Numerical Methods in Kernel Machines

Ulrik Bernhardt Danielsen and Alexander Johansen Ohrt

June 11, 2022

Contents

1 Abstract

This report contains the final practical work in the course Machine Learning at Universitat Politècnica de Catalunya. The objective of this work is to describe the problem of computational complexity for kernel machines, both why the problem arises and some ways to deal with them. Some techniques will be more general, while others will be specific to specific kernel machines.

2 Introduction and previous work

Many common learning algorithms can and have been kernelized. By reformulating the problems by use of inner products between datapoints, a well chosen kernel can often transform the data into a feature space where better predictions or inference can be made. Unfortunately these kernelized methods grow in computational complexity as the data grows larger. In the time of *Big Data*, this poses a real problem.

Plenty of previous work has been done on the manner of making kernelized methods computationally feasible with growing amounts of data. For example, matrix-free iterative methods can be used to work with the Gram matrix in more efficient ways. A specific example of a method that can be used for tackling scalability issues of kernel machines is the conjugate gradient algorithm. This algorithm alleviates the need for storage, since the Gram matrix need not be stored explicitly, as well as improves computational speed, since stochastic gradients and parallelization can be used. However, the condition number $\kappa(K)$ of the Gram matrix is often large, which means that the algorithm needs some remedy. Preconditioning for kernel machines is discussed in [2].

Other ways of making kernel machines less computationally costly are adapting them to be trained on graphics processing units (GPUs). Most modern machine learning models facilitate computations on a GPU, where the models are often trained using Stochastic Gradient Descent (SGD). The training time of a kernel machine can be decreased when given access to a parallel computational resource (like a GPU) by learning a new data and computational resource dependent kernel. An algorithm that modifies the optimization procedure of kernel machines without changing the learned predictor function, extending linear scaling to match the computational power of a GPU, is developed in [10]. More on so-called linear scaling of SGD for convex loss functions can be found in [9]. A solver for Support Vector Machines (SVMs) when training on a GPU is described in [15] . Another discussion of how SVM training can be efficiently implemented on a GPU, for a wide range of kernels, for both binary and multiclass variants, is given in [1]. The implementation that is presented also takes advantage of sparsity in the training set, setting it apart from some other GPU SVM solvers.

3 Background

3.1 Kernel machines

A *kernel* is a positive semi-definite function mapping two arguments (often vectors) from a space $\mathcal{X}$ onto the real line, $k(x, x') : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Given a dataset $\mathcal{D} = \{x_1, \ldots, x_n\}$ the kernel gives rise to the $n \times n$ *Gram matrix* $K_{ij} = k(x_i, x_j)$. As the kernel is PSD so is K. Methods which use the kernel matrix K are called *kernel machines*, and can by construction be tweaked by changing the chosen kernel function and its hyper parameters. In fact, the performance of these methods are highly dependent on the choice of kernel (and thus Gram matrix), since this decides the feature space that is utilized. The kernel implicitly sets the feature space being used; another way of viewing a kernel function is $k(x, x') = \langle \phi(x), \phi(x') \rangle$, where $\phi(\cdot)$ is the implicit feature space mapping. Notice that Mercer (1909) showed that a necessary and sufficient condition for a symmetric function to be a kernel, i.e. to give existence of a corresponding feature space, is that it is positive definite [5].

3.2 Important Classes of Kernels

Computational reduction for kernel-based methods can be provided by using different classes of kernels, for instance compactly supported kernels and separable nonstationary kernels [5]. These types of kernels will be described here.

Compactly supported kernels are kernels that vanish whenever the distance between two vectors is larger than a certain distance d. This distance is often called the range of the kernel [5]. The usage of such kernels can be a great advantage when performing computations in kernel machines, because the vanishing kernel for distances larger than d yields a sparse Gram matrix. This can lead to reduced storage cost and computational cost, because one can solve linear systems involving the matrix with sparse linear algebra techniques. The downfall with this technique is that one has to use compactly supported kernels to yield the performance boost. Fortunately, large classes of such kernels can be constructed [5]. Hamers et al [7] investigate the use of compactly supported Gaussian RBF kernels in least-squares support vector machines (LS-SVM). Notice that in these machines one finds the solution by solving a set of linear equations in stead of a convex quadratic programming problem, which is solved in regular SVMs. This leads to the possibility of using different standard numerical methods for solving linear systems, like Cholesky factorization and Krylov methods (e.g. conjugate gradient), where their respective memory requirements and computational costs can be reduced by the use of compactly supported Gaussian RNF kernels [7].

