Krylov Subspace Recycling for
Fast Iterative Least-Squares in Machine Learning

Filip de Roos and Philipp Hennig
Max Planck Institute for Intelligent Systems Spemannstr. 34, Tübingen, Germany
[fderoos|ph]@tue.mpg.de

Abstract

Solving symmetric positive definite linear problems is a fundamental computational task in machine learning. The exact solution, famously, is cubically expensive in the size of the matrix. To alleviate this problem, several linear-time approximations, such as spectral and inducing-point methods, have been suggested and are now in wide use. These are low-rank approximations that choose the low-rank space a priori and do not refine it over time. While this allows linear cost in the dataset size, it also causes a finite, uncorrected approximation error. Authors from numerical linear algebra have explored ways to iteratively refine such low-rank approximations, at a cost of a small number of matrix-vector multiplications. This idea is particularly interesting in the many situations in machine learning where one has to solve a sequence of related symmetric positive definite linear problems. From the machine learning perspective, such *deflation* methods can be interpreted as transfer learning of a low-rank approximation across a time-series of numerical tasks. We study the use of such methods for our field. Our empirical results show that, on regression and classification problems of intermediate size, this approach can interpolate between low computational cost and numerical precision.

1   Introduction

Many of the most prominent machine learning problems can be seen as a sequence of linear systems, finding $\boldsymbol{x}^{(i)}$ such that

$$A^{(i)}\boldsymbol{x}^{(i)} = \boldsymbol{b}^{(i)} \qquad \text{for} \qquad A^{(i)} \in \mathbb{R}^{n \times n}, \quad \boldsymbol{x}^{(i)}, \boldsymbol{b}^{(i)} \in \mathbb{R}^n, \qquad \text{and} \qquad i \in \mathbb{N}. \qquad (1)$$

A prominent example is nonparametric logistic regression, further explained below. But there are many more: Model adaptation in Gaussian process models [23, §5.2] requires the solution of the problem $k_{\theta,XX}^{-1}\boldsymbol{y}$ for a sequence of parameter estimates θ, where k_{XX} is a kernel Gram matrix over the data set $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]^{\mathsf{T}}$, and $\boldsymbol{y}$ is the vector of target data. Further afield, although this view is not currently the popular standard, deep learning tasks have in the past been addressed by methods like Hessian-free optimization [9], which consist of such sequences. Importantly, in these machine learning examples, the matrix A is usually symmetric positive definite (or, in the generally non-convex case of deep learning, is at least approximated by an spd matrix in the optimizer to ensure a descent step). This means that Eq. (1) is indeed an optimization problem, because $\boldsymbol{x}^{(i)}$ then equals the minimum of the quadratic function

$$\boldsymbol{x}^{(i)} = \underset{\tilde{\boldsymbol{x}}}{\arg\min}\, \frac{1}{2}\tilde{\boldsymbol{x}}^{\mathsf{T}}A^{(i)}\tilde{\boldsymbol{x}} - \tilde{\boldsymbol{x}}^{\mathsf{T}}\boldsymbol{b}^{(i)} \quad \text{with gradient} \quad \boldsymbol{r}^{(i)}(\tilde{\boldsymbol{x}}) := \nabla f(\tilde{\boldsymbol{x}}) = A^{(i)}\tilde{\boldsymbol{x}} - \boldsymbol{b}^{(i)}. \qquad (2)$$

When facing linear problems of small to moderate size (i.e. $n \lesssim 10^4$) in machine learning , the typical approach is to rely on standard algorithms (in particular, Cholesky decompositions) provided by toolboxes like BLAS libraries, or iterative linear solvers like the method of conjugate gradients [8]

(CG). Exact methods like the Cholesky decomposition have cubic cost, $\mathcal{O}(n^3)$, iterative solvers like CG have quadratic cost, $\mathcal{O}(n^2 m)$ for a small number of m iterative steps. These algorithms are self-contained, generic "black boxes"—they are designed to work on *any* spd matrix and approach every new problem in the same way. A more critical way to phrase this property is that these algorithms are *non-adaptive*. If the sequence of tasks $(A^{(i)}, \boldsymbol{b}^{(i)})$ are related to each other, for example because they are created by an outer optimization loop, it seems natural to want to propagate information from one linear solver to another. One can think of this notion as a form of *computational transfer learning*, in the "probabilistic numerics" sense of treating a computation as an inference procedure [7]. As it turns out, the computational linear algebra community has already addressed this issue to some extent. In that community, the idea of re-using information from previous problems in subsequent ones is known as *subspace-recycling* [14]. But these numerical algorithms have not yet found their way into the machine learning community. Below, we explore the utility of such a resulting method for application in machine learning, by empirically evaluating it on the test problem of Bayesian logistic regression (aka. Gaussian process classification, GPC).

