Double Chunking for Solving SVMs for Very Large Datasets

Fernando Pérez-Cruz[†], Aníbal R. Figueiras-Vidal[‡] and Antonio Artés-Rodríguez[‡]
Gatsby Computational Neuroscience Unit University, College London
17 Queen Square. London WC1N 3AR United Kingdom. fernando@gatsby.ucl.ac.uk
Signal Theory and Communications Department, University Carlos III Madrid
Avda. Universidad 30, 28911 Leganés (Madrid) Spain

ABSTRACT

In this paper we present a novel approach to solve the SVM for very large training sets. We propose to substitute the shrinking technique for a double chunk, so we are able to exploit the cache memory more efficiently and obtain the solution in considerably less runtime complexity.

1 INTRODUCTION

Support Vector Machines (SVMs) are state-of-the-art tools for solving input-output knowledge discovery problems (Schölkopf and Smola, 2001; Pérez-Cruz and Bousquet, 2004). The SVM for binary classification defines a margin (exclusion region) between both classes, which is maximized to ensure that the obtained classifier is able to generalise to previously unseen patterns. Given a labelled training dataset ($\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i = \{\pm 1\}$), the SVM computes its maximum margin solution via a constraint optimization problem:

$$\min_{\mathbf{w},\xi_i,b} \left\{ \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i \right\} \tag{1}$$

subject to:

$$y_i(\boldsymbol{\phi}^T(\mathbf{x}_i)\mathbf{w} + b) \geq 1 - \xi_i \tag{2}$$

$$\xi_i \geq 0 \tag{3}$$

where $\mathbf{w}$ and b define the linear classifier in the feature space (nonlinear in the input space, unless $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$) and C is the penalty applied over margin errors (samples that do not fulfil the margin requirement).

To solve this problem, we can either use Lagrange multipliers and construct the Wolfe's Dual (Fletcher, 1987), leading to a quadratic programming (QP) problem, or define a quadratic upper bound which is solved iteratively (Pérez-Cruz et al., 2004), known as the Iterative Re-Weighted Least Square (IRWLS) procedure. In both cases, the SVM solution is based on the Representer theorem (Kimeldorf and Wahba, 1971) which, under fairly general conditions, allows to represent $\mathbf{w}$ as a linear combination of the samples in the training set

$\mathbf{w} = \sum_{i=1}^{n} \alpha_i \mathbf{y}_i \boldsymbol{\phi}(\mathbf{x}_i)$. The use of the Representer theorem allows that the optimization functional be defined in terms of dot products of the samples in the feature space. Therefore we only need to use the kernel ($k(\cdot,\cdot) = \boldsymbol{\phi}^T(\cdot)\boldsymbol{\phi}(\cdot)$) of the transformation instead of the whole nonlinear mapping. Every sample is potentially part of the solution and both optimization procedures need to work with a kernel matrix that contains every pairs of samples ($n \times n$). So they can only be directly applied using a few hundred samples.

In (Osuna et al., 1997), the authors proposed to use a chunking mechanism to solve SVMs for large datasets. The chunking mechanism relies on the solution of the SVM being sparse (many of the α_i are zero) and also many samples present $\alpha_i = C$, those with $\xi_i > 0$, which are known as the bounded support vectors. Only the samples with $\xi_i = 0$ that hold (2) with equality, present an α_i between 0 and C, unbounded support vectors, and their value needs to be determined by the learning procedure. The chunking mechanism allows to solve the SVM with a few training samples (a chunk). Those samples whose α_i is either 0 or C can be left out and other training samples, which do not fulfil the constraints (2) and (3), are used for solving a next chunk. This procedure is repeated until the SVM solution has been reached. The convergence of this procedure has been shown in (Keerthi et al., 1999). This procedure has been efficiently implemented in SVM^{light} (Joachims, 1999), including a cache memory for the most widely used kernels and a shrinking technique, which reduces the training sample to an active set – those that are likely to change their α_i value. This chunking procedure can be solved using only 2 samples in each iteration, in which the QP formulation can be solved analytically. This procedure is known as Sequential Minimal Optimization (SMO) (Platt, 1999), in which it is based the current most powerful SVM solver: LIBSVM[1] (Chang et al., 2000). The LIBSVM is based on taking those samples that violate the constraints by most in each iteration and used it for the next chunk, until the SVM solution has been reached. It incorporates the cache memory and the shrinking mechanism described in (Joachims, 1999).

