University Library

A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Shilton, A;Palaniswami, M;Ralph, D;Tsoi, AC

Title:
Incremental training of support vector machines

Date:
2005-01-01

Citation:
Shilton, A., Palaniswami, M., Ralph, D. & Tsoi, A. C. (2005). Incremental training of support vector machines. IEEE Transactions on Neural Networks, 16 (1), pp.114-131. https://doi.org/10.1109/TNN.2004.836201.

Publication Status:
Published

Persistent Link:
https://hdl.handle.net/11343/34063

Incremental Training of Support Vector Machines

A. Shilton, M. Palaniswami, *Senior Member, IEEE,* D. Ralph, and A.C. Tsoi. *Senior Member, IEEE,*

Abstract— **We propose a new algorithm for the incremental training of Support Vector Machines (SVMs) that is suitable for problems of sequentially arriving data and fast constraint parameter variation. Our method involves using a "warm-start" algorithm for the training of Support Vector Machines (SVMs), which allows us to take advantage of the natural incremental properties of the standard active set approach to linearly constrained optimisation problems. Incremental training involves quickly re-training a support vector machine after adding a small number of additional training vectors to the training set of an existing (trained) support vector machine. Similarly, the problem of fast constraint parameter variation involves quickly re-training an existing support vector machine using the same training set but different constraint parameters. In both cases we demonstrate the computational superiority of incremental training over the usual batch re-training method.**

Index Terms— **Support Vector Machines, Incremental Training, Warm Start Algorithm, Active Set Method, Quadratic Programming.**

I. Introduction

BINARY pattern recognition involves constructing a decision rule to classify vectors into one of two classes based on a training set of vectors whose classification is known a-priori. Support vector machines (SVMs - Vapnik et al [1]) do this by implicitly mapping the training data into a higher-dimensional feature space. A hyperplane (decision surface) is then constructed in this feature space that bisects the two categories and maximises the margin of separation between itself and those points lying nearest to it (called the support vectors). This decision surface can then be used as a basis for classifying vectors of unknown classification.

The main advantages of the SVM approach are:

- SVMs implement a form of structural risk minimisation [1] - They attempt to find a compromise between the minimisation of empirical risk and the prevention of overfitting.
- The problem is a convex quadratic programming problem. So there are no non-global minima, and the problem is readily solvable using quadratic programming techniques.
- The resulting classifier can be specified completely in terms of its support vectors and kernel function type.

Usually, support vector machines are trained using a batch model. Under this model, all training data is given *a priori* and training is performed in one batch. If more training data is later obtained, or we wish to test different constraint parameters, the SVM must be re-trained from scratch. But if we are adding a small amount of data to a large training set, assuming that the problem is well posed, then it will likely have only a minimal effect on the decision surface. Re-solving the problem from scratch seems computationally wasteful.

An alternative is to "warm-start" the solution process by using the old solution as a starting point to find a new solution. This approach is at the heart of active set optimization methods [2], [3] and, in fact, incremental learning is a natural extension of these methods. While many papers have been published on SVM training, relatively few have considered the problem of incremental training. In our previous papers [4], [5] and [6], we outlined our approach to these problems, and also gave some preliminary results. Since the publication of our preliminary results, we have refined our algorithm substantially. In this paper we will present these refinements, and the algorithm as a whole, in some detail.

The SVM optimisation problem is a linearly constrained quadratic programming problem [2]. Therefore we are able to take advantage of the natural incremental properties of this method. One can draw parallels between the algorithm presented here and that of [7], [5]. However, we have entirely separated the issues of the underlying optimisation algorithm and the Hessian matrix factorisation used to realise it. As an adjunct to this, we have considered the issue of the choice of factorisation technique in some detail. We have also given some consideration to the problem of the occurrence of singular Hessian matrices, and how best to deal with this situation when selecting a factorisation for the Hessian matrix. Finally, we demonstrate the natural extension of our algorithm to the problem of constraint parameter variation and kernel selection.

Our investigation is broadly about applying the accumulated knowledge of optimization, e.g. [2], [3], to the computational problem presented by SVMs. Since our earlier work in [4] there have been several parallel contributions by other researchers [7], [8], [9]. The paper [8] is an active set approach to incremental SVMs that relates to the implementation we describe in Subsection IV-C though the former appears to be applied to the standard dual QP [1] rather than to the dual problem that we formulate, (4), as discussed in Section II. A fast interior-point approach to SVMs, though not incremental, appears in [9].

Our paper has been arranged as follows. In section II we give some necessary background material and present a number of fundamental results concerning support vector machines. In section III we present our method for solving the dual optimisation problem that arises from the support vector machine formulation given in section II. As is shown in this section, an important element of our method is the selection

A. Shilton and M. Palaniswami are with the Centre of Expertise on Networked Decision and Sensor Systems, Department of Electrical and Electronic Engineering, The University of Melbourne, Victoria 3010, Australia ({apsh,swami}@ee.mu.oz.au).

D. Ralph is with the Judge Institute of Management Studies, University of Cambridge, Trumpington street, Cambridge CB2 1AG, UK (danny.ralph@jims.cam.ac.uk).

A.C. Tsoi is with the University of Wollongong, Wollongong NSW 2522, Australia (act@uow.edu.au).

of an appropriate factorisation method for the Hessian matrix $\mathbf{H}_R$ (defined later) to facilitate the fast calculation of $\mathbf{r}$ where $\mathbf{s}$ is known and $\mathbf{H}_R\mathbf{r} = \mathbf{s}$. Therefore in section IV we consider the relevant issues pursuant to this selection and study two possible factorisations in detail. We conclude this section by studying the comparative theoretical merits of these factorisations.

In sections V and VI we show how the active set optimisation method given section III may be naturally applied to the problems of incremental learning and fast constraint parameter variation. Finally, in section VII, we give some experimental results obtained using the proposed incremental training and fast constraint parameter variation techniques, and compare these results with those obtained using standard non-incremental techniques.

II. Support Vector Machine Basics

We consider briefly how the SVM binary pattern recognition problem is formulated [10]. Firstly, we define the training set as:

$$\begin{aligned} &\Theta = \{(\mathbf{x}_1, d_1), (\mathbf{x}_2, d_2), \ldots, (\mathbf{x}_N, d_N)\} \\ &\mathbf{x}_i \in \Re^{d_L} \\ &d_i \in \{+1, -1\} \end{aligned}$$

We also (implicitly, as will be seen later) define a mapping from input space to a (usually higher dimensional) feature space, denoted $\boldsymbol{\varphi} : \Re^{d_L} \to \Re^{d_H}$. Assuming that the two training classes are linearly separable when mapped to feature space, we can define a linear discriminant function $g(\mathbf{x})$ in feature space such that:

$$\begin{aligned} &g(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\varphi}(\mathbf{x}) + b \\ &d_i = \operatorname{sgn}(g(\mathbf{x}_i)) \, \forall (\mathbf{x}_i, d_i) \in \Theta \end{aligned} \tag{1}$$

Any such discriminant function defines a linear decision surface in feature space that bisects the two training classes and is characterised by $g(\mathbf{x}) = 0$. However, there may be infinitely many such surfaces. To select the surface best suited to the task, the SVM maximises the distance between the decision surface and those training points lying closest to it (the support vectors). It is easy to show (see [11] for example) that maximising this distance is equivalent to solving:

$$\begin{aligned} &\min_{\mathbf{w}, b} \quad \tfrac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{such that} \quad d_i\left(\mathbf{w}^T\boldsymbol{\varphi}(\mathbf{x}_i) + b\right) \geq 1 \;\; \forall (\mathbf{x}_i, d_i) \in \Theta \end{aligned} \tag{2}$$

If the training classes are not linearly separable in feature space, we must relax the inequalities in (2) using slack variables and modify the cost function to penalise any failure to meet the original (strict) inequalities. The problem becomes [11]:

$$\begin{aligned} &\min_{\mathbf{w}, b, \boldsymbol{\xi}} \quad \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + C\mathbf{1}^T\boldsymbol{\xi} \\ &\text{s.t.} \quad d_i\left(\mathbf{w}^T\boldsymbol{\varphi}(\mathbf{x}_i) + b\right) \geq 1 - \xi_i \forall (\mathbf{x}_i, d_i) \in \Theta \\ &\qquad \boldsymbol{\xi} \geq \mathbf{0} \end{aligned} \tag{3}$$

where $\mathbf{1}$ is a column vector with all elements equal to 1. The constraint parameter C controls the trade-off between the dual objectives of maximising the margin of separation and minimising the misclassification error. We will now move to the dual form of the problem. This is done for two main reasons [10]:

- The constraints in the dual form of the problem are significantly simpler than those in the primal form.
- In the dual form the training data will appear only in the form of dot products. As will be shown shortly, this allows us to deal with very high (or even infinite) dimensional feature spaces in a trivial manner.

Let $\boldsymbol{\alpha}$ be the vector of Lagrange Multipliers associated with the first inequality in (3). After forming the Lagrangian (or Wolfe) dual problem and eliminating the primal vector $\mathbf{w}$, we are left with a quadratic program that still involves minimizing over the primal variable b as well as maximising over the dual variable $\boldsymbol{\alpha}$. Converting this to a minimization (maximization) with respect to $\boldsymbol{\alpha}$ (b) gives the following (partially-)dual form of (3):

$$\begin{aligned} &\max_{b} \min_{\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}} Q(\boldsymbol{\alpha}, b) \\ &Q(\alpha, b) = \tfrac{1}{2}\begin{bmatrix} b \\ \boldsymbol{\alpha} \end{bmatrix}^T \mathbf{H} \begin{bmatrix} b \\ \boldsymbol{\alpha} \end{bmatrix} - \begin{bmatrix} b \\ \boldsymbol{\alpha} \end{bmatrix}^T \begin{bmatrix} 0 \\ \mathbf{1} \end{bmatrix} \end{aligned} \tag{4}$$

where $Q(\boldsymbol{\alpha}, b)$ is the objective function and:

$$\begin{aligned} &\mathbf{H} = \begin{bmatrix} 0 & \mathbf{d}^T \\ \mathbf{d} & \mathbf{G} \end{bmatrix} \\ &\mathbf{G} \in \Re^{N \times N} \\ &G_{i,j} = d_i d_j K(\mathbf{x}_i, \mathbf{x}_j) \\ &K(\mathbf{x}_i, \mathbf{x}_j) = \boldsymbol{\varphi}(\mathbf{x}_i)^T \boldsymbol{\varphi}(\mathbf{x}_j) \end{aligned}$$

Note that the matrix $\mathbf{G}$ is positive semi-definite and the constraints are linear. Hence the optimisation problem is convex (there are no non-global minima), which greatly simplifies the process of finding a solution. It is also worth noting that there is exactly one variable α_i associated with each training pair $(\mathbf{x}_i, d_i)$. Furthermore, only those α_i's corresponding to support vectors will have non-zero values. Hence:

- The discriminant function can be fully specified using only the support vectors.
- If the SVM is re-trained using only the support vectors then the result will be the same as that obtained by training it with the complete training set.

Having obtained the optimal b and $\boldsymbol{\alpha}$, the discriminant function is easily defined, c.f. (1):

$$g(\mathbf{y}) = \sum_{(\mathbf{x}_i, d_i) \in \Theta} \alpha_i d_i K(\mathbf{x}_i, \mathbf{y}) + b$$

The function $K : \Re^{d_L} \times \Re^{d_L} \to \Re$ is called the kernel function. By identifying this kernel function, we may hide the dimensionality of feature space, enabling us to work with very high (or even infinite) values of d_H. Furthermore, so long as the kernel function satisfies Mercer's condition [10], we need never explicitly know what the feature mapping actually is.

The stationary or KKT (Karush-Kuhn-Tucker) conditions

for the program (4), see [2], are as follows:

$$\begin{array}{l} \boldsymbol{\alpha} \geq \mathbf{0} \\ \boldsymbol{\alpha} \leq C\mathbf{1} \\ f = 0 \\ e_i \left\{ \begin{array}{ll} \geq 0 & \text{if } \alpha_i = 0 \\ = 0 & \text{if } 0 < \alpha_i < C \\ \leq 0 & \text{if } \alpha_i = C \end{array} \right. \end{array} \tag{5}$$

where:

$$\left[\begin{array}{c} f \\ \mathbf{e} \end{array} \right] = \mathbf{H} \left[\begin{array}{c} b \\ \boldsymbol{\alpha} \end{array} \right] + \left[\begin{array}{c} 0 \\ \mathbf{1} \end{array} \right] \tag{6}$$

At this point it is appropriate to distinguish between the standard dual QP formulation of SVMs, as seen in [1] for example; and the formulation (5). The standard dual formulation is "fully dual" in the sense that the primal variable b is also eliminated by the inclusion of a single equality constraint $f = 0$, i.e. the constraints become exactly the first three lines of (5). This has the advantage of reducing (4) to a minimisation problem (rather than a max-min problem). It has the disadvantage of changing the feasible set from simple bounds $\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}$ to simple bounds plus an equality constraint. The key to the implementations proposed in this paper is to avoid the equality constraint, because its presence significantly complicates the description and implementation of the active set approach. The advantages of (4) have also been independently discovered in [7].

To give some hints about why a single equality constraint complicates notation and implementation, we mention our previous work [4], [5] and [6] in which an active set was applied to the standard dual QP, a minimization over $\boldsymbol{\alpha}$ with the additional constraint $f = 0$. Our implementation eliminated the equality constraint at each iteration by selecting a kind of "basis" variable whose value could be determined from this equation (this is an alternative to the textbook active set approach [2] which would perform a linear-algebraic reduction of the Hessian $\mathbf{H}$ by one dimension by working on the kernel or nullspace of the constraint $f = 0$. For another active set implementation see [8]). However, updating the choice of the basic variable to facilitate this elimination, and the associated bookkeeping needed to distinguish this variable from other variables, can be avoided when using the (partially-)dual QP (4) in which b is still considered to be a variable.

III. Method of Solution of Dual Problem

As noted previously in Section II, only those α_i's associated with support vectors will have non-zero values. An attractive feature of SVMs is that support vectors usually make up only a small fraction of the total training set (the ratio of support vectors to training set size may increase if the training set is noisy, but even in this case the ratio will tend to remain relatively small). Of all the methods of solving linearly constrained quadratic programming problems, active set methods [2] seem best suited to take advantage of this feature of SVMs. This is because, by using an active set method, they are able to reduce the effective dimensionality of the problem from the number of training points, which may be very large, to the number of support vectors (or some interim guess of the number of support vectors), which is typically small.

In an active set method, constraints are divided into two sets, namely the set of active constraints (the active set) and the set of inactive constraints. The algorithm then iteratively steps towards the solution, adjusting the active set after each step, until the optimal active set (and hence optimal solution) is arrived at. For any given iteration, the step is calculated by treating the active constraints as equality constraints, temporarily discarding the inactive constraints, and solving the resultant unconstrained optimisation problem (for a detailed introduction to active set methods, see for example [2]).