Separable nonstationary kernels are a combination of two classes of kernels: *nonstationary* and *separable*. The most general class of kernels is the class of nonstationary kernels. These kernels depend explicitly on the two input arguments. This is in contrast to *stationary* kernels, which only depend on the vector separating the two arguments (*anisotropic stationary* kernels) or simply the norm of the vector separating the two arguments (*isotropic stationary* kernels). An example of a nonstatinary kernel is the well-known polynomial kernel $k(x, x') = (x^T x' + b)^p$, $b \in \mathbb{R}$, of polynomial degree p. The separable nonstationary kernels are a particular family of nonstationary kernels, that can be written on the form $k(x, x') = k_1(x)k_2(x')$, where k_1 and k_2 are stationary kernels. These can be used to reduce computational burden because the Gram matrix they give rise to can be written as a Kronecker product (a generalization of the outer product from vectors to matrices) of two vectors defined by k_1 and k_2 respectively [5]. For instance, consider again the dataset $\mathcal{D} = \{x_1, \ldots, x_n\}$. Instead of having to evaluate $K_{ij} = k(x_i, x_j)$, $, \forall x_i, x_j \in \mathcal{D}$ it suffices to evaluate $k_1(x_i)$ $, \forall x_i \in \mathcal{D}$ and $k_2(x_i)$, $\forall x_i \in \mathcal{D}$, since the Gram matrix of a separable nonstationary kernel can be calculated by an outer product of two column vectors produced by these evaluations. This leads to a reduction in the number of evaluations required from n^2 to $2n$.

3.3 Gaussian Processes

Gaussian processes (GP) can be defined in multiple ways. In our case it is beneficial to view it as a distribution over functions. Rasmussen and Williams [13] provide the definition *a Gaussian process is a collection of random variables, any finite number of which gave a joint Gaussian distribution.* The processes take advantage of the many convenient properties of the Gaussian distribution. Mathematically, letting $m(x) = E[f(x)]$ be the mean function and $k(x, x') = \operatorname{cov}(f(x), f(x')) = E[(f(x) - m(x))(f(x') - m(x'))]$ the covariance function, a Gaussian process can be written as

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')).$$

Note that the similar notation for kernel functions and covariance functions is not a coincidence; the covariance functions are user defined and tuned in a similar fashion as kernel functions. In GPs the mean function is usually set to be zero, which means that the process is completely described by its covariance function.

A common use of GPs is in regression. We have access to n data pairs $(x_1, y_1), \ldots, (x_n, y_n)$ where y_i is the function evaluation at x_i with added noise; $y_i = f(x_i) + \varepsilon$. As usual assuming iid Gaussian noise ε with variance σ_n^2, the covariance matrix of the observed values is $\operatorname{cov}(y) = K + \sigma_n^2 I$. Our interest lies in predicting new function values f_* at x_*, for simplicity assumed to be a single point. Assuming zero mean function, the joint distribution is

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K + \sigma_n^2 I & k_* \\ k_*^T & k(x_*, x_*) \end{bmatrix}\right).$$

Here $k_* = k(x, x_*)$ is the vector of covariances between the data and the new point x_*. We are after the conditional distribution of f_*, which has the known form

$$f_* | x, y, x_* \sim \mathcal{N}(\bar{f}_*, V[f_*])$$

where

$$\bar{f}_* = k_*^T(K + \sigma_n^2 I)^{-1} y \tag{1}$$

$$V[f_*] = k(x_*, x_*) - k_*^T(K + \sigma_n^2 I)^{-1} k_*. \tag{2}$$

To do model selection, i.e. tuning of the hyper parameters in the chosen covariance function, we also state the log marginal likelihood of y,

$$\log p(y|x) = -\frac{1}{2} y^T (K + \sigma_n^2 I)^{-1} y - \frac{1}{2} \log |K + \sigma_n^2 I| - \frac{n}{2} \log 2\pi. \tag{3}$$

It is now apparent that in order to do GP regression we need to invert $(K + \sigma_n^2 I)$—or for prediction solve the system $(K + \sigma_n^2 I)v = y$ for v. For smaller n this can be done by means of Cholesky factorization, but becomes inefficient as n grows large.

