charge of calculating the distance of two certain vectors and exponentiate its square. We also use the shared memory in each streaming multiprocessor of GPU to reduce the access of global memory. Although this method gives us pretty good result in small dataset (like 10,000 training set), dealing with very large data will fail because of storing $K(x_j, x_i)$ in advanced of running program needs a lot of memory. That motivates us to find another approach to solve this problem.

Like we mentioned before, updating each element of f_i is independent so we assign a thread in GPU for updating one element of f_i. It's worth to mention that we use 32 thread block for 60,000 training set which will give us best execution time as figure 1 shows. In our case we only need 60,000 threads running in GPU so that thread block size can't be too large or the grid size will be much less than 1,000 which is not recommended.

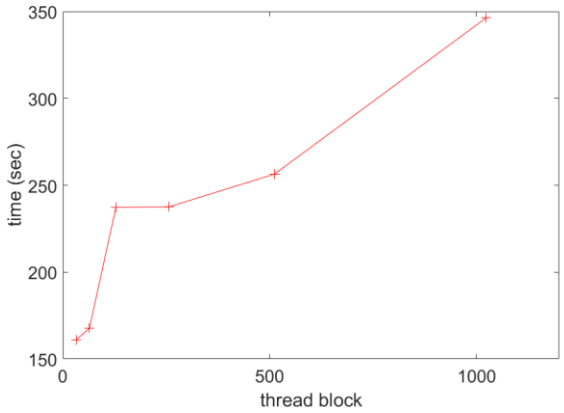

Figure 1

4 EXPERIMENTAL METHODOLOGY

To prove our sequential SMO implementation is correct, we compare our experimental result with programming exercise 6 in coursera course (https://www.coursera.org/learn/machine-learning). We get almost the same results as theirs (see figure 2 for their results, our numerical results are almost the same however we don't visualize them).

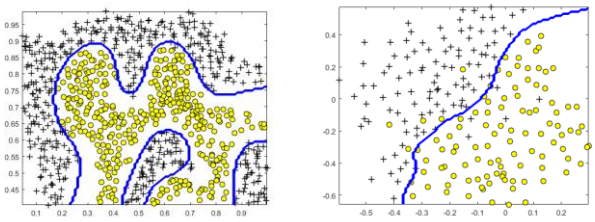

Figure 2

To let our execution time get longer so that we can see clearly our improvement of parallel implementation, we choose famous handwritten digits database MNIST [8] to be our test input. The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. There are four files available on that website. Two of them contain many grey level images with 28x28 pixels. Another two files have the labels indicate which digits the image represents. We use two matlab functions provided by [9] to load their files into a matrix in which each row represents a digit image (see figure 3).

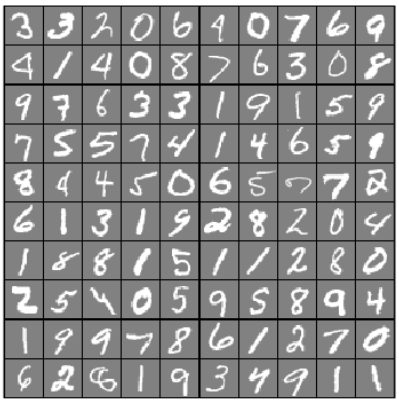

Figure 3: 100 images of MNIST dataset

We also write a matlab function called transformMNISTFormatToCSV which in file transformMNISTFormatToCSV.m to store the matrix as csv format files. Then use csvTolibsvmStyle.py python script to turn csv format file into libsvm style-like input which only stores non-zero pixels in the file so that the memory can be saved a lot.

Since SVM is for binary classification (here we don't consider multi-classes SVM for simplicity) we categorize the MNIST digits into even and odd two classes. To see the result of CUDA version1 implementation which can't deal with too large dataset, we trim training set of 60,000 examples to 10,000, and a test set of 10,000 examples to 1,500. And the two classes are 2 and 5 digits.

Our implementation of modified SMO algorithm for training SVM needs one libsvm style-like file and five parameters as input. The details of how to run our code is described in appendix C.