A. Notation

Before proceeding, it is necessary to introduce some notations. Given that only one of the upper or lower bound constraints may be active for any given α_i, the active set may be defined as follows:

$$\begin{array}{l} \boldsymbol{\alpha} = \left[\begin{array}{c} \boldsymbol{\alpha}_F \\ \boldsymbol{\alpha}_U \\ \boldsymbol{\alpha}_L \end{array} \right], \quad \mathbf{d} = \left[\begin{array}{c} \mathbf{d}_F \\ \mathbf{d}_U \\ \mathbf{d}_L \end{array} \right], \quad \text{etc.} \\ \mathbf{G} = \left[\begin{array}{ccc} \mathbf{G}_F & \mathbf{G}_{UF}^T & \mathbf{G}_{LF}^T \\ \mathbf{G}_{UF} & \mathbf{G}_U & \mathbf{G}_{LU}^T \\ \mathbf{G}_{LF} & \mathbf{G}_{LU} & \mathbf{G}_L \end{array} \right] \\ \mathbf{H}_F = \left[\begin{array}{cc} 0 & \mathbf{d}_F^T \\ \mathbf{d}_F & \mathbf{G}_F \end{array} \right] \\ \Theta = \Theta_F \cup \Theta_U \cup \Theta_L \end{array} \tag{7}$$

where:

- $\boldsymbol{\alpha}_F \in \Re^{N_F}$ are Free variables (not actively constrained).
- $\boldsymbol{\alpha}_U \in \Re^{N_U}$ are actively constrained at Upper bound C.
- $\boldsymbol{\alpha}_L \in \Re^{N_L}$ are actively constrained at Lower bound 0.

We need the following technical result:

Theorem 1: If $N_F = 1$ then $\mathbf{H}_F$ is non-singular.

Proof: Given that $d_i = \pm 1$ for all i, it follows from the definition of $\mathbf{H}_F$ that if $N_F = 1$, $\det(\mathbf{H}_F) = -1$. Therefore $\mathbf{H}_F$ is non-singular. ■

If $\mathbf{H}_F$ is singular and $N_F > 1$ then it follows from theorem 1 that we may partition the free variables (without re-ordering) as follows:

$$\begin{array}{l} \boldsymbol{\alpha}_F = \left[\begin{array}{c} \boldsymbol{\alpha}_{FN} \\ \alpha_{FB} \\ \boldsymbol{\alpha}_{FS} \end{array} \right], \quad \text{etc.} \\ \mathbf{G}_F = \left[\begin{array}{ccc} \mathbf{G}_{FN} & \mathbf{g}_{FBN} & \mathbf{G}_{FSN}^T \\ \mathbf{g}_{FBN}^T & g_{FB} & \mathbf{g}_{FSB}^T \\ \mathbf{G}_{FSN} & \mathbf{g}_{FSB} & \mathbf{G}_{FS} \end{array} \right] \end{array}$$

where:

- $\mathbf{H}_{FN} = \left[\begin{array}{cc} 0 & \mathbf{d}_{FN}^T \\ \mathbf{d}_{FN} & \mathbf{G}_{FN} \end{array} \right]$ is non-singular.
- $\mathbf{H}_{FNB} = \left[\begin{array}{ccc} 0 & \mathbf{d}_{FN}^T & d_{FB} \\ \mathbf{d}_{FN} & \mathbf{G}_{FN} & \mathbf{g}_{FBN} \\ d_{FB} & \mathbf{g}_{FBN}^T & g_{FB} \end{array} \right]$ is singular.
- $\boldsymbol{\alpha}_{FN} \in \Re^{N_{FN}}$ $(1 \leq N_{FN} \leq N_F - 1)$ are those free variables corresponding to the Non-singular sub-Hessian $\mathbf{H}_{FN}$.
- $\alpha_{FB} \in \Re$ may be thought of as lying on the Boundary of the non-singular sub-Hessian $\mathbf{H}_{FN}$.

- $\boldsymbol{\alpha}_{FS} \in \Re^{N_F - N_{FN} - 1}$.

If $N_F = 0$ we define $\mathbf{H}_{FN}$ to be an empty matrix. We also define:

$$\mathbf{H}_R = \begin{cases} \mathbf{H}_F & \text{if } \mathbf{H}_F \text{ is non-singular.} \\ \mathbf{H}_{FN} & \text{otherwise.} \end{cases}$$

as a shorthand for the *relevant* part of the Hessian. $\mathbf{G}_R \in \Re^{N_R \times N_R}$, $\mathbf{d}_R \in \Re^{N_R}$ etc. are defined analogously, where $N_R = N_F$ if $\mathbf{H}_F$ is non-singular, $N_R = N_{FN}$ otherwise.

For (nearly) every iteration of our algorithm, it will be necessary to solve $\mathbf{H}_R \mathbf{r} = \mathbf{s}$ for $\mathbf{r}$, where $\mathbf{H}_R$ and $\mathbf{s}$ are provided. However, actually calculating the inverse of $\mathbf{H}_R$ is too computationally expensive to be feasible. To overcome this problem, we have chosen to calculate (and maintain throughout the algorithm) a factorisation, $\beth$, of $\mathbf{H_R}$. In principle any number of factorisations could be used, the only criterion being the ability to quickly solve $\mathbf{H}_R \mathbf{r} = \mathbf{s}$ for $\mathbf{r}$ using $\beth$ and also that the factorisation itself may be quickly modified to reflect changes in $\mathbf{H}_R$. In the present paper we have looked at two such factorisations, namely the inverse and Cholesky factorisations.

A superscript (k) will be used when necessary to indicate which iteration is being referred to, and a Δ prefix to indicate a change in the value of a variable between iterations. So, for example, $\Delta \mathbf{e}^{(k)} = \mathbf{e}^{(k+1)} - \mathbf{e}^{(k)}$ is the change in $\mathbf{e}$ between iterations k and $k+1$. A superscript $*$ will be used to indicate the optimal solution to (4) if the active constraints are treated as equality constraints and the inactive constraints are ignored. Formally, $(\boldsymbol{\alpha}_F^*, b^*)$ is the solution of the equality-constrained quadratic program (15), to appear in subsection III-D of this section.

We define the following proceedures for use when modifying the active set:

$$\begin{aligned} \mathrm{constrain}\,(i, X; k): \quad & \Theta_F^{(k+1)} = \Theta_F^{(k)} \setminus \{(\mathbf{x}_i, d_i)\}. \\ & \Theta_X^{(k+1)} = \Theta_X^{(k)} \cup \{(\mathbf{x}_i, d_i)\}. \\ \mathrm{free}\,(i, X; k): \quad & \Theta_F^{(k+1)} = \Theta_F^{(k)} \cup \{(\mathbf{x}_i, d_i)\}. \\ & \Theta_X^{(k+1)} = \Theta_X^{(k)} \setminus \{(\mathbf{x}_i, d_i)\}. \end{aligned}$$

where $X \in \{L, U\}$. In either case, $\boldsymbol{\alpha}$, $\mathbf{G}$, $\beth$ etc. must be modified accordingly. Specifically, whenever data is added to any of Θ_F, Θ_U, Θ_L it is placed at the *end* of the vectors/matrices in question (the ordering of points already in the vector/matrix does not change). When removing data from a set, the ordering on either side of the data point in question remains unchanged.

Finally, it will be necessary at times to modify the ordering of the components in $\boldsymbol{\alpha}$, $\mathbf{G}$, $\beth$ etc. without modifying our active constraint set. Hence we define the function $\mathrm{swap}\,(i, j, X; k)$ to mean to operation of swapping components i and j in $\boldsymbol{\alpha}$, $\mathbf{G}$ etc., where $(\mathbf{x}_i, d_i) \in \Theta_X$ and $(\mathbf{x}_j, d_j) \in \Theta_X$.

B. Overall Structure of the Active Set Algorithm

The basic structure of our algorithm is shown in figure 1. In future, the algorithm in figure 1 will be referred to simply as the algorithm. Structurally, the algorithm is typical of active set methods. A Newton step (or some other step if $\mathbf{H}_F$ is singular) for the free variables is calculated. This step is then scaled ($\beta^{(k)}$ is the scale factor), and the scaled step is taken. If the KKT conditions are not met after this, the active set is modified in a minimal fashion (a single constraint is activated or deactivated) and the process is repeated. Otherwise, the algorithm terminates, the optimal solution being:

$$\begin{bmatrix} b^* \\ \boldsymbol{\alpha}^* \end{bmatrix} = \begin{bmatrix} b^{(k+1)} \\ \boldsymbol{\alpha}^{(k+1)} \end{bmatrix}$$

Notes on the algorithm:

- The relevant section number for each block is given beside that block.
- The default active set upon entering the algorithm (if none has already been defined) is for all training points to be constrained at lower bound 0.
- $\left(\Delta \mathbf{e}_U^{(k)}, \Delta \mathbf{e}_L^{(k)}\right)$ is always calculated using the formula:

$$\begin{bmatrix} \Delta \mathbf{e}_U^{(k)} \\ \Delta \mathbf{e}_L^{(k)} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_U & \mathbf{G}_{UF} \\ \mathbf{d}_L & \mathbf{G}_{LF} \end{bmatrix} \begin{bmatrix} \Delta b^{(k)} \\ \Delta \boldsymbol{\alpha}_F^{(k)} \end{bmatrix} \tag{8}$$

- The largest possible scaling factor $\beta^{(k)}$ (which we use) satisfying $\mathbf{0} \leq \boldsymbol{\alpha}_F^{(k)} + \beta^{(k)} \Delta \boldsymbol{\alpha}_F^{(k)} \leq C\mathbf{1}$ is:

$$\beta^{(k)} = \min \left(1, \min_{i:\Delta\alpha_i^{(k)} < 0} \frac{-\alpha_i^{(k)}}{\Delta\alpha_i^{(k)}}, \min_{i:\Delta\alpha_i^{(k)} > 0} \frac{C - \alpha_i^{(k)}}{\Delta\alpha_i^{(k)}} \right) \tag{9}$$

- As will be shown in section III-E, it is sufficient, when checking the KKT conditions (5), to check:

$$\begin{aligned} & \mathbf{e}_U^{(k+1)} < \varepsilon \mathbf{1} \\ & \mathbf{e}_L^{(k+1)} > -\varepsilon \mathbf{1} \\ & -\varepsilon < f^{(k+1)} < \varepsilon \end{aligned} \tag{10}$$

  where the constant $0 < \varepsilon \ll 1$ is necessary to prevent cycling due to cumulative numerical errors. Typically, we found $\varepsilon \simeq 10^{-6}$ was sufficient for the purpose.

C. Modifying the Active Set

In the algorithm, modification of the active set always takes the form of activating or deactivating a single constraint. For simplicity, we have chosen to follow the heuristic of [2], namely:

1: If the most recent step was scaled (ie. $\beta^{(k)} < 1$) then $\mathrm{constrain}\left(p^{(k)}, X; k\right)$, where:

$$p^{(k)} = \arg\min_i \left(\min_{i:\Delta\alpha_i^{(k)} < 0} \frac{-\alpha_i^{(k)}}{\Delta\alpha_i^{(k)}}, \min_{i:\Delta\alpha_i^{(k)} > 0} \frac{C - \alpha_i^{(k)}}{\Delta\alpha_i^{(k)}} \right) \tag{11}$$

and $X = L$ if $\Delta\alpha_i^{(k)} < 0$, $X = U$ if $\Delta\alpha_i^{(k)} > 0$. Thus we constrain whichever element $\alpha_i^{(k)}$ has "run into" a boundary.

2: Otherwise (if $\beta^{(k)} = 1$), find the element $e_{m^{(k)}}^{(k+1)}$ of $\mathbf{e}^{(k+1)}$ corresponding to an actively constrained $\alpha_{m^{(k)}}^{(k+1)}$ that most violates the simplified KKT conditions (10) (according to the criteria detailed below) and $\mathrm{free}\left(m^{(k)}, X; k\right)$.

In case 2, if $N_F^{(k)} > 0$ we select $m^{(k)}$ by finding the element $e_{m^{(k)}}^{(k+1)}$ that violates the KKT condition associated with it and

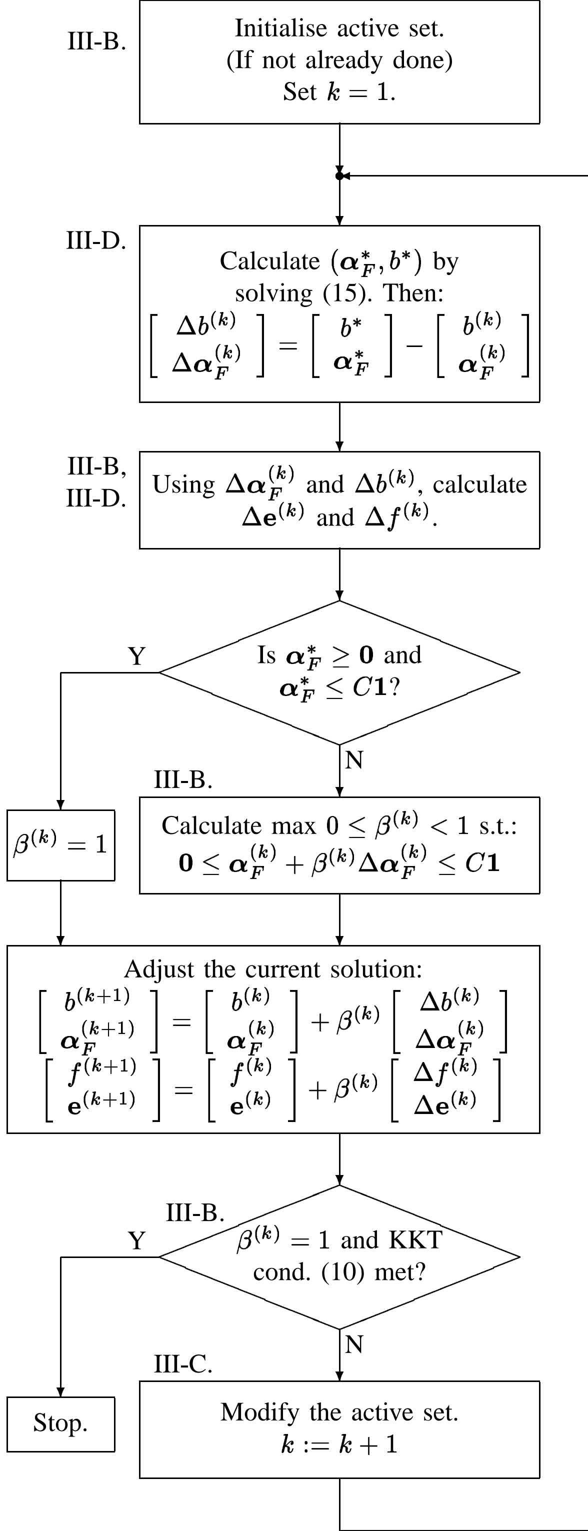


Fig. 1. Outline of active set algorithm.

is largest in magnitude. This is equivalent to using the simple rule:

$$m^{(k)} = \arg\min_i \left(\min_{i:(\mathbf{x}_i,d_i)\in\Theta_U^{(k)}} -e_i^{(k+1)}, \min_{i:(\mathbf{x}_i,d_i)\in\Theta_L^{(k)}} e_i^{(k+1)} \right) \tag{12}$$

where $(\mathbf{x}_i, d_i) \in \Theta_X^{(k)}$. It is not necessary to check that $e_{m^{(k)}}^{(k+1)} \geq \varepsilon$ if $(\mathbf{x}_{m^{(k)}}, d_{m^{(k)}}) \in \Theta_U^{(k)}$ and $e_{m^{(k)}}^{(k+1)} \leq -\varepsilon$ if $(\mathbf{x}_{m^{(k)}}, d_{m^{(k)}}) \in \Theta_L^{(k)}$, because, as will be shown later, if $N_F^{(k)} > 0$ and $\beta^{(k)} = 1$, it follows that $f^{(k+1)} = 0$, and hence either $\mathbf{e}_U^{(k+1)} \not< \varepsilon\mathbf{1}$ or $\mathbf{e}_L^{(k+1)} \not> -\varepsilon\mathbf{1}$, which, by virtue of the minima used in (12), is sufficient to ensure that $e_{m^{(k)}}^{(k+1)} \geq \varepsilon$ if $(\mathbf{x}_{m^{(k)}}, d_{m^{(k)}}) \in \Theta_U^{(k)}$ and $e_{m^{(k)}}^{(k+1)} \leq -\varepsilon$ if $(\mathbf{x}_{m^{(k)}}, d_{m^{(k)}}) \in \Theta_L^{(k)}$.