3.4 Support Vector Machines

Support Vector Machines (SVMs) is a popular machine learning technique commonly used for classifying data into two groups. Again we have n data samples $(x_1, y_1,), \ldots, (x_n, y_n)$, where $y_i \in \{1, -1\}$ now is a binary variable, and $x_i \in \mathbb{R}^d$. We classify the points through a separating hyperplane $w^T x + w_0$, such that the classifying function is $f_w(x) = \text{sgn}(w^T x + w_0)$. The goal is to find the "best" w, w_0, i.e., finding the optimal hyperplane. The SVM finds this hyperplane by solving the optimization problem [8]

$$\begin{array}{ll} \min\limits_{w} & \frac{1}{2}||w||^2 \\ \text{s.t.} & y_i(w^T x_i + w_0) \geq 1. \end{array}$$

This is a quadratic programming problem with linear constraint. It can be solved directly, but it is practical to make use of Lagrangian duality theory and instead work with its dual representation

$$\begin{array}{ll} \max\limits_{\alpha \in \mathbb{R}^n} & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j \\ \text{s.t.} & \alpha_i \geq 0, \\ & \sum_{i=1}^n \alpha_i y_i = 0. \end{array} \tag{4}$$

It is worth noticing that we now have to optimize for n parameters $\alpha_1, \ldots, \alpha_n$, not d as in the primal formulation. This is still a linear classifier—where are the kernels? Noticing that the data points only appear in the methods as parts of an inner products (dot products as $x_i \in \mathbb{R}^d$), we can *kernelize* the method. I.e., we map the input points to a high dimensional space through the function $\phi : \mathbb{R}^d \to \mathbb{R}^D, \ D \gg d$. The hyperplane is then created in $\mathbb{R}^D$ where the data might be separable. We choose ϕ in such a way that we know the function k where

$$\begin{array}{l} k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \\ x, x' \mapsto k(x, x') = \langle \phi(x), \phi(x') \rangle. \end{array}$$

The model is now on the form $f_\alpha(x) = \text{sgn}(\sum_{i=1}^n y_i \alpha_i k(x, x_i) + \alpha_0)$ where α is found though the quadratic optimization problem

$$\begin{array}{ll} \max\limits_{\alpha \in \mathbb{R}^n} & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ \text{s.t.} & \alpha_i \geq 0, \\ & \sum_{i=1}^n \alpha_i y_i = 0. \end{array}$$

When k is positive definite this is a convex problem, which when n is large can be solved more efficiently using numerical techniques. Also note that the theory on SVMs can be developed further to include soft margin hyperplanes, regression, multiclass models, etc.

3.5 Condition number

The *condition number* of a matrix A is defined as $\kappa(A) = ||A|| ||A^{-1}||$ [12]. Note that the norm $|| \cdot ||$ can be any consistent norm, but takes on a different form dependent on which one is chosen. From the definition it is hard to say exactly what the condition number represent. We often talk about the condition number in relation to the numerical properties of a linear system $Ax = b$. The system is considered *ill-conditioned* if $\kappa(A)$ is large, and *well-conditioned* if it is small (close to 1, its lower bound). Having a large condition number means that the linear system is *sensitive* to small perturbations. Meaning that if we change x by a small amount, the solution b could change a lot—and vice versa. This becomes a problem when trying to obtain numerical approximations to the systems, and can cause numerical solvers to perform poorly.

The most commonly used condition number is the one induced by the 2-norm. It can be shown that is has the form $\kappa_2(A) = ||A||_2||A^{-1}||_2 = \frac{\sigma_1(A)}{\sigma_n(A)}$ where $\sigma_1(A)$ and $\sigma_n(A)$ are the maximum and minimum singular value. More importantly—at least for us—is that if A is symmetric positive definite it has the form $\kappa_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$ where $\lambda_{\max}$ and $\lambda_{\min}$ is the largest and smallest eigenvalue of x. This expression provides a more intuitive view of the condition number. If the eigenvalues of a matrix is very similar, it scales vectors x by a similar amount. In the same way if some eigenvalues are very large, then vectors close to their respective eigenvectors will be scaled more than others. Thus some iterative algorithms can face problems with some directions and act more unstable.

