Least Squares Auto-Tuning

Shane Barratt
sbarratt@stanford.edu

Stephen Boyd
boyd@stanford.edu

April 10, 2019

Abstract

Least squares is by far the simplest and most commonly applied computational method in many fields. In almost all applications, the least squares objective is rarely the true objective. We account for this discrepancy by parametrizing the least squares problem and automatically adjusting these parameters using an optimization algorithm. We apply our method, which we call least squares auto-tuning, to data fitting.

1 Introduction

Since its introduction over 200 years ago by Legendre and Gauss, the method of least squares [Leg05, Gau09] has been one of the most widely employed computational techniques in many fields, including machine learning and statistics, signal processing, control, robotics, and finance [BV18]. Its wide application primarily comes from the fact that it has a simple analytical solution, it is easy to understand, and very efficient and stable algorithms for computing its solution have been developed [LH95, GVL12].

In essentially all applications, the least squares objective is not the true objective; rather it is a surrogate for the real goal. For example, in least squares data fitting, the objective is not to solve a least squares problem involving the training data set, but rather to find a model or predictor that generalizes, *i.e.*, achieves small error on new unseen data. In control, the least squares objective is only a surrogate for keeping the state near some target or desired value, while keeping the control or actuator input small.

To account for the discrepancy between the least squares objective and the true objective, it is common practice to modify (or tune) the least squares problem that is solved to obtain a good solution in terms of the true objective. Typical tricks here include modifying the data, adding additional (regularization) terms to the cost function, or varying hyper-parameters or weights in the least squares problem to be solved.

The art of using least squares in applications is generally in how to carry out these modifications or choose these additional terms, and how to choose the hyper-parameters. The choice of hyper-parameters is often done in an ad hoc way, by varying them, solving the least squares problem, and then evaluating the result using the true objective or objectives.

In data fitting, for example, regularization scaled by a hyper-parameter is added to the least squares problem, which is solved for many values of the hyper-parameter to obtain a family of data models; among these, the one that gives the best predictions on a test set of data is the one that is ultimately used. We refer to this general design approach, of modifying the least squares problem to be solved, varying some hyper-parameters, and evaluating the result using the true objective, as *least squares tuning*. It is very widely used, and can be extremely effective in practice.

Our focus in this paper is on automating the process of least squares tuning, for a variety of data fitting applications. We parametrize the least squares problem to be solved by hyper-parameters, and then automatically adjust these hyper-parameters using a gradient-based optimization algorithm, to obtain the best (or at least better) true performance. This lets us automatically search the hyper-parameter design space, which can lead us to better designs than could be found manually, or help us find good values of the hyper-parameters more quickly than if the adjustments were done manually. We refer to the method as *least squares auto-tuning*.

One of our main contributions in this paper is the observation that least squares auto-tuning is very effective for a wide variety of data fitting problems that are usually handled using more complex and advanced methods, such as non-quadratic loss functions or regularizers in regression, or special loss functions for classification problems. In addition, it can simultaneously adjust hyper-parameters in the feature generation chain. Through several examples, we show that ordinary least squares, used for over 200 years, coupled with automated hyper-parameter tuning, can be very effective as a method for data fitting.

The method we describe for least squares auto-tuning is easy to understand and just as easy to implement. Moreover, it is an exercise in calculus to find the derivative of the least squares solution, and an exercise in numerical linear algebra to compute it efficiently. We describe an implementation that utilizes new and powerful software frameworks that were originally designed to optimize the parameters in deep neural networks, making it very efficient on modern hardware and allowing it to scale to (extremely) large least squares tuning problems.

Our contributions. We claim three main contributions. The first contribution is the observation that the least squares solution map can be efficiently differentiated, including when the problem data is sparse; we mirror our description with an open-source implementation for both of these cases. The second contribution is the method of least squares auto-tuning, which can automatically tune hyper-parameters in least squares problems. The final contribution is our unique application of least squares auto-tuning to data fitting.

2 Background and related work

Our work mainly falls at the intersection of two fields: automatic differentiation and hyper-parameter optimization. In this section we review related work.

Automatic differentiation. The general idea of automatic differentiation (AD) is to automatically compute the derivatives of a function given a program that evaluates the function [Wen64, Spe80]. In general, the cost of computing the derivative or gradient of a function can be made about the same (usually within a factor of 5) as computing the function [BS83, GW08]. This means that an optimization algorithm can obtain derivatives of the function it is optimizing as fast as computing the function itself, and explains the proliferation of gradient-based minimization methods [BPRS18, BCN18]. There are many popular implementations of AD, and they generally fall into two categories. The first category is are trace-based AD systems, which trace computations at runtime, as they are executed; popular ones include PyTorch [PGC+17], Tensorflow eager [AMP+19], and autograd [MDA15a]. The second category are based on source transformation, which transform the (native) source code that implements the function into source code that implements the derivative operation. Popular implementations here include Tensorflow [ABC+16], Tangent [vMMW18], and (more recently) Zygote [Inn18].