The MPI implementation is running on nplinux0.cs.nctu.edu.tw, nplinux1.cs.nctu.edu.tw, nplinux2.cs.nctu.edu.tw, nplinux3.cs.nctu.edu.tw

four servers connected with ssh protocol. The CUDA implementation is using SS-lab Tesla K20 GPU provided by professor Yi-Ping You's lab.

5 EXPERIMENTAL RESULTS

5.1 MPI

We implement our MPI version of program on the same server for MPI's homework. Because there are four PCs on the server side, we expected the speedup could be as near four times as possible. However the speedup of our experiment for 4 processors case is only near 3 which is a little bit lower than our expectation (see figure 4).

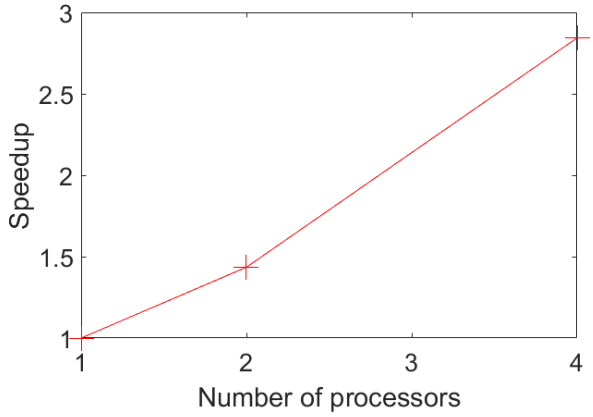

Figure 4: speedup for MPI result

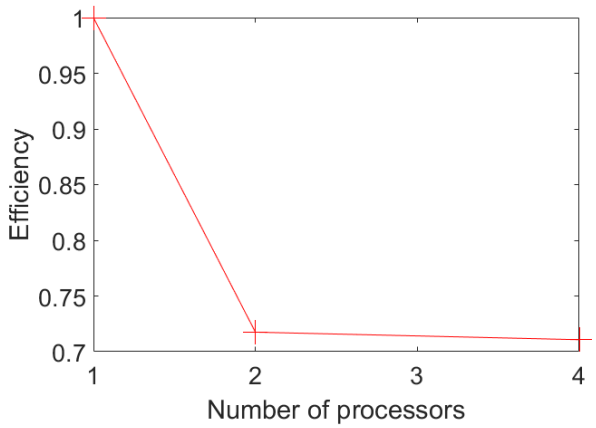

Figure 5: efficiency for MPI result

The reason why the speedup is only about 2.8 could be the processors need to wait each other to finish their round and then synchronize their data before start next round. The count of MPI_Bcast could be thousands times depending on the training data.

5.2 CUDA

At first, we get different results with sequential and parallel version. When we did the final presentation so many audiences including professor You questioning our correctness of parallel version. So we started to debug our CUDA code. Finally we found that if we change all the datatype "float" to "double" then the results of sequential and parallel version are identical. Although CUDA C program complies with IEEE 754 Standard for floating point which is also obeyed by C standard. We still have to be aware of the possibility of computational difference between x86 architecture and NVIDIA GPUs. For example, the Fused Multiply-Add (FMA) operation could be different on x86 architecture and NVIDIA GPUs [10]. Especially for many iterations implementation program, the small difference of floating-point will grow up very quickly. That's just as our case. However, we still don't know why double-precision floating-point will make our parallel program give the same result with sequential result.

Our results can be summarized as table 2 (we use C = 1, γ =0.001). CUDA1 means the first approach mentioned in section 3.2 which can't apply on 60,000 training set because of memory limitation. CUDA2 means our final version which parallel updates f_i using multiple threads with GPU.