4 Computational Feasibility

The problem this work is intended to describe and tackle is the problem of computational feasibility for kernel machines. A simple evaluation of every element in the Gram matrix is $\mathcal{O}(n^2)$ operations. Since this is obviously not the only computation that needs to be done when using these methods, they can often become computationally expensive. In order to remedy this problem one possible solution is using *Matrix-free* methods, which in general do not need the entire Gram matrix explicitly. Instead they work by calculating matrix-vector products Kx for arbitrary x [3]. Some examples of methods are *conjugate gradient*, *power iteration* and *Lanczos algorithm* [16]. These methods are iterative, which means that their numerical stability and convergence rates are related to the condition number of the Gram matrix $\kappa(K)$. Low values of the condition number means that the problem is well-conditioned, which implies numerically stable calculations in few iterations. Large values of the condition number means the the problem is ill-conditioned—this leads to slow convergence or no convergence at all. Pre-conditioning can be used to ensure that $\kappa(K)$ stays low, leading to efficient and accurate iterative methods [3].

5 Condition Number of the Gram Matrix

https://scicomp.stackexchange.com/questions/7561/do-rbf-kernel-matrices-tend-to-be-ill-conditioned

6 Interior point methods

Many kernel machines require solving of a quadratic optimization problems with linear constraints. We have already seen an example in the case of SVMs where we have to solve 4 for α. Interior point methods are a class of optimization methods where we apply convex optimization techniques to the system of equations derived by KKT theory. The methods are most commonly applied to linear problems, but can also be tweaked to work on quadratic problems (or convex problems in general). They are specially effective on semi-definite problems, which always apply to problems in kernel machines.

6.1 Primal-dual path following

The most commonly used interior point method is the primal-dual path following method. A general form of the quadratic problem which we wish to solve is

$$\begin{aligned} \min_x \quad & c^T x + \frac{1}{2} x^T Q x \\ \text{s.t.} \quad & Ax = b, \\ & x \geq 0 \end{aligned}$$

Letting λ and s be the dual variables concerning the equality and inequality constraints respectively, the KKT optimality conditions are

$$\begin{aligned} & A^T \lambda - Qx + s = c \\ & Ax = b \\ & XSe = 0 \\ & x, s \geq 0. \end{aligned}$$

Here X and S are diagonal matrices with the vectors x and s on the diagonal, and e is the vector $(1 \ \dots \ 1)^T$. Defining the function

$$F(x, \lambda, s) = \begin{bmatrix} A^T \lambda - Qx + s - c \\ Ax - b \\ XSe \end{bmatrix}$$

we have an equivalent non-linear system of equations $F(x, \lambda, s) = 0$, which we can solve using familiar techniques such as the Newton-Raphson method. Unfortunately solving this system as is works poorly in practice [11]. Instead we solve the slightly different perturbed system

$$F(x, \lambda, s) = \begin{bmatrix} A^T \lambda - Qx + s - c \\ Ax - b \\ XSe - \tau e \end{bmatrix} = 0.$$

This has the effect of "staying away" from the borders of the feasible region, and instead move towards the solution through the interior, hence interior point methods. In contrast the simplex method moves towards the solution *only* along the border of the feasible region. Interior point methods are often chosen for large problems.

There exists much theory solving this perturbed system efficiently, involving breaking it down into several steps where the structure of the involved matrices are taken advantage of, e.g. by factorization methods, to solve it efficiently. Jack Gondzio [6] proposes a entirely matrix free implementation of an interior point method which can be used on quadratic programming problems.