1.1 Relation to Linear-Cost Methods

For linear problems of *large* dimensionality (data-sets of size $n \gtrsim 10^4$), the current standard approach is to introduce an "a-priori" low-rank approximation: Sampling a small set of approximate eigenvectors of the kernel gram matrix [17, 18], and introducing various conditional independence assumptions over sets of inducing points [24, 16, 21]. These methods achieve *linear* cost $\mathcal{O}(nm^2)$ because they project the problem onto a projective space of dimensionality m, and this space is not adapted over time (when it is adapted [22], additional computational overhead is created outside the solver). The downside of this approach is that it yields an approximation of finite quality—these methods fundamentally can not converge, in general, to the exact solution. The algorithms we consider below can be seen in some sense as the "missing link" between these linear-cost-but-finite-error methods and the cubic-cost, exact solvers for smaller problems: They *adapt* the projective sub-space over time at the cost of a small number of quadratic cost steps, while also attempting to re-use as much information from previous runs as possible. In fact, the "guessed" projective space of the aforementioned methods could be used as the first initialization of the methods discussed below.

2 Method

2.1 Krylov Subspace Recycling

The *Krylov sequence* of subspaces is a core concept of iterative methods for linear systems $A\boldsymbol{x} = \boldsymbol{b}$ [19]. The *Krylov subspace* $\mathcal{K}_j(A, \boldsymbol{r}_0)$ is the span of the truncated power iteration of A operating on $\boldsymbol{r}_0$, where $\boldsymbol{r}_0$ is the gradient (*residual*) from Eq. (2) at the starting point $\boldsymbol{x}_0$ of the optimization:

$$\mathcal{K}_j(A, \boldsymbol{r}_0) = \text{span}\{\boldsymbol{r}_0, A\boldsymbol{r}_0, ..., A^{j-1}\boldsymbol{r}_0\}. \tag{3}$$

Methods that *transfer* information from one such iteration to the next in order to faster converge to a solution in subsequent systems are referred to as *Krylov subspace recycling methods* [14]. The "recycling" of information is traditionally done by *deflation* [20, 4], *augmentation* [5, 12], or combinations thereof [3, 1]. These two approaches differ in their implementation, but have the same goal: restricting the solution to a simpler search space, to speed up convergence. Both store a set of k linearly independent vectors $W^{(i)} \in \mathbb{R}^{n \times k}$. For problem i in the above sequence of tasks, a subspace-recycling Krylov method computes solutions that satisfy

$$\boldsymbol{x}_j^{(i)} \in \boldsymbol{x}_0 + \mathcal{K}_j(A, \boldsymbol{r}_0) \cup \text{span}\{W^{(i)}\}, \tag{4}$$

$$\boldsymbol{r}_j^{(i)} = \boldsymbol{b} - A\boldsymbol{x}_j \perp \mathcal{K}_j(A, \boldsymbol{r}_0) \cup \text{span}\{W^{(i)}\}. \tag{5}$$

An augmented iterative solver keeps the vectors in W and orthogonalizes the updated residuals $\boldsymbol{r}_j^{(i)}$ against W. This method is easily included in methods that contain an explicit orthogonalization step, an example of which is the General Minimum Residual method (GMRES) [12].

For A spd, one usually chooses CG as the iterative solver and deflation is easier incorporated [20]. A deflated method "deflates" a part of the spectrum of A by projecting the solution onto the orthogonal complement of W. This can be viewed as a form of preconditioning[1] with a singular projector

---

[1]Not all authors agree: Gaul et al. [5] argue that the projector P_W should not be considered a preconditioner since its application removes a part of the spectrum of A while leaving the remainder untouched.
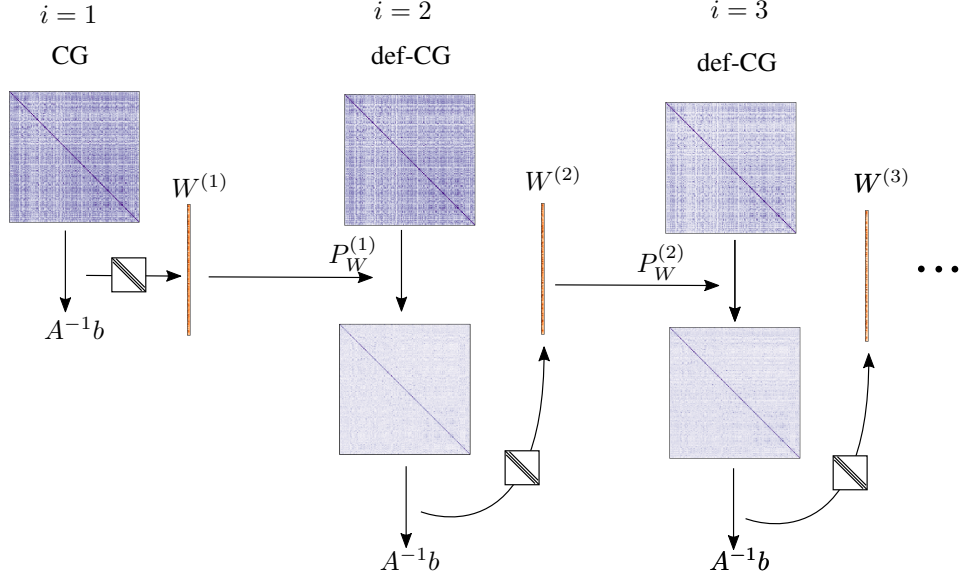
Figure 1: The def-CG algorithm applied to a sequence of linear systems with the implicit preconditioning visualized. The first solution is obtained through normal CG and approximate eigenvectors W, corresponding to the largest eigenvalues are calculated from a low-rank approximation. If prior knowledge about the eigenvectors of the first system is available, def-CG can be used for the first system as well.