Table 2: Comparison of sequential and CUDA results

#Support vectors:

	Serial	CUDA1	CUDA2
10,000 Training set	1366	1366	1366
60,000 Training set	17578	-	17578

b:

	Serial	CUDA1	CUDA2
10,000 Training set	0.27546	0.27546	0.27546
60,000 Training set	-7.17299	-	-7.17299

Test accuracy:

	Serial	CUDA1	CUDA2
10,000 Training set	0.982 1473/1500	0.982 1473/1500	0.982 1473/1500
60,000 Training set	0.9357 9357/10000	-	0.9357 9357/10000

Execution time (sec):

	Serial	CUDA2	Speedup
10,000 Training set	181.86	5.53	32.89
60,000 Training set	8868.78	160.89	55.12

6 RELATED WORK

The very first attempt of improving the performance by exploiting parallelism using cluster of CPUs with MPI is Cao et al. [4]. They have almost 100% efficiency on 2 and 4 CPUs on which we only have 70% efficiency. As we mentioned in section 3.1, we only parallel computes f_i while Cao et al. parallelizes all steps. Not far from Cao's work, Catanzaro et al. [6] proved the effectiveness of implementing the same modified SMO algorithm in CUDA but didn't implement regression. The main idea of parallelism is using map function to compute f_i and search for $b_{\text{up}}, I_{\text{up}}, b_{\text{low}}, I_{\text{low}}$ with reduction operation. Plus they compute rows of $K(x_i, x_j)$ on the fly, as needed by the algorithm, and cache them in the available memory on the GPU. This implementation can save time if the needed two rows of $K(x_i, x_j)$ are present in the cache. Later Carpenter [7] extended Catanzaro's work to SVM regression and gave speedup about 35 compared with libsvm [5]. In Carpenter's work, he used C = 10 which will consume more time than us (C = 1) because he has to find more support vectors. In his work he found 43,729 support vectors in 60,000 training set while we only find 17,578 support vectors. This didn't mean our result will give worse prediction because if there were too many outliers the more support vectors found will overfit the result.

7 CONCLUSIONS

In the report, we implement the modified SMO algorithm for training SVM on multiple processors and GPU respectively. We identify the most time consuming part of modified SMO and find out that part can be parallel computed. We test our implementation on MNIST handwritten digit dataset and discover that the results of sequential and parallel version are the same either with MPI or CUDA. The test accuracy is 0.9357 for classifying the digits as even or odd. And the speedup is 2.8 on 4 CPUs with MPI and 55.12 with GPU.

In the future, we will continue to parallelize the other part of modified SMO algorithm and sustain attention on SVM parallelism research. Also the multi-class SVM is the next goal to finish.

Appendix:

A Lagrange duality

In introduction we define that training SVM problem is solving $\min_{w,b,\xi} \frac{1}{2} w^T w$ (C=0) with some constraints.Since the traditional Lagrange method can only solve the equal sign in the constraints, we have to invoke Karush-Kuhn-Tucker (KKT) conditions to solve the optimization problem of Lagrange function L

subject to the equal and greater than sign equation, then the optimization problem becomes

$$L(w,b,\alpha) = \frac{1}{2}w^T w - \sum_{i=1}^{l} \alpha_i [y_i(w^T \Phi(x_i) + b) - 1]$$

where α_i are the Lagrange multipliers. And we know the optimal solution happen with $\nabla L = 0$. So for w, we have $\frac{\partial L(w,b,\alpha)}{\partial w} = 0$ which gives us

$$w = \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i)$$

For b, we have $\frac{\partial L(w,b,\alpha)}{\partial b} = 0$ which gives us

$$\sum_{i=1}^{l} \alpha_i y^{(i)} = 0$$

$\alpha_i \geq 0, i = 1, \dots, l$ is following the results of KKT method.

Finally, the new object function L can be turn to a function of α only by substituting the equations mentioned above:

$$\begin{aligned} & L(w,b,\alpha) \\ &= \frac{1}{2}w^T w - \sum_{i=1}^{l} \alpha_i [y_i(w^T \Phi(x_i) + b) - 1] \\ &= \frac{1}{2}w^T w - \sum_{i=1}^{l} \alpha_i y_i w^T \Phi(x_i) - b \sum_{i=1}^{l} y_i \alpha_i + \sum_{i=1}^{l} \alpha_i \\ &= \frac{1}{2}\sum_{i=1}^{l} \alpha_i y_i w^T x_i - \sum_{i=1}^{l} \alpha_i y_i w^T x_i + \sum_{i=1}^{l} \alpha_i \\ &= -\frac{1}{2}\sum_{i=1}^{l} \alpha_i y_i w^T x_i + \sum_{i=1}^{l} \alpha_i \\ &= \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{l} y^{(i)} y^{(j)} \alpha_i \alpha_j K(x_i, x_j) \end{aligned}$$