7 Conjugate gradient method

The conjugate gradient method solves linear systems on the form $Ax = b$. This is a common part of many kernel machines, e.g. in GP regression which we have seen needs the solution to $(K + \sigma_n^2 I)v = y$ to solve (1). The algorithm works by treating the system as an optimization problem where we want to minimize

$$\phi(y) = \frac{1}{2} y^T A y - y^T b, \tag{5}$$

which has its optimal solution at x. One of the most common ways of numerically solving such an optimization problem is by choosing an initial point $x^{(0)}$ and iteratively compute descent directions $d^{(k)}$ to find $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$. *Steepest descent* chooses $d^{(k)} = -\nabla \phi(x^{(k)}) = b - Ax^{(k)}$, and since ϕ is quadratic it can be shown [12] that the optimal α_k is given by

$$\alpha_k = \frac{(d^{(k)})^T d^{(k)}}{(d^{(k)})^T A d^{(k)}}.$$

It can be shown that the convergence speed of steepest descent descent is bounded by the condition number of A, $\kappa(A)$, which for symmetric psd matrices is the ratio between the largest and smallest eigenvalue, $\lambda_{(n)}/\lambda_{(1)}$ [14]. This can be a problem if A is ill-conditioned and the steepest descent method needs many iterations to converge.

The details and motivation behind the conjugate gradient algorithm are quite intricate and dense. In essence we notice the properties of the directions of the steepest descent method where it converges quickly, and impose orthogonality restrictions on the descent directions to make the algorithm converge in at most n iterations. By making sure every direction is optimal we can get very close to the optimal solution without completing all of the n iterations. For a detailed and aptly named derivation of the conjugate gradient algorithm consult *An introduction of the conjugate gradient method without the agonizing pain* [14] by Jonathan Richard Shewchuk.

The benefit of using the CG algorithm or steepest descent is that we replace the $\mathcal{O}(n^3)$ linear systems my matrix-vector products which can be solved in $\mathcal{O}(m)$ time where m is the number of non-zero elements in A. Both steepest descent and CG has a space complexity $\mathcal{O}(m)$ and time complexity $\mathcal{O}(m\kappa)$ and $\mathcal{O}(m\sqrt{\kappa})$ respectively [14].

8 Preconditioning

Preconditioning a matrix A refers to applying a *preconditioner* P so that $\kappa(P^{-1}A) \ll \kappa(A)$. [2] Many iterative methods make use of this by solving the preconditioned system $P^{-1}Ax = P^{-1}b$ instead of $Ax = b$. This is done in the popular preconditioned conjugate gradient method. Note that in many algorithms P^{-1} does not need to be known explicitly. Quarteroni et al [12] divides preconditioners into two main categories: algebraic and functional preconditioners. They list diagonal preconditioners, polynomial preconditioners, least-squares preconditioners and incomplete LU/Cholesky factorization as examples.

8.1 Nyström approximation

The goal of the Nyström approximation is not to reduce the condition number of K, but the rank. We choose $m < n$ inducing points, and approximate K by $\tilde{K} = K_{nm}K_{mm}^{-1}K_{mn}$ [13]. K_{mm} is the evaluation of the kernel function over the chosen induced points, and K_{nm} is the evaluation between the original n points. Now the approximation has rank $\leq m$, and the inverse can be computed in $\mathcal{O}(m^3)$ which is significantly lower than $\mathcal{O}(n^3)$.

As for which m inducing points to choose Drineas and Mahoney [4] suggests drawing uniformly random samples with replacement, while Rasmussen and Williams discusses a greedy algorithm for selecting the points. These greedy algorithms generally start from an empty set of chosen indices I, and iteratively selects the best point to include by some chosen criterion.

8.2 Subset of regressors

A popular approximation to GP regression is achieved by only choosing a subset of the regressors. It can be shown that the mean predictor in GPs 1 can be achieved by considering the model $f(x_*) = \sum_{i=1}^{n} \alpha_i k(x_*, x_i)$ with prior $\alpha \sim \mathcal{N}(0, K^{-1})$ [13]. As in the Nyström approximation we choose a subset of m data points, and approximate the model by

$$f_{SR}(x_*) = \sum_{i=1}^{m} \alpha_i k(x_*, x_i), \qquad \alpha_m \sim \mathcal{N}(0, K_{mm}^{-1}).$$

Just as before we apply the properties of the Gaussian distribution to obtain

$$\begin{aligned} \bar{f}_{SR}(x_*) &= k_m(x_*)^T (K_{mn}K_{nm} + \sigma_n^2 K_{mm})^{-1} K_{mn} y \\ V[f_{SR}(x_*)] &= \sigma_n^2 k_m(x_*)^T (K_{mn}K_{nm} + \sigma_n^2 K_{mm})^{-1} k_m(x_*). \end{aligned}$$

The matrix computations in the stated equations take $\mathcal{O}(m^2 n)$ time to carry out, but after they are computed the prediction of new test points are linear in m. In this case, a criterion for the greedy selection of the m points could be the residual sum of squares $|y - K_{nm}\alpha_m|^2$.