P_W, i.e. solving $P_W^{(i)} A^{(i)} \boldsymbol{x}^{(i)} = P_W^{(i)} \boldsymbol{b}^{(i)}$. This property is in contrast to normal preconditioning where the whole spectrum of A is modified by a non-singular matrix. In order to keep the additional computational overhead incurred by subspace recycling low, the dimension of W should be low, and contain vectors that optimally speed up the convergence. For iterative linear solvers, the rate of convergence is directly proportional to the condition number

$$\kappa(A) = \frac{\lambda_n(A)}{\lambda_1(A)}, \tag{6}$$

where λ_j refers to the eigenvalues of A sorted in ascending order [13]. Typically, the smallest eigenvalues are the limiting factors for convergence; hence $W^{(i)}$ should ideally contain the eigenvectors related to the k *smallest* eigenvalues—this yields an effective condition number of $\kappa_{\text{eff}} = \lambda_n/\lambda_{k+1}$, which can drastically improve the convergence rate [19, 3]. Of course the same improvement in the condition number can also be achieved by changing the *largest* eigenvalue.

2.2 Deflated Conjugate Gradient

In the special case when A is symmetric and positive definite (SPD), the conjugate gradient method (CG) [8] is a popular choice to iteratively solve the system. As noted above, in machine learning, where linear tasks almost invariably arise in the form of least-squares problems, this is actually the typical setting. Saad et al. [20] derived a deflated version of CG based on the Lanczos iteration (partly represented in Algorithm 1): it implicitly forms a tri-diagonal low-rank approximation to a Hermitian matrix A, which translates to symmetric when A is real-valued. By storing quantities that are readily available from the CG iterations, the low-rank approximation can be obtained without costly matrix-vector computations. The eigenvalues of the approximation are known as Ritz values, which tend to approximate the extreme ends of the spectrum of A [19]. The corresponding Ritz vectors are used to find good approximations of the eigenvectors that can be used for a deflated subspace W to improve the condition number [see 20, for more details]. The algorithm will be referred to as def-CG(k, ℓ) where ℓ is the number of CG iterations from which information is stored in order to generate k approximate eigenvectors.

For deflated CG, the orthogonality constraint of Eq. (5) is replaced by a constraint of *conjugacy*, i.e. $\boldsymbol{r}_i^\mathsf{T} A \boldsymbol{r}_j = 0$ for $i \neq j$. The associated preconditioner $P_W = I - AW(W^T A W)^{-1} W^T$ projects

the residual onto the A-conjugate complement of W. Figure 1 illustrates the effect of applying P_W to A. For this figure, W was chosen by the def-CG algorithm according to the harmonic projection method [12], so the approximate eigenvectors correspond to the largest eigenvalues. The algorithm (Algorithm 1) differs from the standard method of conjugate gradients only in line 11 and the initialization in line 3. How the eigenvectors are approximated is outlined in Section 2.3. The additional inputs to the solver are a set of k linearly independent vectors in W and, optionally, W multiplied with A if it can be obtained cheaply.

To estimate the computational cost of def-CG we assume $k \ll n$ so terms not containing n in the computational complexity can be ignored. Each iteration in Algorithm 1 has a computational overhead of $\mathcal{O}(kn^2)$ of solving the linear system in line 11. Computing the matrix $W^T A W$ has complexity $\mathcal{O}(n^2 k^2)$ but it only has to be computed once and if the procedure used by Saad et al. [20] and further outlined in section 2.3, W and AW are obtained in $\mathcal{O}(n^2(\ell+1)k)$. By choosing ℓ and k to be small the computational overhead can be kept modest. This shows the importance of choosing vectors in W that significantly reduce the number of required iterations to make up for the computational overhead. Another factor to take into account is the additional storage requirements of the deflated CG. The main contributing factors are the matrices W and AW, which each are of size $n \times k$ and ℓ search directions of size n.

Algorithm 1 Deflated Conjugate Gradient method.