One of the limitations that chunking algorithms presents for very large datasets is that using only a few samples (most of the times only 2) makes the algorithm wander around until a good approximation to the whole solution is built and finally it will converge fast to the solution, thanks to the cache and shrinking. This wandering effect is due to many samples presenting similar output values and not being clear which one should be picked to obtain the largest improvement towards the SVM solution. Many iterations are wasted with little improvement or by undoing previous changes. Also, if the training set is very large, the cache memory will be emptied every few iterations, not gaining anything from it. The shrinking cannot reduce the active set as almost all the samples look potentially good candidates for entering the next chunk.

In this paper we proposed a double chunking technique for solving SVMs with very large datasets that will prevent this wandering around from happening. We know that if we have a good approximation to the solution, we will be able to use the chunking procedure much more efficiently to get to the final SVM solution. So instead of working with all the samples, we compute the solution using only several thousands. For this reduced dataset the chunking

[1] http://www.csie.ntu.edu.tw/∼cjlin/libsvm

will allow us to find a solution very fast. We will then use this partial result, which will give us a rough idea about the SVM solution, to compute the error of the remaining samples and choose the samples that must enter the next "large" chunk. We solve each "large" chunk with a chunking technique, giving name to the algorithm, and repeat this process until the SVM solution has been reached. The use of this double chunking allows us to be able to get faster to the SVM solution than with the standard chunking technique, making good use of the cache memory available and avoiding the wandering nature for very large datasets.

2 DOUBLE CHUNKING

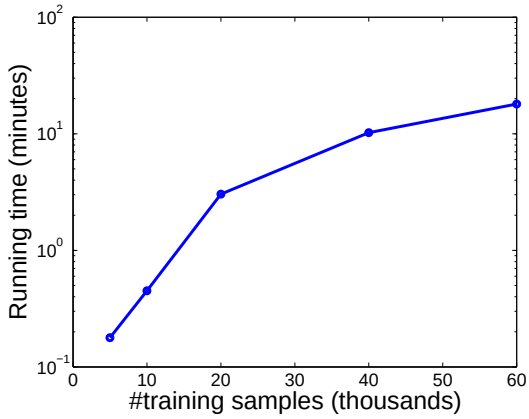
To solve the SVM we need first to compute the $n \times n$ kernel matrix. For problems with small number of samples this problem can be addressed directly. But for larger ones, we would need a different procedure, as we cannot store the kernel matrix into memory. In (Osuna et al., 1997) the authors proposed a chunking scheme in which they solved a sequence of QP problems similar to the SVM, which lead to the SVM solution with all the samples. The chunking procedure works as follows: first it chooses N samples, much less than the number of training examples (usually between 2-100), which can be solved using QP tools, and obtains the SVM solution. Then, it computes the error, $e_i = y_i(\boldsymbol{\phi}^T(\mathbf{x}_i)\mathbf{w}+b)$, for all the samples. Among the samples that do not fulfil the restrictions (2) and (3), choose a subset of size N and solve a modified SVM (the modification is given by the previous solution). Now repeat the procedure until all the samples fulfil the restrictions, ending up with the SVM solution.

In this procedure the computational burden is not in solving the QP problems, but computing the error for every sample, as we need to compute the kernel between all the samples and the support vectors in each iteration. So it is important that we choose well the samples that enter the SVM functional in each iteration to minimise the number of times we need to compute the kernels. This procedure works quite well when the number of training samples is relatively small, some thousands of samples, but when the number of samples gets larger the used heuristics to decide the N samples in each QP problem do not work so accurately and it loses many iterations trying to locate a good solution from which all the samples can be classified and the SVM solved. Moreover the number of rows of the kernel matrix that can be stored are very few, because we have many samples in each row. We will be constantly emptying it and we will not gain anything from it. To illustrate this point we have shown in Figure 1a the running time to solve the hand written digit recognition task (more information in the experimental section) with increasing number of samples, it can be seen that to solve the problem with 60000 samples we need 100 times more computational time than for solving it with a tenth of the training samples. This behaviour is standard for most datasets when solving the SVMs by chunks.