Continuing case 2, if $N_F^{(k)} = 0$ then there is no guarantee that $f^{(k+1)} = 0$. Because of this, care must be taken to ensure that the algorithm does not enter an endless loop. To see why this may occur, consider the explicit form of the next step (indexed by the positions of elements during the present step), from equation (16):

$$\begin{bmatrix} \Delta b^{(k+1)} \\ \Delta\alpha_{m^{(k)}}^{(k+1)} \end{bmatrix} = \begin{bmatrix} G_{m^{(k)},m^{(k)}} f^{(k+1)} - d_{m^{(k)}} e_{m^{(k)}}^{(k+1)} \\ -d_{m^{(k)}} f^{(k+1)} \end{bmatrix} \tag{13}$$

If $\alpha_{m^{(k)}}^{(k+1)} = 0$ and $d_{m^{(k)}} f^{(k+1)} > 0$ (or $\alpha_{m^{(k)}}^{(k+1)} = C$ and $d_{m^{(k)}} f^{(k+1)} < 0$) then the magnitude of $f^{(k+1)}$ will not decrease in the next iteration (as $\beta^{(k+1)} = 0$), $\alpha_{m^{(k)}}^{(k+1)}$ will be immediately re-constrained, and an infinite loop will result. To avoid this problem, we use the following (slightly modified) definition of $m^{(k)}$ if $N_F^{(k)} = 0$:

$$m^{(k)} = \arg\min_i \left(\min_{\substack{i:(\mathbf{x}_i,d_i)\in\Theta_U^{(k)} \\ d_i f^{(k+1)} \geq 0}} -e_i^{(k+1)}, \min_{\substack{i:(\mathbf{x}_i,d_i)\in\Theta_L^{(k)} \\ d_i f^{(k+1)} \leq 0}} e_i^{(k+1)} \right) \tag{14}$$

where the additional constraint on $\operatorname{sgn}\left(d_i f^{(k+1)}\right)$ ensures that a loop condition cannot occur. Note that equations (12) and (14) are equivalent if $f^{(k+1)} = 0$. When proving the convergence of the algorithm (section III-E), we will need the following technical results:

Theorem 2: If $N_F^{(k)} = 0$ then the solution to equation (14) is well defined.

Proof: First, suppose that $f^{(k+1)} = 0$. In this case, the solution to equation (14) must be well defined as $\Theta^{(k)} = \Theta_U^{(k)} \cup \Theta_L^{(k)} \neq \emptyset$.

Otherwise, suppose that the theorem is not correct - ie. the solution to (14) is not well defined. Then $\mathbf{d}_L f^{(k)} > 0$ and $\mathbf{d}_U f^{(k)} < 0$. As $f^{(k+1)} \neq 0$, we know that $N_U^{(k)} > 0$. But this implies that $\operatorname{sgn}\left(f^{(k+1)}\right) = -\operatorname{sgn}\left(f^{(k+1)}\right)$ (as $\operatorname{sgn}\left(f^{(k+1)}\right) = \operatorname{sgn}\left(C\mathbf{d}_U^T\mathbf{1}\right)$), which is a not possible unless $f^{(k+1)} = 0$. This is a contradiction. Hence the theorem must be true.

■

Theorem 3: In the algorithm, if $N_F^{(k)} = 0$ then $\left|f^{(k+2)} - f^{(k)}\right| = \min\left(\left|f^{(k)}\right|, C\right)$ and $\left|f^{(k+2)}\right| \leq \left|f^{(k)}\right|$.

Proof: First, suppose $f^{(k)} = 0$. As $\Delta f^{(k)} = \mathbf{d}^T \Delta\boldsymbol{\alpha}^{(k)}$, it follows that $\Delta f^{(k)} = 0$, $f^{(k+1)} = 0$, and, from equation (13), $f^{(k+2)} = 0$. Hence $\left|f^{(k+2)} - f^{(k)}\right| = \min\left(\left|f^{(k)}\right|, C\right)$ and $\left|f^{(k+2)}\right| \leq \left|f^{(k)}\right|$, as required.

Now suppose $f^{(k)} \neq 0$. Following the same reasoning as for the previous case, $\Delta f^{(k)} = 0$. We see from (13) that $\Delta\boldsymbol{\alpha}_F^{(k+1)} = -\mathbf{d}_F f^{(k)}$, which implies that $\Delta f^{(k+1)} = -f^{(k)}$. We know that $0 < \beta^{(k+1)} \leq 1$, so $\left|f^{(k+2)}\right| \leq \left|f^{(k)}\right|$. Also, as $\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}$ and $\alpha_{m^{(k)}}^{(k+1)} \in \{0, C\}$, it follows from (9) that $\left|f^{(k+2)} - f^{(k)}\right| = \min\left(\left|f^{(k)}\right|, C\right)$. ■

Considering the additional constraint on $\operatorname{sgn}\left(d_i f^{(k+1)}\right)$ required in (14) to prevent a potential infinite loop, it is reasonable to ask whether a similar constraint should be included in (12) in order to avoid the occurrence of a similar infinite loop condition. A full explanation of why such a constraint is unnecessary has been delayed until section III-E but, roughly speaking it follows from theorem 4, which implies that equation (12) will never be required unless $f^{(k+1)} = 0$, rendering any such constraint irrelevant.

D. Calculating the Step

Treating the active constraints as equality constraints, and ignoring the inactive constraints, the optimisation problem (4) reduces to the following unconstrained quadratic programming problem:

$$\begin{aligned} &\max_{b} \min_{\boldsymbol{\alpha}_F} Q_F\left(\boldsymbol{\alpha}_F, b\right) \\ &Q_F\left(\boldsymbol{\alpha}_F, b\right) = \begin{cases} Q_{\mathbf{F}}\left(\boldsymbol{\alpha}_F, b\right) & \text{if} \quad N_F^{(k)} > 0 \\ bf & \text{if} \quad N_F^{(k)} = 0 \end{cases} \\ &Q_{\mathbf{F}}\left(\boldsymbol{\alpha}_F, b\right) = \frac{1}{2} \begin{bmatrix} b \\ \boldsymbol{\alpha}_F \end{bmatrix}^T \mathbf{H}_F \begin{bmatrix} b \\ \boldsymbol{\alpha}_F \end{bmatrix} - \begin{bmatrix} b \\ \boldsymbol{\alpha}_F \end{bmatrix}^T \begin{bmatrix} C\mathbf{d}_U^T \mathbf{1} \\ C\mathbf{G}_{FU}\mathbf{1} - \mathbf{1} \end{bmatrix} \end{aligned} \tag{15}$$

The solution to (15) (assuming that it is well defined) is denoted $\left(\boldsymbol{\alpha}_F^*, b^*\right)$. We aim, if possible, to calculate the step $\left(\Delta\boldsymbol{\alpha}_F^{(k)}, \Delta b^{(k)}\right) = \left(\boldsymbol{\alpha}_F^*, b^*\right) - \left(\boldsymbol{\alpha}_F^{(k)}, b^{(k)}\right)$. If $\mathbf{H}_F$ is non-singular (which implies $N_F^{(k)} > 0$), then:

$$\begin{bmatrix} \Delta b^{(k)} \\ \Delta\boldsymbol{\alpha}_F^{(k)} \end{bmatrix} = -\mathbf{H}_F^{-1} \begin{bmatrix} f^{(k)} \\ \mathbf{e}_F^{(k)} \end{bmatrix} \tag{16}$$

where the matrix inversion is avoided by using our factorisation ⅃, as described in section IV. It follows that:

$$\begin{bmatrix} \Delta f^{(k)} \\ \Delta\mathbf{e}_F^{(k)} \end{bmatrix} = -\begin{bmatrix} f^{(k)} \\ \mathbf{e}_F^{(k)} \end{bmatrix} \tag{17}$$

If the Hessian is singular then either $N_F^{(k)} = 0$ or $N_F^{(k)} \geq 2$ (it follows from theorem 1 that $N_F^{(k)} \neq 1$ here). In either case, it is clear that either the current active set is not optimal or there exists an alternative optimal active set with fewer free variables than the present active set. If $N_F^{(k)} = 0$, noting that the quadratic programming problem (15) is an unconstrained linear programming problem, we have chosen not to take a step, so:

$$\begin{aligned} \Delta b^{(k)} &= 0 \\ \Delta f^{(k)} &= 0 \end{aligned}$$

If the Hessian is singular and $N_F^{(k)} \geq 2$, we want the step to be in a direction of linear non-ascent with respect to $\boldsymbol{\alpha}_F$ and linear non-descent with respect to b. We also want the step to be large enough to lead to the activation of a constraint (ie. $\beta^{(k)} < 1$), thereby preventing termination of the algorithm in a non-optimal state. Consider the step:

$$\begin{aligned} \begin{bmatrix} \Delta b^{(k)} \\ \Delta\boldsymbol{\alpha}_{FN}^{(k)} \end{bmatrix} &= -\theta \mathbf{H}_{FN}^{-1} \begin{bmatrix} d_{FB} \\ \mathbf{g}_{FBN} \end{bmatrix} = -\theta \begin{bmatrix} s_b^{(k)} \\ \mathbf{s}_{\boldsymbol{\alpha}}^{(k)} \end{bmatrix} \\ \begin{bmatrix} \Delta\alpha_{FB}^{(k)} \\ \Delta\boldsymbol{\alpha}_{FS}^{(k)} \end{bmatrix} &= \begin{bmatrix} \theta \\ \mathbf{0} \end{bmatrix} \end{aligned} \tag{18}$$

where we once again avoid the matrix inversion by utilizing our factorisation ⅃.

Consequently:

$$\begin{aligned} \begin{bmatrix} \Delta f^{(k)} \\ \Delta\mathbf{e}_{FN}^{(k)} \end{bmatrix} &= \begin{bmatrix} 0 \\ \mathbf{0} \end{bmatrix} \\ \begin{bmatrix} \Delta e_{FB}^{(k)} \\ \Delta\mathbf{e}_{FS}^{(k)} \end{bmatrix} &= \begin{bmatrix} 0 \\ \mathbf{d}_{FS}\Delta b^{(k)} + \mathbf{G}_{FSN}\Delta\boldsymbol{\alpha}_{FN}^{(k)} + \mathbf{g}_{FSB}^{(k)}\Delta\alpha_{FB}^{(k)} \end{bmatrix} \end{aligned} \tag{19}$$

It is easy to see that this step is in a direction of linear descent/ascent with respect to both $\boldsymbol{\alpha}_F$ and b. When selecting θ, we want to ensure that the magnitude of the step is sufficiently large to ensure that the $\beta^{(k)} < 1$, and furthermore that the direction is one of linear non-ascent with respect to $\boldsymbol{\alpha}_F$ and linear non-descent with respect to b. Unfortunately, it is not in general possible to satisfy both of the constraints on the direction of the step. Because of this, we choose to ignore the requirement of linear non-descent with respect to b if $f^{(k)} \neq 0$ (if $f^{(k)} = 0$ then this requirement will be met regardless of our choice of θ). This makes implementing the algorithm simpler and, as will be shown in section III-E, does not affect the convergence of the algorithm (whereas choosing to ignore the requirement of non-ascent with respect to $\boldsymbol{\alpha}_F$ would lead to problems of cycling when $f^{(k)} = 0$ and subsequent failure of the algorithm to convergence).

It is not difficult to see that the following simple definition of θ will satisfy the requirement of linear non-ascent with respect to $\boldsymbol{\alpha}_F$:

$$\theta = \begin{cases} -\left(1 + \nu\right)C & \text{if } e_{FS}^{(k)} \geq \mathbf{s}_{\boldsymbol{\alpha}}^{(k)T} \mathbf{e}_{FN}^{(k)} \\ \left(1 + \nu\right)C & \text{if } e_{FS}^{(k)} < \mathbf{s}_{\boldsymbol{\alpha}}^{(k)T} \mathbf{e}_{FN}^{(k)} \end{cases} \tag{20}$$

In this equation, $\varepsilon < \nu < 1$ is a small positive constant.

E. Properties of the Algorithm

In this section we prove that the algorithm converges. The proof is split into two parts. First we prove that after a finite number of iterations of the algorithm certain conditions must be met. Having arrived at a point where these conditions are

satisfied, we are then able to directly apply a well-known convergence result, thus preventing unnecessary repetition of results readily available in the literature. In particular, reference will be made to proofs contained in [2] and [3].

Before proceeding to the major result for this section, we need the following preliminary theorem:

Theorem 4: If $f^{(1)} \neq 0$ then after some $l \leq 2N + N_F^{(1)} + 2$ iterations of the algorithm, $f^{(l)} = 0$. Furthermore, for all subsequent iterations $l+n$, $n \geq 0$, $f^{(l+n)} = 0$.

Proof: First, assume that $N_F^{(1)} > 0$. For each subsequent iteration k with $N_F^{(k)} > 0$ either a constraint will be activated ($N_F^{(k+1)} < N_F^{(k)}$); or $\beta^{(k)} = 1$, implying that the most recent step was calculated using (16) and hence that $f^{(k+1)} = 0$. As N_F is bounded and decreasing, after at most $N_F^{(1)}$ iterations either $f^{(k+1)} = 0$ or $N_F^{(k+1)} = 0$, but not both.

Consider the latter possibility. By theorem 3, $\left|f^{(k+2)} - f^{(k)}\right| = \min\left(\left|f^{(k)}\right|, C\right)$ and $\left|f^{(k+2)}\right| \leq \left|f^{(k)}\right|$. Given that $\boldsymbol{\alpha} \leq C\mathbf{1}$, (6) implies that $f \leq NC$. So, starting from the first iteration where $N_F = 0$, f will become zero after at most $m \leq 2N$ iterations. So, if $f^{(1)} \neq 0$, after some $l \leq 2N + N_F^{(1)} + 2$ iterations of the algorithm, $f^{(l)} = 0$. This proves the first part of the theorem.