---

1 **procedure** DEFLATED-CG$(k, \ell)(A, b, x_{-1}, W, (AW), \text{tol})$
Ensure: $W \in \mathbb{R}^{n \times k}$ // k included in Alg. definition for interpretability, can obviously be inferred internally.
2 $\quad$ Choose x_0 such that $W^T r_0 = 0$ where $r_0 = b - Ax_0$
3 $\quad$ $x_0 = x_{-1} + W(W^T A W)^{-1} W^T r_{-1}$
4 $\quad$ Solve $W^T A W \mu_0 = W^T A r_0$ for μ and set $p_0 = r_0 - W\mu_0$ $\quad$ // deflation for initial iteration
5 $\quad$ **while** $|r_j| > \text{tol}$ (For j=1...) **do**
6 $\quad\quad$ $d_{j-1} = p_{j-1}^T A p_{j-1}$
7 $\quad\quad$ $\alpha_{j-1} = r_{j-1}^T r_{j-1} / d_{j-1}$
8 $\quad\quad$ $x_j = x_{j-1} + \alpha_{j-1} p_{j-1}$
9 $\quad\quad$ $r_j = r_{j-1} - \alpha_{j-1} A p_{j-1}$
10 $\quad\quad$ $\beta_{j-1} = r_j^T r_j / r_{j-1}^T r_{j-1}$ $\quad\quad\quad\quad\quad$ // from line 6 to here: standard conjugate gradient
11 $\quad\quad$ $\mu_j = $ Solve $W^T A W \mu_j = W^T A r_j$ $\quad\quad\quad\quad$ // deflation for following iteration
12 $\quad\quad$ $p_j = \beta_{j-1} p_{j-1} + r_j - W\mu_j$
13 $\quad\quad$ **if** $j < \ell$ **then**
14 $\quad\quad\quad$ Store $d_j, \alpha_j, \beta_j, \mu_j, p_j$
15 $\quad\quad$ **end if**
16 $\quad$ **end while**
17 **end procedure**

2.3 Approximate Eigenvectors

One way of obtaining approximate eigenvectors is with the Lanczos algorithm and extract Ritz value/vector pairs. The Lanczos algorithm from which Saad et al. [20] derived def-CG, generates a sequence of vectors $\{\boldsymbol{p}_j\}$ such that

$$\boldsymbol{p}_{j+1} \perp_A \text{span}\{W, \boldsymbol{p}_0, ..., \boldsymbol{p}_j\}.$$

The matrix A can then be transformed into a symmetric and partly tridiagonal matrix $T_{l+k} = Z^T A Z$, with $Z = [W, P_l] \in \mathbb{R}^{n \times (\ell+k)}$ and $P_\ell = [\boldsymbol{p}_0, ..., \boldsymbol{p}_{\ell-1}]$. The eigendecomposition of T_{l+k} produces pairs $(\theta_j, \boldsymbol{u}_j)_{j=1,...,\ell+k}$, of which θ_j are Ritz values that approximate the eigenvalues of A. The corresponding approximate eigenvector is obtained as the Ritz vector $\boldsymbol{v}_j = Z\boldsymbol{u}_j$. For an orthogonal projection technique, such as deflation, the residual of $(A - \theta I)\boldsymbol{v}$ should be orthogonal to Z [1], leading to the generalized eigenvalue problem

$$Z^\intercal (A - \theta I) Z \boldsymbol{u} = 0 \quad \Rightarrow \quad Z^\intercal A Z \boldsymbol{u} = \theta Z^\intercal Z \boldsymbol{u}.$$

This can be transformed to a normal eigenvalue problem by multiplying both sides with $(Z^\intercal Z)^{-1}$, which exist because the columns in Z are linearly independent, but the symmetric properties would be lost. Computing these matrices generates an additional, non-negligible computational cost, because

CG does not generate orthogonal, but A-conjugate directions.[2] By instead considering the base AZ for orthogonality, Morgan [12] rephrased the problem as a Galerkin method for approximating the eigenvalues of the inverse of A

$$(AZ)^T (AZ\boldsymbol{u} - \theta Z\boldsymbol{u}) = 0.$$

This method is referred to as *harmonic projection*. By introducing

$$F = (AZ)^T Z, \qquad G = (AZ)^T (AZ),$$

the problem is conveniently formulated as

$$G\boldsymbol{u} = \theta F\boldsymbol{u}. \tag{7}$$

By using properties of the search directions and residuals generated from Algorithm 1, Saad et al. [20] explicitly formed F and G from sparse matrices containing the stored quantities from the first ℓ iterations of Algorithm 1. This effectively reduces the computational overhead of finding approximate eigenvectors and makes the method competitive. Once Eq. (7) is solved, one chooses k of the $\ell + k$ Ritz values θ with corresponding vectors $\boldsymbol{u}$ and stores them in U. The k approximate eigenvectors for the next system are obtained as $W = ZU$. Preferably the largest or smallest θ are chosen but this is not required.