Now suppose that the solution with the first 5000 samples was already a good starting point for the whole SVM solution and from this solution the final SVM solution is more easily reached. Then, we could use this solution to initialized the chunking procedure and obtain the SVM solution in a few iterations.

Based on these ideas, we propose to solve the SVM with a double chunking procedure.

First we construct a "large" chunk (500-10000 samples) and solve the SVM as it were the only available samples. Problems of this size can not be dealt directly but they can be solved very efficiently using chunking procedures, as the ones previously described, with a "small" chunk (2-100 samples), giving the name to the algorithm. Once we obtain the solution for the "large chunk", we can classify the remaining samples and find out which ones do not fulfil the restrictions. As the solution of the first "large" chunk is a good approximation, we can easily pick those samples that provides the largest improvements and form the next "large" chunk. We show an algorithmic procedure in Figure 1b. The functions *selection*



(a)

```
do {
        do {
                form_H1_y1(it);
                irwls();
                error(it);
                it++;
        }while(selection());
} while(selection2());
```

(b)

Figure 1: We show in (a) the runtime to solve the SVM with increasing number of samples. In (b) the pseudocode of the double chuncking procedure is represented.

and *selection2* choose respectively the samples that form the "small" and "large" chunk in each iteration. *form_H1_y1* construct the kernel matrix that it is going to be used in the optimization step in *irwls*. Finally *error* computes the error of all the samples in the "large" chunk and maintains the cache memory. If we drop the first loop we are left with a standard chunking procedure without shrinking.

The double chunking procedure can be readily improved when computing the error for deciding which samples enter the "large" chunk (*selection2*), which is the most computational costly step as we will show in the experimental section, because we need to compute the kernel matrix between all the support vectors in the "large" chunk and all the training samples to obtain the error of each sample. We propose that instead of computing the error for all the samples and then decide which ones form the new "large" chunk, we compute the error sample by sample and those samples that do not fulfil the restrictions in (2) and (3) are introduced in the "large" chunk until we fill it up. This procedure avoids computing the error for many samples in the first iterations, significantly reducing the runtime complexity. One possible drawback is that we are not selecting the best set of samples in each iteration and that might force us to take more iterations that we need to. But, first of all, it is not clear that getting the samples that violates the restrictions by most will improve the solution by most and, if we do have to perform extra iteration/s, they are considerably cheap (in terms of computational time) compare to computing the error of all the samples after each

"large" chunk has been solved, so we will be actually gaining speed of convergence. Besides, this way of computing the error, has a second advantage. We will know if we have processed all the training samples (if we have computed their error), before solving the "large" chunk. In the case we have not, the resolution of the "large" chunk does not need to be obtained exactly, as we know that there will be extra iterations to come. So when we solve the SVM, if we have not evaluated all the samples yet, we will not enforce that every nonbounded support vector fulfils the restrictions, obtaining an approximate partial solution in significantly fewer iterations.

Finally to increase the robustness of the procedure, we do not only include the samples that do not fulfil the restrictions in each chunk, but we also include all the samples that are unbounded support vectors (they will change their weight in the resolution of the next "large" chunk) and some samples from the previous chunk, that present an $e_i \sim 0$ and are likely to change their α_i value. The introduction of these samples takes into consideration the previous solution and avoids that the new samples will reverse the previous solution.

3 Shrinking versus double chunk

Shrinking is one of the techniques that standard chunking programs use to reduce the runtime complexity and are effective for very large datasets. This technique tries to locate those samples that clearly fulfil their restrictions and eliminate them from the learning procedure, so it will be like working with fewer training samples. This allows to use the cache memory over the samples that might present an $e_i \sim 0$ and get faster to the SVM solution. When the SVM solution is reached these samples are revaluated to verify if they still fulfil their restrictions. This can be seen a pruning mechanism of the training examples. The double chunking that we presented in the previous section can be regarded as a growing technique, we start with a few samples and then we move on to evaluate more of the samples in the training set. The double chunk seeks the same purpose as the shrinking, as we do not want to evaluate the error of all the sample after each iteration over a "small" chunk.