Argmin differentiation. Given an optimization problem parametrized by some parameters, the solution map is a set-valued map from those parameters to a set of solutions. If the solution map is differentiable (and in turn unique), then we can differentiate the solution map [DR09]. For convex optimization problems that satisfy strong duality, the solution map is given by the set of solutions to the KKT conditions, which can in some cases be differentiated using the implicit function theorem [Bar18]. This idea has been applied to convex quadratic programs [AK17], stochastic optimization [DAK17], games [LFK18, LFK19], physical systems [dABPSA+18], control [AJS+18], and structured inference [BM16, BYM17]. In machine learning, these techniques were originally applied to neural networks [LSAH98, EN99] and ridge regression [Ben00], and more recently to lasso [MBP12], support vector machines [CVBM02], and log-linear models [KSC07, FDN08] A notable AD implementation of these methods is the PyTorch implementation `qpth`, which can compute derivatives of the solution map of quadratic programs [Amo17].

Unrolled optimization. Another approach to argmin differentiation is unrolled optimization. In unrolled optimization, one fixes the number of iterations in an iterative minimization method, and differentiates the steps taken by the method itself [Dom12, BP14]. The idea of unrolled optimization was originally applied to optimizing hyper-parameters in deep neural networks, and has been extended in several ways to adjust learning rates, regularization parameters [MDA15b, FLFC16, LD18], and even to learn weights on individual data points [RZYU18]. It is still unclear whether argmin differentiation should be performed via implicit differentiation or unrolled optimization. However, when the optimization problem is nonconvex, differentiation by unrolled optimization seems to be the only practical one.

Hyper-parameter optimization. The idea of adjusting hyper-parameters to obtain better true performance in the context of data fitting is hardly new, and routinely employed in settings more sophisticated than least squares. For example, in data fitting, it is standard

practice to vary one or more hyper-parameters to generate a set of models, and choose the model that attains the best true objective, which is usually error on an unseen test set. The most commonly employed methods here include grid search, random search [BB12], Bayesian optimization [Moč75, Ras04, SLA12], and covariance matrix adaptation [HO96].

3 Least squares auto-tuning

In this section we describe the idea of least squares tuning, our method for least squares auto-tuning, and our implementation.

3.1 Least squares problem

The *matrix least squares problem* that depends on a *hyper-parameter vector* $\omega \in \Omega \subseteq \mathbf{R}^p$ has the form

$$\text{minimize} \quad \|A(\omega)\theta - B(\omega)\|_F^2, \tag{1}$$

where the variable is $\theta \in \mathbf{R}^{n\times m}$, the *least squares optimization variable* or *parameter matrix*, and $A : \Omega \to \mathbf{R}^{k\times n}$ and $B : \Omega \to \mathbf{R}^{k\times m}$ map the hyper-parameter vector to the least squares *problem data*. The norm $\|\cdot\|_F$ denotes the Frobenius norm, *i.e.*, the squareroot of the sum of squares of the entries of a matrix. We assume throughout this paper that $A(\omega)$ has linearly independent columns, which implies that it is tall, *i.e.*, $k \geq n$. Under these assumptions, the *least squares solution* is unique, given by

$$\theta^{\text{ls}}(\omega) = A(\omega)^\dagger B(\omega) = (A(\omega)^T A(\omega))^{-1} A(\omega)^T B(\omega), \tag{2}$$

where $A(\omega)^\dagger$ denotes the (Moore-Penrose) pseudo-inverse. Solving a least squares problem for a given hyper-parameter vector corresponds to computing $\theta^{\text{ls}}(\omega)$. We will think of the least squares solution θ^{ls} as a function mapping the hyper-parameter $\omega \in \Omega$ to a parameter $\theta^{\text{ls}}(\omega) \in \mathbf{R}^{n\times m}$.

Multi-objective least squares. In many applications we have multiple least squares objectives. These are typically scalarized by forming a positive weighted sum, which leads to

$$\text{minimize} \quad \lambda_1\|A_1(\omega)\theta - B_1(\omega)\|_F^2 + \cdots + \lambda_r\|A_r(\omega)\theta - B_r(\omega)\|_F^2, \tag{3}$$

where $\lambda_1, \ldots, \lambda_r$ are the positive objective weights. This problem is readily expressed as the standard least squares problem (1) by stacking the objectives, with

$$A(\omega) = \begin{bmatrix} \sqrt{\lambda_1}A_1(\omega) \\ \vdots \\ \sqrt{\lambda_r}A_r(\omega) \end{bmatrix}, \quad B(\omega) = \begin{bmatrix} \sqrt{\lambda_1}B_1(\omega) \\ \vdots \\ \sqrt{\lambda_r}B_r(\omega) \end{bmatrix}. \tag{4}$$

We will often write least squares problems in the form (3), and assume that the reader understands that the problem data can easily be transformed into (4). The objective weights $\lambda_1, \ldots, \lambda_r$ can also be considered hyper-parameters themselves, or to depend on hyper-parameters; to keep the notation light we do not show this dependence.