Remark: Connection to First-Order Methods In very high-dimensional machine learning models, second-order optimization methods based on the quadratic model of Eq. (2) are not as popular as first-order methods, like (stochastic) gradient descent and its many flavors. Although we do not further consider this area below, it may be helpful to note that the schemes described here have a weak connection to it, at least in the case of noise-free gradients. That is because the first step of CG is simply gradient descent. The recycling schemes described above can thus be compared conceptually to methods like momentum-based gradient descent [15] and other acceleration techniques.

3   Experiments

We test the utility of def-CG in a classical machine learning benchmark: The infinite MNIST [11] suite is a tool to automatically create arbitrary size datasets containing images of "hand-written" digits, by applying transformations to the classic MNIST set[3] of actual hand-written digits. We used it to generate a training set X of $36\,551$ images for the digits three and five, each of size 28×28 gray-scale pixels (This means the training set is three times larger than the set of threes and fives in the original MNIST set). By the standards of kernel methods, this is thus a comparably big data set. We consider binary probabilistic classification on this dataset, and follow a setup made popular by Kuss and Rasmussen [10]. This setting is a good example of the role of linear solvers in machine learning. It involves two nested loops of repeated linear optimization problems.

This involves computing a (Gaussian) Laplace approximation to the posterior arising from a Gaussian process prior on a latent function f (in our case, $p(f) = \mathcal{GP}(0, k)$, with the Gaussian/RBF kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \theta^2 \exp\left(-(\boldsymbol{x}_i - \boldsymbol{x}_j)^2 / 2\lambda^2\right)$), and the logistic link function $p(y_i \,|\, f_i) = \sigma(y_i f_i) = 1/(1 + e^{-y_i f_i})$ as the likelihood (see also [23, §3.7.3, which also outlines the explicit algorithm]).

The outer loop will find the optimal hyperparameters for the kernel and the inner (which is the focus of this study) will find the $\boldsymbol{f}$ that maximize

$$\Psi(\boldsymbol{f}) = \log p(\boldsymbol{y}|\boldsymbol{f}) + \log p(\boldsymbol{f}|X) = \log p(\boldsymbol{y}|\boldsymbol{f}) - \frac{1}{2}\boldsymbol{f}^T K^{-1}\boldsymbol{f} - \frac{1}{2}\log |K| - \frac{n}{2}\log 2\pi, \tag{8}$$

for a given kernel matrix K with Newton's method.

Newton's method converges to an extremum by evaluating the Jacobian and Hessian of a function at the current location $\boldsymbol{x}_n$ and finds a new location by computing $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \mathrm{Hess}_{(\Psi)}^{-1} \mathrm{Jac}_{(\Psi)}$. These iterations involve the solution of a linear system that changes in each iteration. For Laplace approximation the system to be solved can be made numerically stable by restructuring the computations

---

[2]Despite its name, the gradients produced by CG are actually orthogonal, not conjugate. The name arose out of the historical context, because it is a *conjugate* directions method that uses *gradients*.

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

Table 1: Iterative solvers operating on the MNIST classification task. The table shows the progress over Newton iterations. Within each Newton iteration the system in Eq. (10) needs to solved. Both iterative solvers were set to run until they achieve a relative error of $\epsilon = 10^{-5}$. The column labeled t shows cumulative runtimes for each method with the time to extract W included for def-CG.

	Cholesky		CG			def-CG($k=8$, $\ell=12$)		
It.	$\log p(\boldsymbol{y}|\boldsymbol{f})$	t [s]	$\log p(\boldsymbol{y}|\boldsymbol{f})$	rel. error δ	t [s]	$\log p(\boldsymbol{y}|\boldsymbol{f})$	rel. error δ	t [s]
1	-4926.523	426	-4968.760	$8.573 \cdot 10^{-3}$	231	-4968.760	$8.573 \cdot 10^{-3}$	245
2	-1915.537	896	-1931.348	$8.254 \cdot 10^{-3}$	492	-1938.585	$1.203 \cdot 10^{-3}$	436
3	-919.124	1366	-924.891	$6.274 \cdot 10^{-3}$	715	-926.668	$8.208 \cdot 10^{-3}$	617
4	-549.182	1875	-551.432	$4.097 \cdot 10^{-3}$	920	-551.796	$4.760 \cdot 10^{-3}$	790
5	-407.058	2362	-408.010	$2.339 \cdot 10^{-3}$	1088	-408.133	$2.641 \cdot 10^{-3}$	947
6	-353.632	2856	-354.040	$1.154 \cdot 10^{-3}$	1246	-354.085	$1.281 \cdot 10^{-3}$	1101
7	-335.575	3342	-335.711	$4.053 \cdot 10^{-4}$	1444	-335.744	$5.036 \cdot 10^{-4}$	1258
8	-331.326	3815	-331.346	$6.036 \cdot 10^{-5}$	1647	-331.355	$8.753 \cdot 10^{-5}$	1418
9	-330.997	4317	-330.998	$2.309 \cdot 10^{-6}$	1821	-330.996	$4.018 \cdot 10^{-6}$	1571