We believe that the double chunking is more effective than the shrinking for very large dataset, because the shrinking process takes a long time before it can reduced the number of training samples. If the dataset is very large, the chunking procedure will wander around, the shrinking will not take place, and we will be emptying the cache memory constantly. While the double chunking procedure will only concentrate on a subset of the samples in each iteration and it will be able to exploit the cache memory much better.

4 Experiments

We have compared the double chunk implementation with the LIBSVM that it is the best SVM code nowadays. We have used the MNIST handwritten digit recognition because it is one of the standards datasets in the machine learning community and it is one of the hardest problems to solve with many samples. Because it presents thousands of samples as nonbounded support vectors, which are the ones that slow down SVM software most, because they need to be specified exactly.

We have implemented the double chunking procedure over the IRWLS procedure proposed in (Pérez-Cruz et al., 2000), which was the fastest chunking scheme of its time, and we have compared it with the LIBSVM procedure. The MNIST dataset has 60000 training samples in 784 dimensional space. We have selected a Gaussian RBF kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \exp(\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma^2)$ with $\sigma = \sqrt{d/24}$, being d the input space dimension, and have set $C = 5$ (this parameters have been found to give optimal results using cross validation techniques). We have used 50Mbytes for the cache memory with 4000 samples in the double chunk and 60% of them are from constraint breaking samples. We have used 40 samples in the IRWLS for the "small" chunk. The same stopping criteria was used: maximum constraint deviation 10^{-3}. We have reported the training time as we increase the number of samples in the task to compute the binary classifier of the zero labelling against the rest.

Number of Samples	5000	10000	20000	40000	60000
LIBSVM	0.17	0.45	2.32	9.97	17.95
double-chunk	0.22	0.50	1.81	6.66	11.63
Number of error	35	33	29	19	17

Table 1: We present the runtime complexity in minutes for the LIBSVM and the double chunking procedure with increasing number of training samples. In the last column we show the test error over 10000 examples

We can see that for large datasets (5000 and 10000 samples) the 2 procedures takes similar running time to converge to the SVM solution. For very large datasets > 20000, the double chunk is 40% better than the LIBSVM. This is explained because the double chunking avoids the wandering behaviour for the first iterations, because it is solving an SVM with 5000 samples. While the LIBSVM has a hard time guessing which are the best samples to be optimised in each iteration.

For solving the 60000 samples example, we needed 4 iterations over the "large" chunk of 4000 samples, and each one of them took: 3.91s, 13.95s, 29.5s and 19.8s. The remaining time was spent in *selection2* computing the error of the samples outside the "large" chunk in each iteration. It can be seen than most of the time was used to compute the error over the samples not in the actual optimization, as we commented in Section 2.

Positive Label	0	1	2	3	4	5	6	7	8	9
LIBSVM	18.0	11.1	31.3	37.3	24.6	32.9	19.3	24.1	43.8	37.6
Double-chunk	11.6	6.1	21.6	29.6	13.7	22.5	12.2	14.0	32.9	22.0
# SVs	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000

Table 2: We present the runtime complexity in minutes for the LIBSVM and the double chunking procedure with different labelling for the positive class. The last solumns shows the number of support vectors (SVs) and most of them are nonbounded.

We will now concentrate on the problem with 60000 samples. We have run experiments

using all other possible labelling: one versus all, two versus all and so on, using the same parameters as we did for the zero versus all. We have plotted the training time for both procedure in Table 2. In this table it can be seen that the results are consistently better for a wide range of different solutions, the number of support vectors varies significantly from one labelling to the other.

We now change some of the parameters to show that the double chunking is faster in most conditions. We have modified the cache memory from 10 to 120Mb. We have change the C parameter between 1 and 100 and σ between $\sqrt{d/8}$ and $\sqrt{d/64}$. The results have been plotted in Table 3.