Consider all possible methods of calculating subsequent steps, namely (16) and (18). In either case, $f^{(k)} = 0$ implies $\Delta f^{(k)} = 0$, proving the second part of the theorem. ■

Theorem 5: Given any starting point $\boldsymbol{\alpha}, b$ satisfying $\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}$, the algorithm is guaranteed to find an optimal solution to (4) in a finite time, where an optimal solution is one satisfying (5).

Proof: There are two main steps to this proof:

1. Show that the algorithm will not terminate until an optimal solution has been found.
2. Show that an optimal solution will be found in a finite time.

Consider item 1. From (10) it is clear that the algorithm will not terminate unless $f^{(k+1)} = 0$, $\mathbf{e}_U^{(k+1)} \leq \mathbf{0}$ and $\mathbf{e}_L^{(k+1)} \geq \mathbf{0}$ (to within precision ε). Furthermore, $\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}$ throughout the algorithm. All that remains to be shown is that the algorithm will not terminate unless $\mathbf{e}_F^{(k+1)} = \mathbf{0}$ (or $N_F^{(k)} = 0$).

If $N_F^{(k)} > 0$ then the final step of the algorithm must be calculated using (16) and, furthermore, it must be the case that $\beta^{(k)} = 1$. Therefore from (17) it follows that if $N_F^{(k)} > 0$ then on termination of the algorithm $\mathbf{e}_F^{(k+1)} = \mathbf{0}$. So the algorithm cannot terminate unless the solution is optimal (ie. the KKT conditions (5) are met).

Now consider item 2. Clearly, each iteration of the algorithm will take a finite time to complete. Hence proving that the algorithm will terminate after a finite time is equivalent to proving that it will terminate after a finite number of iterations.

Firstly, suppose that $f^{(1)} \neq 0$. We know from theorem 4 that after some finite number of iterations, l, $f^{(l)} = 0$, and that for all subsequent iterations, $l+n$, $n \geq 0$, $f^{(l+n)} = 0$.

The usual form of the SVM optimisation problem (see, for example, [10]) is:

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \tfrac{1}{2}\boldsymbol{\alpha}^T\mathbf{G}\boldsymbol{\alpha} - \boldsymbol{\alpha}^T\mathbf{1} \\ &\text{such that:} \quad \mathbf{d}^T\boldsymbol{\alpha} = 0 \\ &\qquad\qquad\quad\; \mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1} \end{aligned}$$

So if $f = 0$ we can identify (16) with Equations (10.2.2) and (10.3.1) in [2]. If we assume that $\mathbf{G}$ is positive definite then the proof of convergence given in [2] can be directly applied to our algorithm. For the more general case where $\mathbf{G}$ is positive semidefinite our method may be identified as a more general form of that given in [3]. By analogy with the proof given in [3] it is straightforward to prove that the algorithm will terminate after a finite number of iterations.

So, in general, we know that after some finite number of iterations, l, $f^{(l)} = 0$. Furthermore, we know that for all subsequent iterations of the algorithm, $f^{(k)} = 0$, and that given this condition the algorithm will terminate after a finite number of iterations (after the first iteration where $f^{(l)} = 0$).

Hence, given any starting point $\boldsymbol{\alpha}, b$ satisfying $\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}$, the algorithm is guaranteed to find an optimal solution to (4) in a finite time. ■

IV. Factoring the Hessian

As defined previously, $\beth$ is the inverse or some other factorisation of $\mathbf{H}_R$ defined in such a way as to facilitate the fast calculation of $\mathbf{r}$ when $\mathbf{s}$ is known and $\mathbf{H}_R\mathbf{r} = \mathbf{s}$. In this section we will consider two such factorisations in detail, viz., the inverse and the Cholesky factorisation respectively. For both cases, we will consider the following issues:

- How to initially calculate $\beth$ upon entering the algorithm (unless $\beth$ is already known).
- How to quickly find $\mathbf{r}$ when $\mathbf{s}$ is known and $\mathbf{H}_R\mathbf{r} = \mathbf{s}$ using $\beth$ (fast matrix inversion).
- How to re-invert or re-factorise $\beth$ quickly when constraints are activated and de-activated, using a rank-1 updating procedure.

This section is organized as follows. In subsection IV-A we give some fundamental results related to the Hessian $\mathbf{H}_R$, and in particular how the form of $\mathbf{H}_R$ may (or may not) change when the active set is modified.

In subsections IV-B and IV-C we introduce the two factorisation methods and give any necessary background information and notation required in subsequent sections. In subsections IV-D, IV-E, IV-F and IV-G we consider in some detail how the factorisation is setup, modified and used. Finally, in subsection IV-H we show how the algorithm is optimised, and in IV-I compare the merits of the inverse and Cholesky factorisations.

A. Properties of the Hessian Matrix

For the purposes of this subsection alone, it is useful, if $N_R > 1$ to, to partition $\mathbf{G}_R$, $\mathbf{d}_R$, etc. as follows:

$$\mathbf{d}_R = \begin{bmatrix} \mathbf{d}_{Ra} \\ d_{Rb} \end{bmatrix}, \quad \text{etc.}$$
$$\mathbf{G}_R = \begin{bmatrix} \mathbf{G}_{Ra} & \mathbf{g}_{Rab} \\ \mathbf{g}_{Rab}^T & g_{Rb} \end{bmatrix}$$

It should also be noted that the ordering of the free variables in the partition (7) is arbitrary and may be changed as is convenient for the problem at hand. Hence when we speak of re-ordering variables, we mean re-ordering the free variables in (7).

If $N_F = 0$ then $\mathbf{H}_F = [0]$ is singular by definition. If $N_F = 1$ then, as we have shown in theorem 1, $\mathbf{H}_F$ is non-singular. Suppose $N_F > 1$. By definition, $\mathbf{H}_R$ is non-singular. If $N_R = 1$ then the form of $\mathbf{G}_R$ is restricted only by the requirement of positive semi-definiteness. Otherwise:

Theorem 6: If $N_R > 1$ then either $\mathbf{G}_R$ is non-singular or, after re-ordering, $\mathbf{G}_R$ is singular, $\mathbf{G}_{Ra}$ is non-singular ($g_{Rb} = \mathbf{g}_{Rab}^T \mathbf{G}_{Ra}^{-1} \mathbf{g}_{Rab}$) and $d_{Rb} \neq \mathbf{g}_{Rab}^T \mathbf{G}_{Ra}^{-1} \mathbf{d}_{Ra}$.

Proof: See appendix I. ∎

There are three basic forms which $\mathbf{H}_R$ may take, namely:

1) $N_F = 0$ and $\mathbf{H}_R$ is empty.
2) $N_R = 1$ and/or $\mathbf{G}_R$ is positive definite.
3) $N_R > 1$, $\mathbf{G}_R$ is singular and, after appropriate re-ordering, $\mathbf{G}_{Ra}$ is non-singular and $d_{Rb} \neq \mathbf{g}_{Rab}^T \mathbf{G}_{Ra}^{-1} \mathbf{d}_{Ra}$.

It is important to consider the effect on the singular nature of the Hessian matrix of activating and de-activating constraints.

Theorem 7: For all iterations $k \geq 1$ followed by the operation $\text{free}\left(m^{(k)}, X; k\right)$, $N_R^{(k+1)} \geq N_R^{(k)}$.

Proof: See appendix I. ∎

Theorem 8: For all iterations $k \geq 1$ followed by the operation $\text{constrain}\left(p^{(k)}, X; k\right)$, $N_R^{(k+1)} \geq N_R^{(k)} - 1$.

Proof: See appendix I. ∎

On a practical note, we must allow for the finite numerical precision of the computer. One implication of this is that theorem 8 may not, in practice, be seen to be correct. However, the results do provide a useful guide as to what may be expected from an implementation.

B. The Inverse Update Method

The obvious method of factorising the Hessian matrix $\mathbf{H}_R$ is by direct inversion. This makes the solution of the fast matrix inversion problem trivial (ie. use matrix multiplication), and, as will be shown in subsequent sections, updating the factorisation may be done efficiently using a rank-1 updating technique.

During the preparation of this paper, we found an independent work by Cauwenberghs and Poggio [7]. This paper describes a similar (although notationally different) algorithm to our own. However, [7] appear to not have considered the case where the Hessian matrix becomes singular. Furthermore, we have separated the issues of the general structure of the algorithm, and the selection of an appropriate method of inverting or factorising the Hessian matrix. By doing this, we are able to consider the relative merits of two such approaches, namely the inverse and Cholesky factorisations (more details on the latter may be found in section IV-C), in section IV-I. In each case, a rank-1 updating technique allows very efficient implementation.

We have the following definition:

Definition 9: For the *inverse update* method $\beth = \{\mathbf{V}\}$, where $\mathbf{V} = \mathbf{H}_R^{-1}$ (the inverse of an empty matrix is defined to be an empty matrix).

By default, $\mathbf{V}$ is assumed to be empty and $N_R = 0$.

C. Cholesky Update Method

It is well known [12] that, for any positive definite and symmetric matrix $\mathbf{M}$, there exists a lower triangular matrix $\mathbf{L}$ with positive diagonal entries such that $\mathbf{M} = \mathbf{L}\mathbf{L}^T$. The matrix $\mathbf{L}$ is known as the Cholesky factorisation of $\mathbf{M}$, denoted $\text{chol}(\mathbf{M})$, and has some nice numerical properties [13].

We know that, so long as it exists, $\mathbf{G}_R$ is positive semidefinite and symmetric. When dealing with a Cholesky factorisation, if $N_F \geq 1$ we will use the following notation:

$$\mathbf{d}_R = \begin{bmatrix} d_{Ra} \\ \mathbf{d}_{Rb} \end{bmatrix}, \quad \text{etc.}$$
$$\mathbf{G}_R = \begin{bmatrix} g_{Ra} & \mathbf{g}_{Rba}^T \\ \mathbf{g}_{Rba} & \mathbf{G}_{Rb} \end{bmatrix}$$

Assuming that $g_{Ra} \neq 0$ (for numerical purposes, $g_{Ra} \geq \varepsilon$) and using theorem 6, it is not difficult to show that we can define $\mathbf{L}$ such that:

$$\begin{bmatrix} g_{Ra} & d_{Ra} & \mathbf{g}_{Rba}^T \\ d_{Ra} & 0 & \mathbf{d}_{Rb}^T \\ \mathbf{g}_{Rba} & \mathbf{d}_{Rb} & \mathbf{G}_{Rb} \end{bmatrix} = \mathbf{L}\mathbf{J}\mathbf{L}^T \tag{21}$$

where $\mathbf{L}$ is a lower triangular matrix with positive diagonal elements, and:

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & \mathbf{0}^T \\ 0 & -1 & \mathbf{0}^T \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}$$

(21) is analogous to the standard Cholesky factorisation, except that the matrix that is being factorised not positive definite (although it is non-singular), and $J_{2,2} = -1$. If $g_{Ra} < \varepsilon$ then we cannot define $\mathbf{L}$ to satisfy (21). In this case we can differentiate two distinct cases, namely:

1) $N_F = 1$ or $G_{2,2} < \varepsilon$. In this case $N_R = 1$, but there is no way to factorise $\mathbf{H}_R \in \Re^{2\times 2}$ using a Cholesky type factorisation. Fortunately, however, the inversion of $\mathbf{H}_R$ is trivial in this case.
2) $N_F > 1$ and $G_{2,2} \geq \varepsilon$. In this case, after performing the operation $\text{swap}\left(1, 2, F; k\right)$ it is possible to find lower triangular $\mathbf{L}$ with positive diagonal elements satisfying (21).

We define a binary variable NoChol $\in \{\text{TRUE}, \text{FALSE}\}$ to indicate whether $\mathbf{L}$ may (NoChol $=$ FALSE, default) or may not (NoChol $=$ TRUE, case 1) be formed to satisfy equation (21). So, if at any point $g_{Ra} < \varepsilon$, we may either $\text{swap}(1, 2, F; k)$ if $N_F > 1$ and $G_{2,2} \geq \varepsilon$, or set NoChol := TRUE.

Formally, we define the Cholesky factorisation as follows:

Definition 10: For the *Cholesky update* method, $\sqsupset = \{\mathbf{L}, \text{NoChol}\}$. So long as $N_F > 0$ and NoChol $=$ FALSE:

$$\begin{bmatrix} g_{R1} & d_{R1} & \mathbf{g}_{R2}^T \\ d_{R1} & 0 & \mathbf{d}_{R2}^T \\ \mathbf{g}_{R2} & \mathbf{d}_{R2} & \mathbf{G}_{R3} \end{bmatrix} = \mathbf{L}\mathbf{J}\mathbf{L}^T$$

If $N_F = 0$ or NoChol $=$ TRUE then for completeness $\mathbf{L}$ is defined to be an empty matrix.

By default, $\mathbf{L}$ is assumed to be empty, NoChol $=$ FALSE and $N_R = 0$.

1) Basic Operations with Cholesky Factorisations: In this section we will give some basic results and algorithms which will be required later. Firstly, suppose that we are given the column vector $\mathbf{z}$ and wish to find $\mathbf{r}$ such that either $\mathbf{L}\mathbf{r} = \mathbf{z}$ or $\mathbf{L}^T\mathbf{r} = \mathbf{z}$. We can do these problems using *forward elimination* and *back substitution*, respectively, as described in algorithms 4.1-1 and 4.1-2 of [12]. As both algorithms are quadratic time algorithms, this allows us to perform fast matrix inversions.

Another algorithm that we will need for this section is a rank-1 update algorithm. Suppose that we are given some matrix $\bar{\mathbf{M}} = \bar{\mathbf{Q}}\bar{\mathbf{Q}}^T$, $\bar{\mathbf{Q}} = \text{chol}(\bar{\mathbf{M}})$, and we want to find $\mathbf{Q} = \text{chol}(\mathbf{M})$ such that $\mathbf{M} = \mathbf{Q}\mathbf{Q}^T$, where $\mathbf{M} = \bar{\mathbf{M}} + \mathbf{h}\mathbf{h}^T$. An algorithm that calculates $\mathbf{Q}$ from $\bar{\mathbf{Q}}$ and $\mathbf{h}$ is called a rank-1 update algorithm. Appropriate algorithms may be found in the standard texts [12], [14].

D. Setup algorithm

In this section, we detail how the factorisation $\sqsupset$ is calculated, if necessary, upon entering the algorithm. As the methods used are essentially standard algorithms with minor modifications, details have been deferred to appendix II. Algorithm 1 is used if an inverse factorisation method is chosen, and algorithm 2 is a Cholesky factorisation is chosen (see [12], for example). Both algorithms share two important features, namely:

1) If, as a result of a constraint being either activated or deactivated, $\mathbf{H}_R$ is improperly defined in such a way that $\mathbf{H}_{FNB}$ is non-singular, the setup algorithm may be called to extend $\mathbf{H}_R$ to its correct size.
2) If as a result of running the algorithm it is found that $N_F \neq N_R$ then $(s_b, \mathbf{s}_{\boldsymbol{\alpha}})$ in (18) will already be, at least partially, calculated. Hence most of the computational cost involved in calculating a step using equation (18) is avoided.