[10]. In each iteration the target and matrix get a new value

$$\boldsymbol{b}^{(i)} = H^{[i]\frac{1}{2}} K(H^{(i)}\boldsymbol{f}_X^{(i)} + \nabla \log p(y|\boldsymbol{f}_X^{(i)})), \tag{9}$$

$$A^{(i)} = I + H^{(i)\frac{1}{2}} K H^{(i)\frac{1}{2}}, \tag{10}$$

with $H = -\nabla\nabla \log p(\boldsymbol{y}|\boldsymbol{f}_X^{(i)})$. This restructuring assures that the eigenvalues λ_i of A are contained in $[1, n \max_{ij}(K_{ij})/4]$ and is therefore well-conditioned for most kernels [23].

Note how this task fits the setting sub-space recycling methods are designed for: It is difficult to analytically track how an update to $\boldsymbol{f}_X^{(i)}$ affects the elements of $(A^{(i)})^{-1}$ and $\boldsymbol{b}^{(i)}$, due the non-linear dependence in $\boldsymbol{f}_X$. But as the Newton optimizer converges, the iterates change less and less: $|\boldsymbol{f}_X^{(i)} - \boldsymbol{f}_X^{(i-1)}| < |\boldsymbol{f}_X^{(i-1)} - \boldsymbol{f}_X^{(i-2)}|$. Thus, $A^{(i)}$ and $\boldsymbol{b}^{(i)}$ will change less and less between iterations, and subspace recycling should become increasingly advantageous—up to a point, because of course we have to limit the dimensionality of the deflated space for computational reasons.

Solving the linear system in Eq. (10) with a Cholesky decomposition is $\mathcal{O}(n^3)$ expensive, where n is the dimension of A i.e. the size of the training set X. Iterative methods, such as CG due to the kernel matrix K being SPD, have been used to speed up the mode-finding of $\hat{\boldsymbol{f}}$ [23, 2]. Table 1 compares the cumulative computational cost and accuracy of the exact Cholesky decomposition, standard CG and deflated CG for each iteration in Newtons method. A reduction of the relative error $\epsilon = |\boldsymbol{b} - A\boldsymbol{x}_i|/|\boldsymbol{b}|$ was used as stopping criterion and was chosen to be 10^{-5}.

Table 1 shows that iterative methods on their own already save computations. Figure 2 also shows a plot of these results. The increasingly steep downward slope of the CG error in later optimization problems suggests that the systems are getting easier to solve, possibly because the matrix $A^{(i)}$ becomes better conditioned. But the figure also shows the additional advantage of sub-space recycling in def-CG. Eight approximate eigenvectors were used, yielding a saving of at least 12 CG iterations per system ($\sim 25\%$). Enough to outweigh the the computational overhead of finding W and AW. Initially, the reduction of iterations for def-CG stagnate (become parallel to CG) over the course of a few Newton iterations, which suggests the recycled subspace fails to reduce the effective condition number further. Either the recycled vectors do not find good approximations to the extreme eigenvalues and the algorithm can be improved, or the difference between successive $A^{(i)}$ is too significant so the propagated information is less useful. Additional experiments (not shown) suggest that both effects play a role, but the former, i.e. numerical stability, dominates. Methods that try to alleviate this problem by estimating the convergence of the approximate eigenvectors exist [6], but cause additional computational overhead.

To better understand the effect of the deflation on individual solutions of Eq. (10), Figure 3 compares the convergence of def-CG and standard CG. Each solver was set to run until a relative error of 10^{-8} was achieved and the results are shown in Fig. 3. The figure indicates that the computational savings do not stem from the initial projection onto the A-orthogonal complement of W as one could suspect,
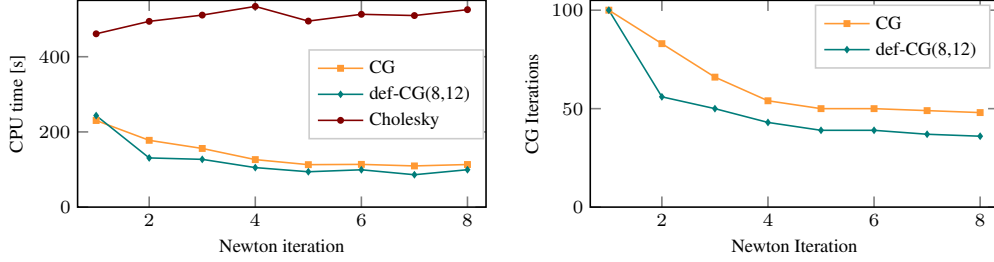
Figure 2: **Left:** Computational cost (CPU time) per iteration of Newton's method. **Right:** number of iterations required for CG and def-CG(8,12) to solve a single system (Eq. (10)) to a relative error of $\epsilon = 10^{-5}$. The stopping criterion for the Newton iteration was $\Delta\Psi(\boldsymbol{f}) < 1$, thus only requiring the first two terms in Eq. (8) to be computed.

but rather from a steeper convergence. This fits with the idea that deflation lowers the effective condition number of deflated system relative to the original problem.