Cache (MB)	10	20	50	80	120
LIBSVM	18.6	18.1	18.0	17.3	16.4
Double-chunk	15.2	12.4	11.6	12.0	9.1
C	1	2.5	5	25	100
LIBSVM	14.2	16.5	18.0	18.0	18.0
Double-chunk	9.6	19.8	11.6	11.5	11.5
σ	$\sqrt{d/8}$	$\sqrt{d/16}$	$\sqrt{d/24}$	$\sqrt{d/32}$	$\sqrt{d/64}$
LIBSVM	12.1	15.6	18.0	20.1	39.4
Double-chunk	6.1	10.2	11.6	13.1	45.5

Table 3: We present the runtime complexity in minutes for the LIBSVM and the double chunking procedure as a function of the cache memory and the SVM parameters.

Finally, we have computed the SVM solution using only the double chunking modifying some of its parameters to show that the runtime complexity its is robust to its setting, Table 3.

"large" chunk size	2000	4000	6000	8000
Double-chunk	14.6	11.6	9.2	10.4
"small" chunk size	20	40	70	100
Double-chunk	16.5	11.6	11.5	11.7
Fraction	0.2	0.4	0.6	0.8
Double-chunk	14.8	10.9	11.6	11.9

Table 4: We present the runtime complexity in minutes for the double chunking procedure with different number of samples in the "large" and "small" chunk and the fraction of samples that violates the constrain that enter in the "large" chunk in each iteration.

5 Conclusions and future work

In this paper we have presented a novel approach for dealing with very large datsets problems and get the SVM solution in a reduced runtime compare to the best chunking scheme. We exploit the fact that for very large datasets the cache can not store many rows of the kernel

matrix, so if we concentrate of a few samples (thousands) we can get an approximate solution and obtain the exact SVM solution in less runtime complexity.

This method can still be further improve, first of all it would be advisable that the "large" chunk will not be rigid and we could incorporate more or less samples to improve the runtime complexity and avoid extra iterations. This can be done monitoring the number of nonbounded support vectors, but it is an open issue how it can be done.

References

Chang, C.-C., Hsu, C.-W., and Lin, C.-J. (2000). The analysis of decomposition methods for support vector machines. *IEEE Transaction on Neural Networks*, 11(4).

Fletcher, R. (1987). *Practical Methods of Optimization*. John Wiley and Sons, Chichester, second edition.

Joachims, T. (1999). Making large scale SVM learning practical. In Schölkopf, B., Burges, C. J. C., and Smola, A. J., editors, *Advances in Kernel Methods— Support Vector Learning*, pages 169–184. M.I.T. Press, Cambridge, (MA).

Keerthi, S., Shevade, S., Bhattacharyya, C., and Murthy, K. (1999). Improvements to platt's SMO algorithm for SVM classifier design. Technical report, Dept of CSA, IISc, Bangalore, India. http://guppy.mpe.nus.edu.sg/~mpessk/smo_mod.ps.gz.

Kimeldorf, G. S. and Wahba, G. (1971). Some results in tchebycheffian spline functions. *Journal of Mathematical Analysis and Applications*, 33:82–95.

Osuna, E., Freund, R., and Girosi, F. (1997). An improved training algorithm for support vector machines. In Principe, J., Gile, L., Morgan, N., and Wilson, E., editors, *Neural Networks for Signal Processing VII — Proceedings of the 1997 IEEE Workshop*, pages 276 – 285, Amelia Island, New York. IEEE Press.

Pérez-Cruz, F., Alarcón-Diana, P. L., Navia-Vázquez, A., and Artés-Rodríguez, A. (2000). Fast training of support vector classifiers. In Leen, T. K., Dietterich, T. G., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13*, Cambridge, MA. M.I.T. Press.

Pérez-Cruz, F., Bousoño-Calzón, C., and guez, A. A.-R. (2004). Convergence of the irwls procedure to the support vector machine solution. *Neural Computation*. Accepted.

Pérez-Cruz, F. and Bousquet, O. (2004). Kernel methods and their potential use in signal processing. *Signal Processing Magazine*, 21(3):57–65.

Platt, J. C. (1999). Sequential minimal optimization: A fast algorithm for training suppor vector machines. In Schölkopf, B., Burges, C. J. C., and Smola, A. J., editors, *Advances in Kernel Methods— Support Vector Learning*, pages 185–208. M.I.T. Press, Cambridge, (MA).

Schölkopf, B. and Smola, A. (2001). *Learning with kernels*. M.I.T. Press.