Indeed, based on point 1 above, the setup algorithm may be thought of as a means maximising N_R. Any prior knowledge (in the form of a partial factorisation) is used to "kick-start" the algorithm, minimising computational cost. As the default active set definition upon entering the algorithm in a "cold-start" situation is for all variables α_i to be actively constrained at a lower bound (ie. $N_F = 0$), the setup algorithm will rarely be required to increase N_R significantly, and hence contributes little to the computational cost of the algorithm.

Consider algorithm 1. If this algorithm terminates with $N_F \neq N_R$ then, as noted previously, the variable $(s_b, \mathbf{s}_{\boldsymbol{\alpha}})$ calculated most recently in algorithm 1 may be used directly during the subsequent iteration of the main algorithm when calculating the step using equation (18).

The analogous situation when using a Cholesky update method is not quite so simple. However, if algorithm 2 terminates with $N_F \neq N_R$ a significant computational cost saving may still be had when calculating $(s_b, s_{\boldsymbol{\alpha}})$ by solving:

$$\mathbf{L}^T \begin{bmatrix} s_{\boldsymbol{\alpha}a} \\ s_b \\ \mathbf{s}_{\boldsymbol{\alpha}b} \end{bmatrix} = \mathbf{a}$$

where $\mathbf{a}$ is as calculated during the final iteration of algorithm 2, and:

$$s_{\boldsymbol{\alpha}} = \begin{bmatrix} s_{\boldsymbol{\alpha}a} \\ \mathbf{s}_{\boldsymbol{\alpha}b} \end{bmatrix}$$

is the vectorial part of $(s_b, s_{\boldsymbol{\alpha}})$ to be used directly in equation (18).

E. Fast Matrix Inversion

We now consider how the factorisation $\sqsupset$ of the relevant part of the Hessian matrix $\mathbf{H}_R$ may be used to quickly calculate the vector $\mathbf{r}$, where $\mathbf{s}$ is known, $\mathbf{H}_R\mathbf{r} = \mathbf{s}$ and, by assumption, $\mathbf{H}_R$ is non-singular and $N_R > 0$.

If we are using an inverse factorisation method, then this calculation is trivial, as $\mathbf{r} = \mathbf{V}\mathbf{s}$. If we use a Cholesky factorisation and NoChol $=$ FALSE then, apart from some minor housekeeping, the calculation is not too difficult. In detail:

- Swap elements 1 and 2 in $\mathbf{s}$.
- Using forward elimination, find $\mathbf{t}$, where $\mathbf{L}\mathbf{t} = \mathbf{s}$.
- Negate element 2 of $\mathbf{t}$.
- Using back substitution, find $\mathbf{r}$, where $\mathbf{L}^T\mathbf{r} = \mathbf{t}$.
- Swap elements 1 and 2 in $\mathbf{r}$.

If we use a Cholesky factorisation and NoChol $=$ TRUE then may simply calculate the solution explicitly, ie.:

$$\begin{bmatrix} r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} -G_{1,1}s_1 + d_1 s_2 \\ d_1 s_1 \end{bmatrix}$$

F. Activating a Constraint

When a constraint is activated by the function $\text{constrain}\left(p^{(k)}, X; k\right)$, $\sqsupset$ must be updated to reflect the modification to the active set. We consider the following possibilities:

1) $p^{(k)} > N_R^{(k)} + 1$.
2) $p^{(k)} = N_R^{(k)} + 1$.

3) $p^{(k)} = N_R^{(k)} = N_F^{(k)} = 1$.
4) $p^{(k)} = N_R^{(k)} = 1$ and $N_F^{(k)} > 1$.
5) $p^{(k)} \leq N_R^{(k)}$ and $N_R^{(k)} > 1$.

Where the Hessian $\mathbf{H}_F$ must be singular in cases 1, 2 and 4, and may be singular in case 5. For the first 4 cases, the modification to $\beth$ is as follows:

1: As neither $\mathbf{H}_{FN}$ nor $\mathbf{H}_{FNB}$ are affected, $\beth^{(k+1)} = \beth^{(k)}$.
2: While $\mathbf{H}_{FN}$ is not affected, $\mathbf{H}_{FNB}$ is, and so it may be possible to increase N_R if $N_R^{(k)} \neq N_F^{(k)} - 1$. Hence if $N_R^{(k)} \neq N_F^{(k)} - 1$ then, after making the appropriate changes to $\mathbf{H}_F$, algorithm 1 (or 2, depending on the update method used) is called to update $\beth$ and maximise N_R.
3: After this operation, $\beth$ will revert to its default form. Hence all matrices in $\beth^{(k+1)}$ will be empty, $N_R^{(k+1)} = 0$ and NoChol = FALSE if a Cholesky update method is used.
4: $\beth$ must be re-built from scratch after the appropriate changes have been made to $\mathbf{H}_F$.

For case 5, the modification is dependent on the factorisation used, as will now be described.

1) Inverse update method: The inverse of $\mathbf{H}_R$ at iteration k, $\mathbf{V}^{(k)}$, may be partitioned as follows:

$$\mathbf{V}^{(k)} = \begin{bmatrix} \mathbf{V}_a & \mathbf{v}_d & \mathbf{V}_b^T \\ \mathbf{v}_d^T & v_e & \mathbf{v}_f^T \\ \mathbf{V}_b & \mathbf{v}_f & \mathbf{V}_c \end{bmatrix}$$

where $\mathbf{V}_a \in \Re^{\left(p^{(k)}-1\right) \times \left(p^{(k)}-1\right)}$.

If $\mathbf{H}_F$ was non-singular prior the activation of the constraint (ie. $N_R^{(k)} = N_F^{(k)}$) then:

$$\mathbf{V}^{(k+1)} = \begin{bmatrix} \mathbf{V}_a & \mathbf{V}_b^T \\ \mathbf{V}_b & \mathbf{V}_c \end{bmatrix} - \frac{1}{v_e} \begin{bmatrix} \mathbf{v}_d \\ \mathbf{v}_f \end{bmatrix} \begin{bmatrix} \mathbf{v}_d \\ \mathbf{v}_f \end{bmatrix}^T \quad (22)$$

Otherwise, we may still use equation (22) to calculate an interim form of $\beth^{(k+1)}$, and then call algorithm 1 to increase (to a maximum) $N_R^{(k+1)}$.

According to theorem 8, equation (22) must be well defined, and, in particular, $v_e \neq 0$. Unfortunately, due to cumulative numerical errors, we may find that this is not true. Even if $v_e \neq 0$, if $|v_e| < \varepsilon$ it would be inadvisable to use equation (22), as the likely result would be large numerical errors in $\mathbf{V}$.

Our strategy for dealing with this problem is to attempt to reduce $N_R^{(k)}$ by some minimal amount to ensure that $\mathbf{V}^{(k+1)}$ is non-singular to working precision (ie. $|v_e| \geq \varepsilon$ when we attempt to use equation (22)).

Let us assume $p^{(k)} = N_R^{(k)}$ (if this is not true then, noting that $\mathbf{V}$ may be re-ordered in the same way as $\mathbf{H}_F$, we may perform the operation $\mathrm{swap}\left(p^{(k)}, N_R^{(k)}, F; k\right)$ to make it so). Hence $\mathbf{V}_c$ has zero size. If $N_R^{(k)} \geq 4$ then our strategy is to partition $\mathbf{V}$ as follows:

$$\mathbf{V}^{(k)} = \begin{bmatrix} \mathbf{V}_\alpha & \mathbf{V}_\beta^T \\ \mathbf{V}_\beta & \mathbf{V}_\gamma \end{bmatrix}$$

where $\mathbf{V}_\gamma \in \Re^{N_\gamma \times N_\gamma}$. We aim to find the smallest value of N_γ in the range $2 \leq N_\gamma \leq \left\lfloor \frac{N_R^{(k)}}{2} \right\rfloor$ such that $\det\left(\mathbf{V}_\gamma\right) >= \varepsilon$. If we succeed in finding such an N_γ then:

$$\mathbf{V}^{(k+1)} = \mathbf{V}_\alpha - \mathbf{V}_\beta^T \mathbf{V}_\gamma^{-1} \mathbf{V}_\beta$$

where $\mathbf{V}_\gamma^{-1}$ may be calculated using a simple variant of algorithm 1. Note that it is not computationally worthwhile to increase N_γ past the limit set above, as the computational cost of calculating $\mathbf{V}_\gamma^{-1}$ would then exceed the computational cost of re-calculating $\mathbf{V}^{(k+1)}$ from scratch using algorithm 1. If this method fails we set $N_R := 0$, $\mathbf{V}$ empty and use algorithm 1 to calculate $\mathbf{V}^{(k+1)}$ from scratch.

From a computational perspective, this method is far from optimal if N_γ becomes large. However, in our experience cases where $N_\gamma > 2$ are extremely rare, and in any case indicate that a significant cumulative numerical error has occurred that is best rectified by re-calculating $\mathbf{V}$ from scratch.

2) Cholesky update method: Firstly, let us suppose that $p^{(k)} \neq 1$. Hence $\mathbf{L}^{(k)}$ may be partitioned as follows:

$$\mathbf{L}^{(k)} = \begin{bmatrix} \mathbf{L}_a & \mathbf{0} & \mathbf{0} \\ \mathbf{l}_d^T & l_e & \mathbf{0}^T \\ \mathbf{L}_b & \mathbf{l}_f & \mathbf{L}_c \end{bmatrix}$$

where $\mathbf{L}_a \in \Re^{\left(p^{(k)}-1\right) \times \left(p^{(k)}-1\right)}$. Assuming that $N_R^{(k+1)} = N_R^{(k)} - 1$:

$$\mathbf{L}^{(k+1)} := \begin{bmatrix} \mathbf{L}_a & \mathbf{0} \\ \mathbf{L}_b & \bar{\mathbf{L}}_c \end{bmatrix}$$

where:

$$\bar{\mathbf{L}}_c \bar{\mathbf{L}}_c^T = \mathbf{L}_c \mathbf{L}_c^T + \mathbf{l}_f \mathbf{l}_f^T$$

If $N_R^{(k)} \neq N_F^{(k)}$ then algorithm 2 is called subsequently to maximise N_R.

If, however, $p^{(k)} = 1$, then our only recourse is make $\mathbf{L}$ empty, $N_R = 0$ and NoChol = FALSE, make the necessary changes to $\mathbf{H}_F$ etc., and then use algorithm 2 to make $\mathbf{L}^{(k+1)}$ from scratch. In extreme cases, we may find (as a result of numerical errors) that $N_R^{(k+1)} < N_R^{(k)} - 1$. However, unlike the similar problem when dealing with an inverse update, this may occur only if $p^{(k)} = 1$, and even then is, in our experience, quite rare.

G. De-activating a Constraint

When a constraint is de-activated by the function $\mathrm{free}\left(m^{(k)}, X; k\right)$, $\beth$ must be updated to reflect the modification to the active set. Clearly, if $\mathbf{H}_F$ is singular and $N_F^{(k)} > 1$ then adding a row and column to the end of this matrix will not effect $\mathbf{H}_R$. Hence if $N_R^{(k)} \neq N_F^{(k)}$ and $N_F^{(k)} > 1$, $\beth^{(k+1)} = \beth^{(k)}$. Otherwise, the re-entrant properties of algorithms 1 and 2 may be used to advantage by simply updating $\mathbf{H}_F$ etc. appropriately and calling the relevant factorisation algorithm, 1 or 2.

H. Optimisations

1) Computational cost optimisation: In theorem 4 it was shown if $f^{(1)} \neq 0$ then there exists some l such that $f^{(l+n)} = 0$ for all $n \geq 0$. Furthermore, for most of these iterations, if $N_F^{(l+n)} > 0$ then $\mathbf{e}_F^{(l+n)}$ will have the form:

$$\mathbf{e}_F^{(l+n)} = \begin{bmatrix} \mathbf{0} \\ \mathbf{e}_{Fnz}^{(l+n)} \end{bmatrix}$$

where $\mathbf{e}_{Fnz}^{(l+n)} \in \Re^{N_{Fnz}^{(l+n)}}$. Indeed, in most cases (all if theorem 8 holds true in spite of numerical errors), $N_{Fnz}^{(l+n)} = 1$. Using this fact, it is possible to significantly accelerate the calculation of equation (16) using $\beth$.

First, suppose we are using an inverse factorisation. In this case, if $f^{(k)} = 0$ and $N_{Fnz}^{(k)} < N_F^{(k)}$, (16) reduces to:

$$\begin{bmatrix} \Delta b^{(k)} \\ \Delta \boldsymbol{\alpha}_F^{(k)} \end{bmatrix} = -\mathbf{V}_{nz} \mathbf{e}_{Fnz}^{(k)}$$

where $\mathbf{V}_{nz} \in \Re^{(N_F^{(k)}+1) \times N_{Fnz}^{(k)}}$ and:

$$\mathbf{V} = \begin{bmatrix} \mathbf{V}_z & \mathbf{V}_{nz} \end{bmatrix}$$

If we are using a Cholesky factorisation then, if $f^{(k)} = 0$, NoChol = FALSE, $N_F^{(k)} > 1$ and $N_{Fnz}^{(k)} < N_F^{(k)}$, (16) may be solved as follows:

- Using forward elimination, find $\mathbf{t}$, where $\mathbf{L}_{nz}\mathbf{t} = \mathbf{e}_{Fnz}^{(k)}$.
- Using back substitution, find $\mathbf{r}$, where $\mathbf{L}^T\mathbf{r} = \begin{bmatrix} \mathbf{0} \\ \mathbf{t} \end{bmatrix}$.
- Swap elements 1 and 2 in $\mathbf{r}$.
- $\begin{bmatrix} \Delta b^{(k)} \\ \Delta \boldsymbol{\alpha}_F^{(k)} \end{bmatrix} = -\mathbf{r}$

where $\mathbf{L}_{nz} \in \Re^{N_{Fnz}^{(k)} \times N_{Fnz}^{(k)}}$ and:

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_z & \mathbf{0} \\ \mathbf{L}_{nzz} & \mathbf{L}_{nz} \end{bmatrix}$$

Another area where some computational cost savings may be made is the calculation of $\left(\Delta \mathbf{e}_U^{(k)}, \Delta \mathbf{e}_L^{(k)}\right)$ when $\mathbf{H}_F$ is singular. As was shown in equation (18), $\Delta \boldsymbol{\alpha}_{FS}^{(k)} = \mathbf{0}$ in this case. Extending our notation in the obvious manner, we see that equation (8) may be simplified to:

$$\begin{bmatrix} \Delta \mathbf{e}_U^{(k)} \\ \Delta \mathbf{e}_L^{(k)} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_U & \mathbf{G}_{UFN} & \mathbf{g}_{UFB} \\ \mathbf{d}_L & \mathbf{G}_{LFN} & \mathbf{g}_{LFB} \end{bmatrix} \begin{bmatrix} \Delta b^{(k)} \\ \Delta \boldsymbol{\alpha}_{FN}^{(k)} \\ \Delta \alpha_{FB}^{(k)} \end{bmatrix}$$

2) Memory usage optimisation: As our algorithm stores $\mathbf{H} \in \Re^{N \times N}$ in full and also any matrices associated with the factorisation $\beth$, it will naturally use significant amounts of memory if our training set size N is excessively large. Hence, the algorithm we have described is intended mainly for problems where the amount of training data is small to moderate. Because of this, it is important to consider issues of memory use, and how it may be minimised.