9 Conclusion

The Gram matrix in kernel machines has good and bad qualities when used in numerical approximations. Firstly—it is large, which is good. This means that traditional solutions requires much from our computer in terms of memory allocation and speed. Secondly—it is symmetric positive semi-definite. Having this property make the numerical computations much easier, and we can make use of a wide range of methods having this as a criteria. For instance, in SVMs the quadratic programming problem is convex as a consequence.

10 Future Work

In this work we looked how numerical approximations can help when doing GP regression and SVMs for classifications. Obviously there are many more kernel machines, many more kernels and many more methods of numerical approximation out there. Numerical methods are a field of its own, and is all about tweaking known methods like Cholesky factorization or Newton-Raphson to suit the specific problems. In that sense numerical approximation for kernel machines is a two part problem, first understand the properties of the kernel machine, and then find a way to use those properties in the numerical approximation. Given the wide range of kernel machines and applications, specific problems are plentiful. There are much more to read and understand, and much active research.

References

[1] Andrew Cotter, Nathan Srebro, and Joseph Keshet. "A GPU-tailored approach for training kernelized SVMs". In: *KDD*. 2011.

[2] Kurt Cutajar et al. *Preconditioning Kernel Matrices*. 2016. DOI: `10.48550/ARXIV.1602.06693`. URL: `https://arxiv.org/abs/1602.06693`.

[3] Alex Davies and Zoubin Ghahramani. *The Random Forest Kernel and other kernels for big data from random partitions*. 2014. DOI: `10.48550/ARXIV.1402.4293`. URL: `https://arxiv.org/abs/1402.4293`.

[4] Petros Drineas and Michael W. Mahoney. "On the Nyström Method for Approximating a Gram Matrix for Improved Kernel-Based Learning". In: *JOURNAL OF MACHINE LEARNING RESEARCH* 6 (2005), p. 2005.

[5] Marc G. Genton. "Classes of Kernels for Machine Learning: A Statistics Perspective". In: *J. Mach. Learn. Res.* 2 (Mar. 2002), pp. 299–312. ISSN: 1532-4435.

[6] J. Gondzio. *Matrix-free interior point method*. 2012.

[7] Bart Hamers, Johan Suykens, and Bart De Moor. "Compactly Supported RBF Kernels for Sparsifying the Gram Matrix in LS-SVM Regression Models". In: vol. 2415. Aug. 2002, pp. 720–726. ISBN: 978-3-540-44074-1. DOI: `10.1007/3-540-46084-5_117`.

[8] Jyrki Kivinen, Alex J. Smola, and Robert C. Williamson. *Learning With Kernels*. 2002.

[9] Siyuan Ma, Raef Bassily, and Mikhail Belkin. *The Power of Interpolation: Understanding the Effectiveness of SGD in Modern Over-parametrized Learning*. 2017. DOI: `10.48550/ARXIV.1712.06559`. URL: `https://arxiv.org/abs/1712.06559`.

[10] Siyuan Ma and Mikhail Belkin. *Kernel machines that adapt to GPUs for effective large batch training*. 2018. DOI: `10.48550/ARXIV.1806.06144`. URL: `https://arxiv.org/abs/1806.06144`.

[11] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. English (US). Springer Series in Operations Research and Financial Engineering. Springer Nature, 2006, pp. 1–664.

[12] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*. Springer, 2010.

[13] Carl Edward Rasmussen and Williams Christopher K I. *Gaussian processes for machine learning*. MIT Press, 2008.

[14] Jonathan Richard Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. Tech. rep. 1994.

[15] Narayanan Sundaram and Bryan Catanzaro. *Accelerating Machine Learning Applications on Graphics Processors.*

[16] Wikipedia. *Matrix-free methods — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Matrix-free%20methods&oldid=1052656767`. [Online; accessed 25-May-2022]. 2022.