3.1 Comparison to Linear-Cost Approximations

We now investigate the utility of sub-space recycling relative to the linear cost approximation methods of finite error discussed above in Section 1.1, in particular to inducing point methods. These methods assume that the training set elements $\boldsymbol{f}_n$ at $X \in \mathbb{R}^{n \times d}$ are approximately independent of each other when conditioned on a smaller set of values $\boldsymbol{f}_m$ at representer points $X_m \in \mathbb{R}^{m \times d}$. In GPC, this effectively reduces the task to optimizing only the latent variables $\boldsymbol{f}_m \in \mathbb{R}^m$ with $m < n$, to maximize the objective in Eq. (8). The latent variables for the remaining points in the data set are then *induced* by the conditional distribution $p(\boldsymbol{f}_{n-m} \mid \boldsymbol{f}_m)$, with mean $\mathbb{E}[\boldsymbol{f}_{n-m} \mid \boldsymbol{f}_m] = K_{(n-m)m}K_{mm}^{-1}\boldsymbol{f}_m$. A measure of the performance over the training set is then obtained by evaluating the objective with the inferred latent variables.

We compared the accuracy of iterative methods to inducing point methods with a randomly selected subset of the training data. Figure 4 shows the convergence of the Newton optimizer for subsets X_m of $\log p(\boldsymbol{y} \mid \boldsymbol{f})$ of varying sizes. (each time, error was evaluated on the entire training set X). The results confirm the expected picture: The approximate methods can be significantly faster than the iterative solves, but they also incur a significant approximation error. If an accurate solution is required, the iterative solvers can be competitive. For this experiment, the iterative methods have a computational cost comparable to that of the approximate methods running on a (comparably large)
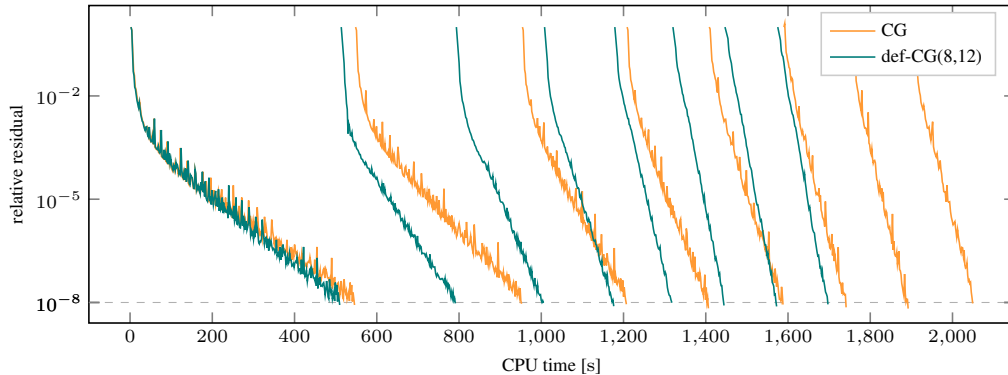

Figure 3: Relative residual over multiple solutions of Newton's method i.e. the systems described in Eq. (10) were solved. Each solver had a relative error of $\epsilon = 10^{-8}$ as stopping criterion. Similarly to the results in Fig. 2, the time required to find a solution that satisfies the error bound becomes faster for each Newton iteration. The deflated method achieves a faster convergence rate as seen by the slope of the relative residual. This confirms that re-using information from a previous system indeed lowers the effective condition number.
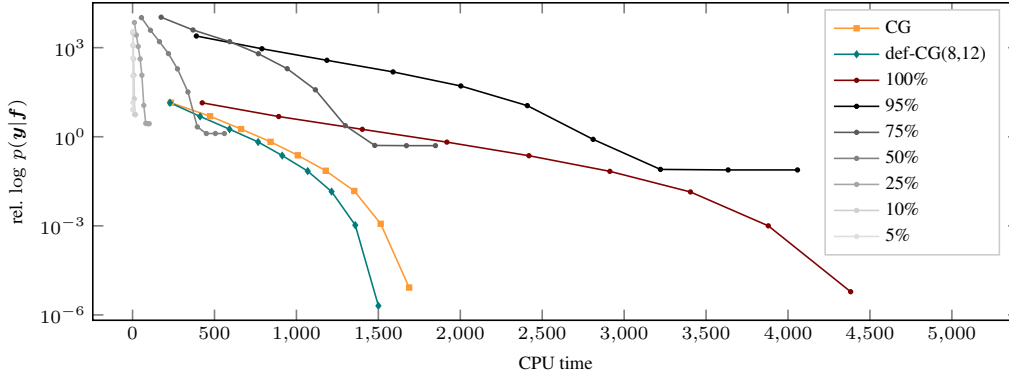
7