It will be noted that all matrices used in our algorithm are either lower triangular ($\mathbf{L}$, as used in the Cholesky update method) or symmetric (the large matrix $\mathbf{H}$, and also $\mathbf{V}$, as used in the inverse update method). We take advantage of this symmetry by storing all of these matrices in lower triangular form, thus reducing matrix memory usage from $N^2 + O(N)$ (assuming $N_R \ll N$) to $\frac{1}{2}N^2 + O(N)$.

I. Comparative Merits

In this section, we have introduced two related rank-1 updating algorithms for the re-inversion or re-factorisation of the Hessian matrix, viz., the inverse update method, and the Cholesky update method respectively. In this subsection, we will compare their relative merits and de-merits.

1) Inverse update method: The advantages and disadvantages of the inverse update method are:

- Advantages:
  1) Speed - updating the inverse Hessian is significantly faster than calculating it from scratch.
  2) Simplicity - the algorithms to form and update the factorisation are simpler than their Cholesky update counterparts.
- Disadvantages:
  1) Numerical stability - as was seen in section IV-F.1, numerical errors can easily lead to significant inaccuracies in our inverse factorisation $\mathbf{V}$ and, consequently, our optimal solution.

2) Cholesky update method: The advantages and disadvantages of the Cholesky update method are:

- Advantages:
  1) Speed - calculating the step is significantly faster than calculating from scratch.
  2) Numerical stability - while it is still possible that, due to numerical errors, we may have problems with previously known non-singular matrices "appearing" non-singular as a result of the de-activation of a constraint, the likelihood of such an event is much lower when using a Cholesky update method than when using an inverse update method.
- Disadvantages:
  1) Complexity - the Cholesky update method is significantly more complex than the inverse update method.

3) Computational complexity comparisons: Another way in which we can compare the two proposed algorithms is to consider their respective computational complexities.

Consider the computational cost of an iteration. This may be split into factorisation dependent and factorisation independent components, where "factorisation" refers to either the inverse or Cholesky approach. The factorisation independent component of each iteration is the cost of calculating $\Delta \mathbf{e}_U^{(k)}$ and $\Delta \mathbf{e}_L^{(k)}$, as well as performing the update. This operation takes in the order of $2N_F(N_U + N_L) + 2N + 2N_F$ flops (a flop is defined here as a floating point multiplication, addition or square root). The factorisation dependent cost of each iteration is the cost of calculating $\Delta \boldsymbol{\alpha}_F^{(k)}$ and $\Delta b^{(k)}$ and also the cost of updating the factorisation whenever a constraint is activated or de-activated.

We now consider the factorisation dependent cost in more detail. For simplicity, we will assume that $N_{Fnz} = 1$ (as is most often the case), and also that the Hessian is always non-singular. With the exception of the final step, each such step calculation will be followed by a constraint activation or de-

activation. Hence it makes sense to look at the combined step plus constraint activation/de-activation cost.

For a step followed by a constraint activation, the computational cost of the inverse update method is approximately $3N_F + \left(N - p^{(k)}\right)^2$ flops, compared to $N_F^2 + 4\left(N - p^{(k)}\right)^2$ flops for the Cholesky method. Clearly, the inverse update method is significantly faster here. For a step followed by a constraint de-activation, however, the computational cost of the inverse update method is approximately $3N_F^2$ flops, which makes it significantly slower than the Cholesky method, which takes approximately N_F^2 flops.

From this, we observe that neither algorithm has a significant computational advantage over the other. For our dataset, we found the Cholesky method to be marginally faster than the inverse update method. However, it is known that usually $N_F \ll N$, so the factorisation dependent cost tends to be swamped by the factorisation independent cost. Therefore, in most cases, the computational cost issues of the two factorisation methods are likely to be less important than the complexity versus numerical stability trade-off.

V. Incremental Learning and Forgetting

The application of our algorithm to the problem of incremental learning is straightforward. Suppose that we have found an optimal solution to (4) for a given training set $\Theta = \{(\mathbf{x}_1, d_1), (\mathbf{x}_2, d_2), \ldots, (\mathbf{x}_N, d_N)\}$ to which we wish to add the additional training points $\hat{\Theta} = \left\{\left(\hat{\mathbf{x}}_1, \hat{d}_1\right), \left(\hat{\mathbf{x}}_2, \hat{d}_2\right), \ldots, \left(\hat{\mathbf{x}}_{\hat{N}}, \hat{d}_{\hat{N}}\right)\right\}$. Define:

$$\begin{aligned}
&\hat{\boldsymbol{\alpha}} = \mathbf{0} \\
&\hat{\mathbf{e}} = \hat{\mathbf{G}}\boldsymbol{\alpha} + \hat{\mathbf{d}}b - \mathbf{1} \\
&\hat{G}_{i,j} = \hat{d}_i d_j K\left(\hat{\mathbf{x}}_i, \mathbf{x}_j\right) \\
&\hat{\hat{G}}_{i,j} = \hat{d}_i \hat{d}_j K\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j\right)
\end{aligned} \tag{23}$$

We then add the new training points to our existing solution, actively constraining all elements of $\hat{\boldsymbol{\alpha}}$ to 0, as follows:

$$\begin{aligned}
\boldsymbol{\alpha} &:= \begin{bmatrix} \boldsymbol{\alpha} \\ \hat{\boldsymbol{\alpha}} \end{bmatrix}, \text{etc.} \\
\mathbf{G} &:= \begin{bmatrix} \mathbf{G} & \hat{\mathbf{G}}^T \\ \hat{\mathbf{G}} & \hat{\hat{\mathbf{G}}} \end{bmatrix}
\end{aligned}$$

If $\hat{\mathbf{e}} \geq \mathbf{0}$ then the KKT conditions (5) will be satisfied and our solution is optimal. Otherwise, we re-enter the optimisation algorithm (starting with our current solution) to find the optimal solution. Note that we do not re-calculate our inverse or factorisation $\beth^{(1)}$, as we can use the old factorisation from the last time we ran the algorithm.

Forgetting may be implemented in a similar manner. We simply remove the α_i's corresponding to the training points that we want to remove, re-calculate f and $\mathbf{e}$ according to (6), and re-enter our optimisation algorithm if the KKT conditions (5) are not satisfied. Note that this will only happen if we remove support vectors.

During forgetting, the inverse or factorisation $\beth^{(1)}$ may be altered due to the removal of free variables. The effect of the factorisation is the same as for the activation of a constraint on that variable, and so we can update our factorisation in the same manner as we would for a constraint activation.

VI. Constraint Parameter Variation

When designing and training an SVM, we need to select the kernel function and the constraint parameter C. Traditionally, this has been done by trial-and-error for kernel selection and (near) exhaustive search for C. More recently, [15] has given an algorithm for automatically selecting the kernel parameters in order to minimise an upper bound on the generalisation error. In either case, we must optimise the SVM for a large number of different choices of parameter. Using a batch re-solve method tends to make the process extremely slow.

Suppose that we have optimised our SVM for a given set of parameters, and want to repeat the process for a slightly different set of parameters. It is reasonable to expect that, assuming the difference between the parameters is sufficiently small, the solution for the new parameters will be close to the solution for the old parameter set. We will now show how the warm-start concept may be used to take advantage of this concept.

A. C Variation

Suppose that we vary C by ΔC, where $C + \Delta C > 0$. Assuming for the moment that $\boldsymbol{\alpha}_F \leq \mathbf{1}(C + \Delta C)$, we modify our present solution as follows:

$$\begin{aligned}
&\boldsymbol{\alpha}_U := \boldsymbol{\alpha}_U + \Delta C \mathbf{1} \\
&\begin{bmatrix} f \\ \mathbf{e} \end{bmatrix} := \begin{bmatrix} f \\ \mathbf{e} \end{bmatrix} + \Delta C \begin{bmatrix} \mathbf{d}_U^T \\ \mathbf{G}_{UF}^T \\ \mathbf{G}_U \\ \mathbf{G}_{LU} \end{bmatrix} \mathbf{1}
\end{aligned} \tag{24}$$

If $N_U = 0$ then the KKT conditions (5) will be satisfied and our solution will be optimal. Otherwise, we re-enter the optimisation algorithm (starting with our current solution) to find the optimal solution. Once again, we need not re-calculate $\beth^{(1)}$ from scratch.

If $\boldsymbol{\alpha}_F \not\leq \mathbf{1}(C + \Delta C)$ then we can complete our variation as a number of small steps. For each step, we take the largest step possible satisfying the previous condition, and then activate the relevant upper bound constraint (updating the factorisation as described previously). The process is then repeated until the required change in C has been achieved. Specifically, we calculate:

$$\begin{aligned}
\delta C &= - \min_{(\mathbf{x}_i, d_i) \in \Theta_U} (C - \alpha_i) \\
j &= \arg\min_{i:(\mathbf{x}_i, d_i) \in \Theta_U} (C - \alpha_i)
\end{aligned}$$

We then vary C by δC using (24) as described above. After completing this operation, we activate an upper bound constraint on α_j as described previously. We then repeat the process using $\Delta C := \Delta C - \delta C$. As N_F is finite, after a finite number of repetitions we will find that either $\boldsymbol{\alpha}_F \leq \mathbf{1}(C + \Delta C)$ or $N_F = 0$. In either case, we will have affected the required adjustment to C.

B. Kernel Variation

If the kernel function is modified, the matrix $\mathbf{G}$ will change, and hence the Hessian will need to be re-factorised. We can

then re-calculate $(\mathbf{e}, f)$ accordingly using (6), and re-enter our optimisation algorithm if necessary. Unfortunately, the cost of re-factorising the Hessian is likely to be significant in this case. Nevertheless, it is theoretically possible to use this method for modifying the kernel function.

VII. Experimental Results

A. Implementation Details

We implemented both our algorithm and Platt's SMO algorithm [16] in C++[1]. Two computers were used during the experiment. Smaller tests were carried out on a 1GHz Pentium III with 512MB running Windows 2000, using DJGPP for code compilation. Larger tests were carried out on a 128 processor Compaq Alphaserver SC, with 64 GByte of memory and 1.4 TByte of disk running UNIX[2], using cc for code compilation.

To ensure the comparison was based on algorithmic efficiency rather than the efficiency of the implementation, ability of the compiler to optimise the code and the speed of the computer on which the experiment was done, computational cost has been measured explicitly in terms of flops, where one flop is defined here as one floating point addition, multiplication or square root (we are assuming here that an fpu is present).

For the SMO algorithm, an accuracy of $\varepsilon = 10^{-2}$ was chosen. All training data was normalised (to give a zero mean and unity variance) and randomly ordered prior to use. The kernel function used for all experiments was the quadratic kernel $K(\mathbf{x}, \mathbf{y}) = \left(1 + \frac{1}{d_L}\mathbf{x}^T\mathbf{y}\right)^2$.

B. Experimental Methodology

Our aim here is to investigate the advantages and disadvantages of using an incremental training algorithm instead of a batch training algorithm in those situations where a partial solution (either in the form of an SVM trained on some subset of the complete training set, or an SVM trained using different parameters) is available. As such, the accuracy of the resultant SVM (which is essentially independent of the training algorithm in any case) is of secondary importance, with the computational cost of updating (or re-training) the SVM being our primary concern.

When we speak of the computational cost of updating an SVM, either through incremental learning or incremental parameter variation, it is assumed that we are given the "old" solution a-priori. So the computational cost of updating this SVM will include only the cost of modifying this old solution in an appropriate manner. Specifically, when investigating incremental training we are concerned with the cost of updating an SVM with a *base* training set of N training pairs, to which we add M additional training pairs.

For our first experiment, we consider incremental learning where the problem at hand is relatively simple, and a high degree of accuracy may be achieved using only a small fraction of the complete training set (UCI mushroom toxicity dataset [17]). In this case, an error rate of less than 5% (tested using 800 vectors not contained in the training set) was achieved using a training set of less than 100 vectors out of a possible 7000. This allows us investigate the "steady state" computational cost where the decision surface is essentially static, suffering only occasional perturbations when new training data is added to the training set. The accuracy of the SVM for different training set sizes is shown in figure 2.

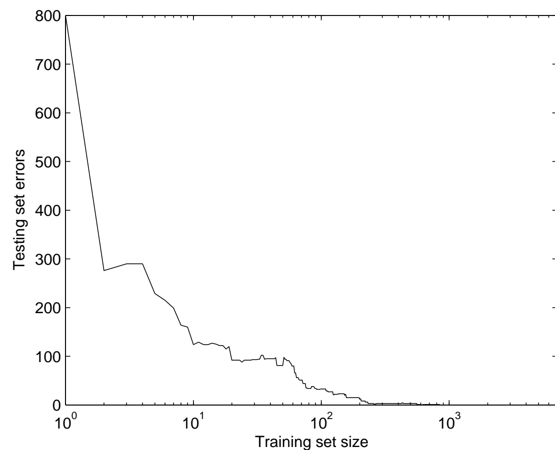


Fig. 2. Testing set error versus training set size - mushroom dataset

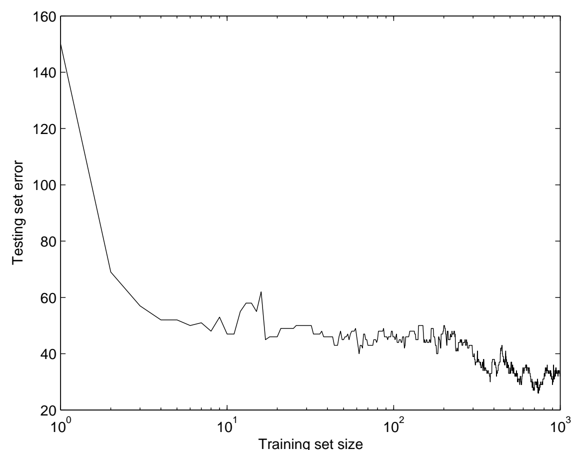


Fig. 3. Testing set error versus training set size - adult dataset

For our second experiment, we consider an incremental training problem of significantly greater difficulty (the adult dataset from the delve repository[3]). In this case we were unable to achieve an error rate less than approximately 20%, as shown by figure 3 (measured using 150 vectors not contained in the training set). This allows us to investigate the properties of the algorithm when the solution changes significantly with each increment.

For our third and final experiment, we consider the computational cost of incrementally varying the constraint parameter C, as may be required to find the optimal value for this constraint parameter. For speed of calculation, we have chosen

[1] code available at http://www.ee.mu.oz.au/staff/swami/svm/

[2] http://www.vpac.org

[3] see http://www.cs.toronto.edu/∼delve/data/datasets.html

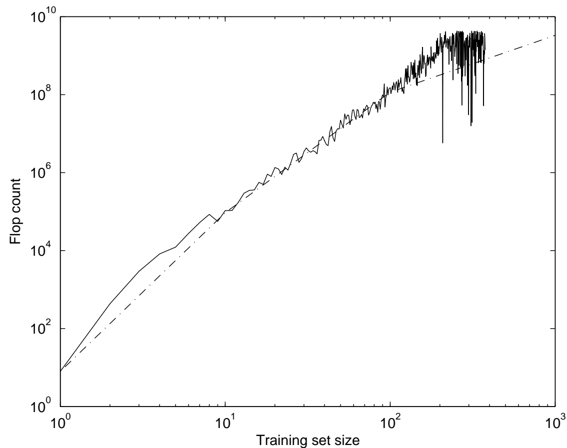


Fig. 4. Flop count vs. training set size (adult dataset) - SMO algorithm (dashed line batch, solid line incremental ($M = 1$)).