B Kernel $K(x_i, x_j)$

We have known that $K(x_i, x_j) \equiv \Phi(x_i)^T \Phi(x_j)$, where $\Phi(x)$ maps from the attributes to the features. It's counterintuitive that we usually know $K(x_i, x_j)$'s form while don't know $\Phi(x)$'s. For example, suppose $x, z \in R^n$, and consider

$$K(x,z) = (x^T z)^2$$

If we want to get $\Phi(x)$, we have to write $K(x,z)$ as

$$\begin{aligned} K(x,z) &= \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_j z_j\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j \\ &= \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j) \end{aligned}$$

Then for n = 3, we know $K(x,z) = \Phi(x)^T \Phi(z)$, so $\Phi(x)$ can be represented as

$$\Phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}$$

Obviously calculating the high-dimensional $\Phi(x)$ requires $O(n^2)$ time, finding $K(x,z)$ takes only $O(n)$ time which is linear in the dimension of the input attributes. Table 2 lists the most common kernel functions.

Table 2: Standard Kernel Functions

Linear	$K(x_i, x_j) = x_i \cdot x_j$
Polynomial	$K(x_i, x_j) = (a x_i \cdot x_j + b)^d$

Gaussian	$K(x_i, x_j) = exp(-\gamma\|x_i - x_j\|^2)$
Sigmoid	$K(x_i, x_j) = \tanh(ax_i \cdot x_j + b)$

C Guide to run our code

Our codes and needed data have been placed on github:
https://github.com/waitmeteng/parallel_svm.
Only training set of 10,000 examples and a test set of 1500 examples are there due to the limit of 100MB file size in github. Running our code on MNIST database comprises two steps. First, training support vector machine and second, testing the result of training model. At first stage you need one libsvm style-like input file to train SVM and five parameters including number of training set x_i, dimension n of x_i, regularization parameter C, γ for $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$ and tolerance parameter which we recommend to set it 0.001. To know how to choose these parameters wisely please refer to [5]. You also have to give the name of output file which will contain the support vectors and associated Lagrange parameters α_i. At second stage, the output file of first stage will become the input file here. You also have to give the file containing the test set. Then the test accuracy will be delivered.

REFERENCES

[1] http://cs229.stanford.edu/
CS229 Lecture notes part V: Support Vector Machines by Andrew Ng.

[2] J.C. Platt, "Fast training of support vector machines using sequential minimal optimization", in B. Scholkopf, C. Burges, A. Smola. *Advances in Kernel Methods: Support Vector Machines*, MIT Press, Cambridge, MA, December 1998.

[3] S.S. Keerthi, S.K. Shevade, C. Bhattaacharyyaand K.R.K. Murthy. "Improvements to Platt's SMO algorithm for SVM classifier design" Neural Computation, Vol. 13, pp. 637-649, 2001.

[4] L.J. Cao, et al. "Parallel sequential minimal optimization for the training of support vector machines". Neural Networks, IEEE Transactions on, 17(4):1039-1049, July 2006.

[5] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2(3):27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[6] B. Catanzaro, N. Sundaram, and K. Keutzer. "Fast support vector machine training and classification on graphics processors". In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 104-111, New York, NY, USA, 2008. ACM. Software available at http://www.cs.berkeley.edu/~catanzar/GPUSVM/.

[7] A. Carpenter. "CUSVM: A CUDA implementation of support vector classification and regression". Technical report (2009)

[8] http://yann.lecun.com/exdb/mnist/

[9] https://github.com/justdark/matlab_code-ufldl-exercise-/tree/master/softmax_exercise

[10] https://docs.nvidia.com/pdf/Floating_Point_on_NVIDIA_GPU.pdf