Figure 4: Comparison of accuracy between the iterative methods CG, def-CG and differently sized subsets of data measured as the relative error of $\log p(\boldsymbol{y} \mid \boldsymbol{f})$ to the "exact" (up to machine precision) value achieved by a direct (Cholesky) solver on the full data set. Each dot represents the approximated $\log p(\boldsymbol{y} \mid \boldsymbol{f})$ for the full training set after each iteration of Newton's method. The CPU time refers to the cumulative time spent solving the linear systems in Eq. (10), which is the computationally most expensive part of each Newton iteration.

subset of between 25% and 50% of the data set. But they also achieve an improvement of about 6 orders of magnitude in precision.

4   Conclusion

We have investigated the use of Krylov sub-space recycling methods for a realistic example application in machine learning. ML problems often involve outer loops of (hyper-) parameter optimization, which produce a sequence of interrelated linear, symmetric positive definite (aka. least-squares) optimization problems. Subspace-recycling methods allow for iterative linear solvers to share information across this sequence, so that later instances become progressively cheaper to solve. Our experiments suggests that doing so can lead to a useful reduction in computational cost. While non-trivial, sub-space recycling methods can be implemented and used with moderate coding overhead.

We also compared empirically to the popular option of fixing a low-rank sub-space a priori, in the form of spectral or inducing point methods. The overarching intuition here is that these methods can achieve much lower computational cost when they use a low-dimensional basis. But in exchange they also incur a significant computational error. In applications where computational *precision* is at least as important as computational *cost*, sub-space recycling iterative solvers provide a reliable answer of high quality. While their run-time scales quadratically with data-set size, they are certainly scalable, at acceptable run-times, to data-sets containing $\sim 10^5$ to $\sim 10^6$ data points.

References

[1] A. Chapman and Y. Saad. Deflated and augmented Krylov subspace techniques. *Numerical linear algebra with applications*, 4(1):43–66, 1997.

[2] A. Davies. *Effective Implementation of Gaussian Process Regression for Machine Learning*. PhD thesis, University of Cambridge, 2014.

[3] G. Ebadi, N. Alipour, and C. Vuik. Deflated and augmented global Krylov subspace methods for the matrix equations. *Applied Numerical Mathematics*, 99:137–150, 2016.

[4] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. *SIAM Journal on Scientific Computing*, 23(2):442–462, 2001.

[5] A. Gaul, M. Gutknecht, J. Liesen, and R. Nabben. A framework for deflated and augmented Krylov subspace methods. *SIAM Journal on Matrix Analysis and Applications*, 34(2):495–518, 2013.

[6] P. Gosselet, C. Rey, and J. Pebrel. Total and selective reuse of Krylov subspaces for the resolution of sequences of nonlinear structural problems. *International Journal for Numerical Methods in Engineering*, 94(1):60–83, 2013.

[7] P. Hennig, M. Osborne, and M. Girolami. Probabilistic numerics and uncertainty in computations. In *Proceedings of the Royal Society A*, volume 471, page 20150142, 2015.

[8] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems1. *Journal of Research of the National Bureau of Standards*, 49(6), 1952.

[9] M. J. Deep learning via Hessian-free optimization. In J. Fürnkranz and T. Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning*, pages 735–742, 2010.

[10] M. Kuss and C. Rasmussen. Assessing approximations for Gaussian process classification. In Y. Weiss, P. B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems*, volume 18, pages 699–706. 2006.

[11] G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling. In *Large Scale Kernel Machines*, pages 301–320. MIT Press, 2007.

[12] R. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

[13] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, second edition, 2006.

[14] M. Parks, E. De Sturler, G. Mackey, D. Johnson, and S. Maiti. Recycling Krylov subspaces for sequences of linear systems. *SIAM Journal on Scientific Computing*, 28(5):1651–1674, 2006.

[15] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 4(5):791–803, 1964.

[16] J. Quiñonero-Candela and C. Rasmussen. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research*, 6(Dec):1939–1959, 2005.

[17] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20, pages 1177–1184. 2008.

[18] A. Rahimi and B. Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 21, pages 1313–1320. 2009.

[19] Y. Saad. *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. SIAM, 2011.

[20] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h. A deflated version of the conjugate gradient algorithm. *SIAM Journal on Scientific Computing*, 21(5):1909–1926, 2000.

[21] E. Snelson and Z. Ghahramani. Local and global sparse Gaussian process approximations. In M. Meila and X. Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 524–531, 2007.

[22] M. Titsias. Variational learning of inducing variables in sparse Gaussian processes. In D. van Dyk and M. Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 567–574, 2009.

[23] C. Williams and C. Rasmussen. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[24] C. Williams and M. Seeger. Using the Nyström method to speed up kernel machines. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13, pages 682–688. 2001.