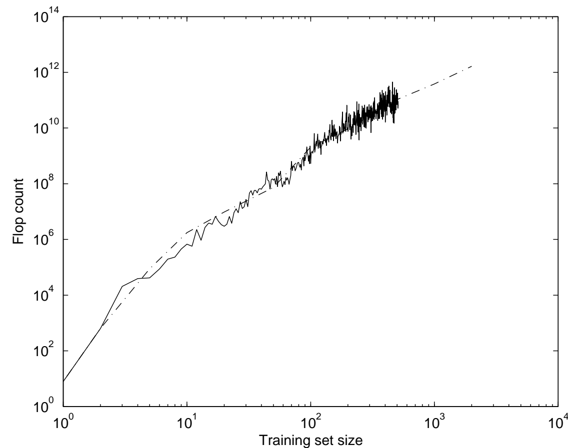


Fig. 5. Flop count vs. training set size (mushroom dataset) - SMO algorithm (dashed line batch, solid line incremental ($M = 1$)).

to use the smaller adult dataset for this component of the experiment.

C. Experimental results

1) Incremental Training: Figure 4 is typical of the computational cost for the SMO algorithm (adult dataset). The dashed line in this figure shows the computational cost of batch optimising the SVM using an SMO algorithm for different training set sizes, and the solid line the computational cost of incrementally adding $M = 1$ training points to an existing SVM (optimised for an existing training set size) using the SMO algorithm[4].

As can be seen from this graph, there is no advantage to be had using incremental SMO methods. Figure 5 shows the same result for the mushroom toxicity dataset using the same increment size ($M = 1$). For both datasets, we found our algorithm was significantly faster than SMO (compare, for example, figures 4 and 6). However, it must be born in mind that the datasets considered in the present paper are relatively small, and the SMO algorithm is optimised for much larger datasets where it becomes impractical to store the entire Hessian in memory. On average, we found that the number of flops our algorithm took to train the SVM (batch method) was usually around $\frac{1}{20}^{\mathrm{th}}$ of the number of flops required by the SMO to complete the same problem.

Figure 6 shows a comparison between batch and incremental cost using the active set algorithm (adult dataset) for an increment size of $M = 1$. In this figure, the *base cost* is the cost of incrementally extending the matrix $\mathbf{G}$ and also calculating the gradient $\hat{\mathbf{e}}$, and thus represents the minimum possible incremental update cost for an increment of size $M = 1$ given an existing training set of the size indicated

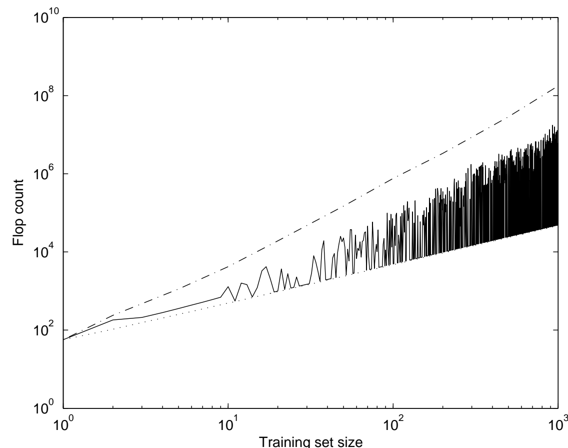


Fig. 6. Flop count vs. training set size (adult dataset) - active set algorithm (Cholesky factorisation) (in order of increasing magnitude: base cost, incremental cost ($M = 1$) and batch cost).

on the x-axis. As shown by the graph, the computational cost for a significant proportion (52%) of updates is exactly equal to the base cost, which indicates that the optimal SVM has not been changed when the new training data was added. Even when the incremental cost exceeds this base cost, the incremental computational cost is almost always significantly less (and always, for our datasets, at least slightly less) than the comparable batch optimisation cost. On average, we found that the incremental update method was 870 times faster than the comparable batch method for this example. If only those increments which modified the SVM in a non-trivial manner were considered, this dropped to 37 times faster on average, which is still a significant improvement.

For larger increment sizes (for example, $M = 100$ for the adult dataset is shown in figure 7, $M = 1000$ for mushroom toxicity dataset in figure 8), the incremental cost is more likely to exceed the base cost. However, we still found the incremental method to be faster than the batch method in all

[4]Incremental SMO learning may be achieved by simply keeping the old $\boldsymbol{\alpha}$ value, setting $\hat{\boldsymbol{\alpha}} = \mathbf{0}$, and starting at this value. This method is not applicable to decremental learning or constraint parameter variation, as in general we must start the SMO algorithm with $f = 0$.

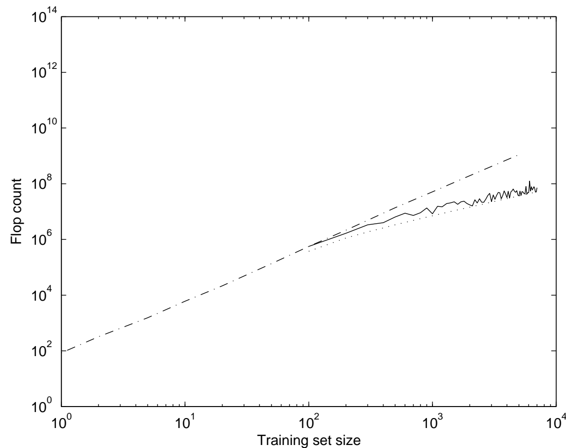


Fig. 7. Flop count vs. training set size (mushroom dataset) - active set algorithm (inverse factorisation) (in order of increasing magnitude: base cost, incremental cost ($M = 100$) and batch cost).

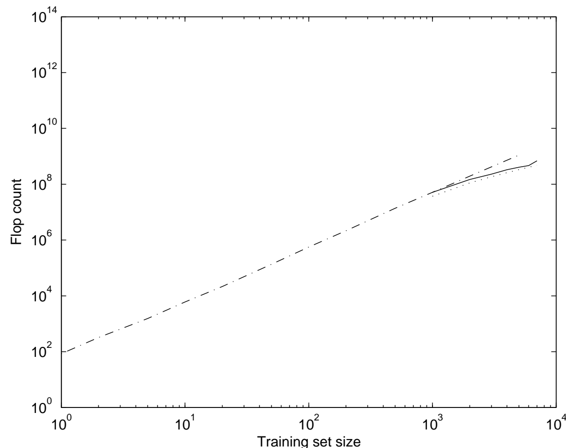


Fig. 8. Flop count vs. training set size (mushroom dataset) - active set algorithm (Cholesky factorisation) (in order of increasing magnitude: base cost, incremental cost ($M = 1000$) and batch cost).

cases (for the datasets in question).

For this first part of the experiment, we found no significant differences between the performance of the two proposed factorisation methods (inverse and Cholesky), either in computational cost or the optimal SVM found. However, as will be seen in the following section, this does not indicate that there are no differences in general.

2) Incremental Constraint Parameter Variation: Table I gives the computational cost of incrementally changing C from some initial value (given at the top of the column) to new value (given at the left of the row), as well as the batch computational cost of batch-optimising for the C value in question along the diagonal, for the adult dataset.

It will be noted that, so long as we are increasing C, the computational cost of the incrementally modifying an SVM is usually smaller than the computational cost associated with batch re-training for the new value of C. Indeed, for most cases

TABLE I

COMPUTATIONAL COST MATRIX (MFLOPS) - C VARIATION (ACTIVE SET METHOD - CHOLESKY FACTORISATION). INITIAL C VALUES ARE SHOWN IN THE TOP ROW, TARGET C VALUES IN THE LEFT COLUMN. DIAGONAL ENTRIES SHOW BATCH COSTS.

batch	0.01	0.1	1	10	100	1000	10000
0.01	*35.3*	82.3	133	159	178	185	184
0.1	8.79	*38.7*	123	156	180	188	187
1	21.6	18.6	*73*	156	192	200	200
10	69.7	64.8	50.6	*177*	181	215	227
100	143	160	148	81	*424*	214	274
1000	320	335	299	180	87.3	*1020*	326
10000	426	451	468	257	167	107	*1810*

shown in table I it is computationally cheaper to batch train an SVM for a small value of C (C_{small}) and then incrementally modify C to some larger value (C_{large}) than it is to batch train the SVM using C_{large}.

When decreasing C, however, we found that in many cases (especially when either the change in C was large or the target C was small) it was computationally cheaper to batch retrain the SVM using the new value of C rather than using an incremental approach. This is not too surprising, as when C is decreased we are more likely to have to modify the Hessian factorisation ⊐ than when C is increased, resulting in a higher computational cost for the former.

We were unable to complete the table using an inverse factorisation method, as we found that the numerical errors resulting from the inverse update factorisation procedure quickly became unacceptably high, leading to convergence problems. This result, combined with the similarity of computational cost between the two factorisation methods, would appear to indicate that, in general, the Cholesky factorisation method is superior to the inverse factorisation method, in spite of the additional algorithmic complications involved in its implementation.

VIII. CONCLUSIONS

We have investigated the use of incremental active set methods in the training of support vector machines. We have presented a simple warm-start active set algorithm applicable to the SVM problem, and demonstrated the computational advantages which incremental methods have over batch methods. We have also introduced a technique for fast constraint parameter variation using the same warm-start active set concept. All of these methods have been demonstrated on a practical, non-trivial problem, and the computational benefits have been shown to be significant.

Appendix I

Proofs of Properties of the Hessian Matrix

The following proofs refer to theorems in section IV-A.

Theorem 6: If $N_R > 1$ then either $\mathbf{G}_R$ is non-singular or, after re-ordering, $\mathbf{G}_R$ is singular, $\mathbf{G}_{Ra}$ is non-singular ($g_{Rb} = \mathbf{g}_{Rab}^T \mathbf{G}_{Ra}^{-1} \mathbf{g}_{Rab}$) and $d_{Rb} \neq \mathbf{g}_{Rab}^T \mathbf{G}_{Ra}^{-1} \mathbf{d}_{Ra}$.

Proof: Suppose $\mathbf{G}_R$ is non-singular. Given that $\mathbf{d}_R$ is non-zero and $\mathbf{G}_R$ is positive definite, it follows that $\mathbf{d}_R^T \mathbf{G}_R^{-1} \mathbf{d}_R \neq 0$. Therefore $\mathbf{H}_R$ is non-singular as required as we can form its inverse:

$$\mathbf{H}_R^{-1} = \begin{bmatrix} -\frac{1}{\mathbf{d}_R^T \mathbf{G}_R^{-1} \mathbf{d}_R} & \frac{\mathbf{d}_R^T \mathbf{G}_R^{-1}}{\mathbf{d}_R^T \mathbf{G}_R^{-1} \mathbf{d}_R} \\ \frac{\mathbf{G}_R^{-1} \mathbf{d}_R}{\mathbf{d}_R^T \mathbf{G}_R^{-1} \mathbf{d}_R} & \mathbf{G}_R^{-1} - \frac{\mathbf{G}_R^{-1} \mathbf{d}_R \mathbf{d}_R^T \mathbf{G}_R^{-1}}{\mathbf{d}_R^T \mathbf{G}_R^{-1} \mathbf{d}_R} \end{bmatrix}$$

Now suppose that $\mathbf{G}_R$ is singular. First, consider the case where $\mathbf{G}_R = \mathbf{0}$. As $N_R > 1$ it follows that $\mathbf{H}_R$ is singular, which contradicts the known non-singularity of $\mathbf{H}_R$. So $\mathbf{G}_R \neq \mathbf{0}$. Hence it must be possible (after some re-ordering) to write $\mathbf{G}_R$ in the form:

$$\mathbf{G}_R = \begin{bmatrix} \mathbf{G}_{RX} & \mathbf{G}_{RY} \\ \mathbf{G}_{RY}^T & \mathbf{G}_{RY}^T \mathbf{G}_{RX}^{-1} \mathbf{G}_{RY} \end{bmatrix}$$
$$\mathbf{d}_R = \begin{bmatrix} \mathbf{d}_{RX} \\ \mathbf{d}_{RY} \end{bmatrix}$$

where $\mathbf{G}_{RX}$ is positive definite. As $\mathbf{H}_R$ is non-singular its inverse must be:

$$\mathbf{H}_R^{-1} = \begin{bmatrix} \mathbf{G}_{RX}^{-1} + \mathbf{M}^T \mathbf{D}^{-1} \mathbf{M} & -\mathbf{M}^T \mathbf{D}^{-1} \\ -\mathbf{D}^{-1} \mathbf{M} & \mathbf{D}^{-1} \end{bmatrix}$$

where:

$$\mathbf{D} = \begin{bmatrix} \mathbf{0} & \mathbf{d}_{RY} - \mathbf{G}_{RY}^T \mathbf{G}_{RX}^{-1} \mathbf{d}_{RX} \\ \mathbf{d}_{RY}^T - \mathbf{d}_{RX}^T \mathbf{G}_{RX}^{-1} \mathbf{G}_{RY} & -\mathbf{d}_{RX}^T \mathbf{G}_{RX}^{-1} \mathbf{d}_{RX} \end{bmatrix}$$
$$\mathbf{M} = \begin{bmatrix} \mathbf{G}_{RY}^T \\ \mathbf{d}_{RX}^T \end{bmatrix} \mathbf{G}_{RX}^{-1}$$

So, $\mathbf{D}$ is singular iff $\mathbf{H}_R$ is singular. The two necessary conditions for $\mathbf{D}$ to be non-singular are as stated in the theorem. ■

Theorem 7: For all iterations $k \geq 1$ followed by the operation $\mathrm{free}\left(m^{(k)}, X; k\right)$, $N_R^{(k+1)} \geq N_R^{(k)}$.

Proof: If $N_R^{(k)} = 0$ then the theorem is trivial. Otherwise, note that the operation $\mathrm{free}\left(m^{(k)}, X; k\right)$ will result in a new row and column being added to the end of $\mathbf{H}_F$, but otherwise $\mathbf{H}_F$ will not be modified, so N_R cannot decrease. ■

Theorem 8: For all iterations $k \geq 1$ followed by the operation $\mathrm{constrain}\left(p^{(k)}, X; k\right)$, $N_R^{(k+1)} \geq N_R^{(k)} - 1$.

Proof: Clearly, if we are performing the operation $\mathrm{constrain}\left(p^{(k)}, X; k\right)$, $N_F^{(k)} \geq 1$ and hence, from theorem 1, $N_R^{(k)} \geq 1$ likewise.

If $N_R^{(k)} = 1$ then, as $N_R^{(k)} \geq 1$ by definition, it follows that $N_R^{(k+1)} \geq N_R^{(k)} - 1$.

If $N_R^{(k)} = 2$ then $N_F^{(k+1)} \geq 1$, and hence by theorem 1, $N_R^{(k)} \geq 1$, so $N_R^{(k+1)} \geq N_R^{(k)} - 1$.

The case $N_R^{(k)} \geq 3$ is not so simple. First off, suppose that $p^{(k)} > N_R^{(k)}$. As $\mathbf{H}_R^{(k)}$ is unaffected by the operation, it follows that its non-singular nature will not change and hence $N_R^{(k+1)} \geq N_R^{(k)} - 1$.

Now consider the case where $1 \leq p^{(k)} \leq N_R^{(k)}$. Firstly, we note that for the positive semidefinite matrix $\mathbf{G}_R$ if $G_{i,i} = 0$, due to the condition of positive semidefiniticity, $G_{i,j} = G_{j,i} = 0$ for all j. So, at most 1 element on the diagonal of $\mathbf{G}_R$ may be zero. To see why this is so, suppose n diagonals of $\mathbf{G}_R$ are zero. Then re-order things so that all of the zero diagonals of $\mathbf{G}_R$ lie at the bottom right-hand of the matrix, thus:

$$\mathbf{G}_R = \begin{bmatrix} \mathbf{G}_{RX} & \mathbf{G}_{RY} \\ \mathbf{G}_{RY}^T & \mathbf{G}_{RZ} \end{bmatrix}$$

where $\mathbf{G}_{RZ} = \mathbf{0} \in \Re^{n \times n}$ and $\mathbf{G}_{RY} = \mathbf{0}$. But by theorem 6, this is not possible, so we conclude that at most one element on the diagonal of $\mathbf{G}_R$ may be zero. We assume that $G_{1,1} \neq 0$ (if this is not true, then it may be made so by appropriate re-ordering, so there is no loss of generality involved in making such an assumption), and also assume that $p^{(k)} = N_R^{(k)}$ (again, if this is not true, it may be made so by appropriate re-ordering).

Borrowing the notation of section IV-C, let us define:

$$\bar{\mathbf{H}}_R = \begin{bmatrix} g_{Ra} & d_{Ra} & \mathbf{g}_{Rba}^T \\ d_{Ra} & 0 & \mathbf{d}_{Rb}^T \\ \mathbf{g}_{Rba} & \mathbf{d}_{Rb} & \mathbf{G}_{Rb} \end{bmatrix}$$

It is not difficult to show that under these circumstances we may always calculate a lower triangular matrix $\bar{\mathbf{L}}$ with positive diagonals such that:

$$\bar{\mathbf{H}}_R = \bar{\mathbf{L}} \begin{bmatrix} 1 & 0 & \mathbf{0}^T \\ 0 & -1 & \mathbf{0}^T \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \bar{\mathbf{L}}^T$$

Partition $\bar{\mathbf{H}}_R$ and $\bar{\mathbf{L}}$ as follows:

$$\bar{\mathbf{H}}_R = \begin{bmatrix} \bar{\mathbf{H}}_{Rn} & \bar{\mathbf{h}}_{Rnl} \\ \bar{\mathbf{h}}_{Rnl}^T & \bar{h}_{Rl} \end{bmatrix}$$
$$\bar{\mathbf{L}} = \begin{bmatrix} \bar{\mathbf{L}}_n & \mathbf{0} \\ \bar{\mathbf{l}}_{nl}^T & \bar{l}_l \end{bmatrix}$$

But this implies that we can write:

$$\bar{\mathbf{H}}_{Rn} = \bar{\mathbf{L}}_n \begin{bmatrix} 1 & 0 & \mathbf{0}^T \\ 0 & -1 & \mathbf{0}^T \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \bar{\mathbf{L}}_n^T$$

where $\bar{\mathbf{L}}_n$ is a lower triangular matrix with positive diagonals, and so $\bar{\mathbf{H}}_{Rn}$ must be non-singular. Hence we may conclude that if $1 \leq p^{(k)} \leq N_R^{(k)}$ then $N_R^{(k+1)} \geq N_R^{(k)} - 1$.

So, in general, for all iterations $k \geq 1$ followed by the operation $\mathrm{constrain}\left(p^{(k)}, X; k\right)$, $N_R^{(k+1)} \geq N_R^{(k)} - 1$. ■

Appendix II

Setup Algorithms

Algorithms 1 and 2 are used to calculate $\beth$ during the initialisation phase of the optimisation algorithm, and also to update $\beth$ when the active set is modified. The algorithms themselves are very slight variants of standard algorithms from the literature (for example, [12]), and are therefore presented without commentary.

Algorithm 1: Calculate $\mathbf{V}$.
INVFACT($\mathbf{H}_F$)

(1) **if** $N_F = 0$
(2) $N_R := 0$
(3) $\mathbf{V} :=$ empty.
(4) **return**
(5) **if** $N_R = 0$
(6) $N_R := 1$
(7) $\mathbf{V} := \begin{bmatrix} -G_{1,1} & d_1 \\ d_1 & 0 \end{bmatrix}$
(8) **while** $N_R < N_F$
(9) $\begin{bmatrix} s_b \\ s_{\boldsymbol{\alpha}} \end{bmatrix} := \mathbf{V} \begin{bmatrix} d_{FB} \\ \mathbf{g}_{FBN} \end{bmatrix}$
(10) $\mathbf{h}_{FB} := \begin{bmatrix} d_{FB} \\ \mathbf{g}_{FBN} \end{bmatrix}$
(11) $x := g_{FB} - \mathbf{h}_{FB}^T \begin{bmatrix} s_b \\ s_{\boldsymbol{\alpha}} \end{bmatrix}$
(12) **if** $|x| < \varepsilon$ **then return**
(13) $\mathbf{z} := \begin{bmatrix} s_b \\ s_{\boldsymbol{\alpha}} \\ 1 \end{bmatrix}$
(14) $\mathbf{V} := \begin{bmatrix} \mathbf{V} & \mathbf{0} \\ \mathbf{0}^T & 0 \end{bmatrix} + \frac{1}{x}\mathbf{z}\mathbf{z}^T$
(15) $N_R := N_R + 1$
(16) **return**

Algorithm 2: Calculate $\mathbf{L}$.
CHOLFACT($\mathbf{H}_F$)

(1) **if** $N_F = 0$
(2) $N_R := 0$
(3) NoChol := FALSE
(4) $\mathbf{L} :=$ empty.
(5) **return**
(6) **if** $N_R = 0$ or NoChol = TRUE
(7) **if** $G_{1,1} < \varepsilon$
(8) **if** $G_{2,2} < \varepsilon$ or $N_F = 1$
(9) $N_R := 1$
(10) NoChol := TRUE
(11) $\mathbf{L} :=$ empty.
(12) **return**
(13) swap $(1, 2, F; k)$
(14) $N_R := 1$
(15) NoChol := FALSE
(16) $\mathbf{L} := \begin{bmatrix} \sqrt{G_{1,1}} & 0 \\ \frac{d_1}{\sqrt{G_{1,1}}} & \frac{1}{\sqrt{G_{1,1}}} \end{bmatrix}$
(17) **if** $N_F = 1$ **then return**
(18) **while** $N_R < N_F$
(19) Solve $\mathbf{L}\mathbf{a} = \begin{bmatrix} g_{FBNa} \\ d_{FB} \\ \mathbf{g}_{FBNb} \end{bmatrix}$ for $\mathbf{a}$.
(use forward elimination)
(21) $\mathbf{b} := \mathbf{J}\mathbf{a}$
(22) $c := g_{FB} - \mathbf{a}^T\mathbf{b}$
(23) **if** $c < \varepsilon$ **then return**
(24) $c := \sqrt{c}$
(25) $\mathbf{L} := \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{b}^T & c \end{bmatrix}$
(26) $N_R := N_R + 1$
(27) **return**

REFERENCES

[1] V. Vapnik, *The Nature of Statistical Learning Theory*. Springer-Verlag, New York, 1995.

[2] R. Fletcher, *Practical Methods of Optimisation*. John Wiley and Sons, Chichester, 1981.

[3] T. F. Coleman and L. A. Hulbert, "A direct Active Set Method for Large Sparse Quadratic Programs with Simple Bounds," *Mathematical Programming*, vol. 45, pp 373-406, 1989.

[4] A. Sanderson and A. Shilton, *Image Tracking and Recognition with Applications to Fisheries - An Investigation of Support Vector Machines*, Final year honours thesis, Department of Electrical and Electronic Engineering, The University of Melbourne, 1999.

[5] M. Palaniswami, A. Shilton, D. Ralph and B. D. Owen, "Machine Learning using Support Vector Machines," in *Proceedings of Interna-*

tional Conference on Artificial Intelligence in Science and Technology, AISAT2000, 2000.

[6] A. Shilton, M. Palaniswami, D. Ralph and A. C. Tsoi, "Incremental Training of Support Vector Machines," in *Proceedings of International Joint Conference on Neural Networks, IJCNN'01*, 2001, CD version only.

[7] G. Cauwenberghs and T. Poggio, "Incremental and Decremental Support Vector Machine Learning," in *NIPS*, 2000, pp 409-415. Available: http://citeseer.nj.nec.com/cauwenberghs00incremental.html.

[8] S. Fine and K. Scheinberg, "Incremental Learning and Selective Sampling via Parametric Optimization Framework for SVM," in *Advances in Neural Information Processing Systems 14*, T. G. Dietterich and S. Becker and Z. Ghahramani, ed., MIT Press, Cambridge, MA, 2002.

[9] S. Fine and K. Scheinberg, "Efficient Implementation of Interior Point Methods for Quadratic Problems arising in Support Vector Machines," Paper presented at NIPS2000

[10] C. J. C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Knowledge Discovery and Data Mining*, vol. 2(2), pp. 121-167, 1998.

[11] C. Cortes and V. Vapnik, "Support Vector Networks," *Machine Learning*, vol. 20, pp 273-297, 1995.

[12] G. H. Golub and C. F. Van Loan, *Matrix Computations.* John Hopkins University Press, Baltimore, 1983.

[13] J. Wilkinson, "A Priori Error Analysis of Algebraic Processes," in *Proc. International Congress Math.*, 1968, pp 629-639.

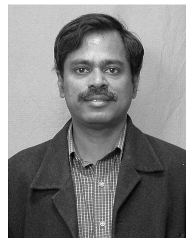
[14] P. E. Gill, G. H. Golub, W. Murray and M. Saunders, "Methods for Modifying Matrix Factorizations," *Mathematics of Computation*, vol. 28(126), pp 505-535, April 1974.

[15] O. Chapelle, V. Vapnik, O. Bousquet and S. Mukherjee, "Choosing Multiple Parameters for Support Vector Machines," *Machine Learning*, vol. 46, pp 131, 2002.

[16] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," Technical Report 98-14, Microsoft Research, Redmond, Washington, April 1998. citeseer.nj.nec.com/platt98sequential.html

[17] Blake, C.L. and Merz, C.J., *UCI Repository of machine learning databases*, http://www.ics.uci.edu/ mlearn/MLRepository.html, University of California, Department of Information and Computer Science, 1998.

Alistair Shilton received his combined B.Sc. / B.Eng. degree from the University of Melbourne, Melbourne, Australia, in 2000, specialising in physics, applied mathematics and electronic engineering. He is currently pursuing the Ph.D. degree in electronic engineering at the University of Melbourne. His research interests include machine learning, specializing in support vector machines; signal processing, communications and differential topology.



M. Palaniswami received his BE(Hons) from the University of Madras, ME from the Indian Institute of science, India, MEngSc from the University of Melbourne and Ph.D from the University of Newcastle, Australia before rejoining the University of Melbourne. He has been serving the University of Melbourne for over 16 years. He has published more than 180 refereed papers and a huge proportion of them appeared in prestigious IEEE Journals and Conferences. He was given a Foreign Specialist Award by the Ministry of Education, Japan in recognition of his contributions to the field of Machine Learning. He served as associate editor for Journals/transactions including IEEE Transactions on neural Networks and Computational Intelligence for Finance.. His research interests include SVMs, Sensors and Sensor Networks, Machine Learning, Neural Network, Pattern Recognition, Signal Processing and Control. He is the Co-Director of Centre of Expertise on Networked Decision & Sensor Systems. He is the associate editor for International Journal of Computational Intelligence and Applications and serves on the editorial board of ANZ Journal on Intelligent Information processing Systems. He is also the Subject Editor for International Journal on Distributed Sensor Networks.

Daniel Ralph obtained his B.Sc. (Hons) from The University of Melbourne, Australia, and his M.S. and Ph.D. from the University of Wisconsin-Madison, USA. He is now a senior lecturer at Cambridge University, UK, subsequent to lecturing at The University of Melbourne for seven years.

His research interests broadly relate to analysis and algorithms in nonlinear programming and nondifferentiable systems. Quadratic programming methods are a special interest, including their application to machine learning, discrete-time optimal control, and model predictive control. He has published numerous refereed papers, and co-authored a research monograph on an area of bilevel optimization called mathematical programming with equilibrium constraints. His editorial board memberships in recent years include Mathematical Programming (Series B), Mathematics of Operations Research, SIAM J on Optimization and The ANZIAM Journal. Conference activities, apart from invited lectures and session organization, include co-organising the 2002 International Conference on Complementarity Problems (Cambridge University). He has been the recipient of a number of research grants from the Australian Research Council.

Ah Chung Tsoi was born in Hong Kong. He received the Higher Diploma in Electronic Engineering from Hong Kong Technical College in 1969, and the M.Sc. degree in electronic control engineering, and Ph.D. degree in control engineering from University of Salford in 1970 and 1972 respectively. He also received the B.D. degree from University of Otago in 1980.

From 1972 to 1974 he was a Senior Research Fellow at the Inter-University Institute of Engineering Control, University College of North Wales, Bangor, Wales. From 1974 to 1977 he was a Lecturer at the Paisley College of Technology, Paisley Refrewshire, Scotland. From 1977 to 1985 he was a Senior Lecturer at the University of Auckland, New Zealand. From 1985 to 1990, he was a Senior Lecturer at the University College, University of New South Wales. From 1990 to 1996, he was an Associate Professor, and then a Professor in Electrical Engineering at the University of Queensland, Australia. Since July 1996 he has been at the University of Wollongong, where he had been Dean, Faculty of Informatics (1996 to 2001), and Director of Information Technology Services (1999 to 2001). Since February 2001, he is Pro Vice Chancellor (Information Technology and Communications) at the University of Wollongong. In the role of Pro Vice-Chancellor (Information Technolgoy and Communications) he has initiated a number of infrastructure projects: gigabit ethernet for the campus, voice over IP telephony, wireless access for the entire campus, implementation of a student management package. His research interests include aspects of neural networks and fuzzy systems and their application to practical problems, adaptive signal processing and adaptive control including data mining, internet search engine designs. In Feb 2004 he took up the position of Executive Director, Mathematics, Information and Communications Sciences, Australian Research Council (ARC).

He chaired the ARC Expert Advisory Committee on Mathematics, Information and Communications Sciences in 2001 and 2002. He chaired the ARC Linkage Infrastructure Equipment and Facilities Committee for two years. He was a committee member of the ARC Library Advisory Committee.

He received the Outstanding Alumni Award, Hong Kong Polytechnic University in 2001.