Solving the least squares problem. For a given value of ω, there are many ways to solve the least squares problem (1), including dense or sparse QR or other factorizations [Gol65, BD80], iterative methods such as CG or LSQR [HS52, PS82], and many others [LH95, GVL12]. Very efficient libraries for computing the least squares solution that target multiple CPUs or one or more GPUs have also been developed [DCHD90, ABB+99, WCV11, SK10]. We note that the problem is separable across the columns of θ, *i.e.*, the problem splits into m independent least squares problems with vector variables and a common coefficient matrix.

We give a few more details here for two of these methods. First we consider the case where $A(\omega)$ and $B(\omega)$ are stored and manipulated as dense matrices. One attractive option (for GPU implementation) is to form the Gram matrix $G = A^TA$, along with $H = A^TB$. This requires around (order) kn^2 and knm flops, respectively, but these matrix-matrix multiplies are BLAS level 3 operations, which can be carried out very efficiently. To compute $\theta^{\rm ls}$, we can use a Cholesky factorization of G, $G = LL^T$, which costs order n^3 flops, solve the triangular equation $LY = H$, which costs order n^2m flops, and then solve the triangular equation $L^T\theta^{\rm ls} = Y$, which costs order n^2m flops. Overall, the complexity of solving a dense least squares problem is order $kn(n+m)$.

The other case for which we give more detail is when $A(\omega)$ is represented as an abstract linear operator, and not as a matrix. This is a natural representation when $A(\omega)$ is large and sparse, or represented as a product of small (or sparse) matrices. That is, we can evaluate $A(\omega)u$ for any $u \in \mathbf{R}^n$, and $A(\omega)^Tv$ for any $v \in \mathbf{R}^k$. (This is the so-called *matrix-free* representation.) We can use CG or LSQR to solve the least squares problem, in parallel for each column of θ. The complexity of CG or LSQR depends on the problem, and can vary considerably based on the data, size, sparsity, choice of pre-conditioner, and required accuracy [HS52, PS82].

3.2 Least squares tuning problem

In a *least squares tuning problem*, our goal is to choose the hyper-parameters to achieve some goal. We formalize this as the problem

$$\text{minimize} \quad F(\omega) = \psi(\theta^{\rm ls}(\omega)) + r(\omega), \tag{5}$$

with variable $\omega \in \Omega$ and objective $F : \Omega \to \mathbf{R} \cup \{+\infty\}$, where $\psi : \mathbf{R}^{n\times m} \to \mathbf{R}$ is the *true objective function*, and $r : \Omega \to \mathbf{R} \cup \{+\infty\}$ is the *hyper-parameter regularization function*. We use infinite values of r (and therefore F) to encode constraints on the hyper-parameter ω, and will assume that $r(\omega)$ is defined as ∞ for $\omega \notin \Omega$. A least squares tuning problem is specified by the functions A, B, ψ, and r. We will make some additional assumptions about these functions below.

The hyper-parameter regularization function r can itself contain a few parameters that can be varied, which of course affects the hyper-parameters chosen in the least squares auto-tuning problem (5), which in turn affects the parameters selected by least squares. We refer to parameters that may appear in the hyper-parameter regularization function as *hyper-hyper-parameters*.

The least squares tuning problem (5) can be formulated in several alternative ways, for example as the constrained problem with variables $\theta \in \mathbf{R}^{n\times m}$ and $\omega \in \Omega$

$$\begin{array}{ll} \text{minimize} & \psi(\theta) + r(\omega) \\ \text{subject to} & A(\omega)^T A(\omega)\theta = A(\omega)^T B(\omega). \end{array} \tag{6}$$

In this formulation, θ and ω are independent variables, coupled by the constraint, which is the optimality condition for the least squares problem (1). Eliminating the constraint in this problem yields our formulation (5).

Solving the least squares tuning problem. The least squares tuning problem is in general nonconvex, and difficult or impossible as a practical matter to solve exactly [Pol87, BV04]. (One important exception is when ω is a scalar and Ω is an interval, in which case we can simply evaluate $F(\omega)$ on a grid of values over Ω.) This means that we will need to resort to a local optimization or heuristic method in order to (approximately) solve it.

We will assume that A and B are differentiable in ω, which implies that θ^{ls} is differentiable in ω, since the mapping from ω to θ^{ls} is differentiable. We will also assume that ψ is differentiable, which implies that the true objective $\psi(\theta^{\mathrm{ls}}(\omega))$ is differentiable in the hyper-parameters ω. This means that the first term (the true objective) in the least squares tuning problem (5) is differentiable, while the second one (the hyper-parameter regularizer) need not be. There are many methods that can be used to (approximately) solve such a composite problem [DR56, LM79, Sho85, BPC+11, Nes13b, PB14].

For completeness, we describe one of the simplest methods, the proximal gradient method (which stems from the proximal point method [Mar70]; for a modern reference see [Nes13a]), given by the iteration

$$\omega^{k+1} = \mathbf{prox}_{t^k r}(\omega^k - t^k \nabla_\omega \psi(\theta^{\mathrm{ls}}(\omega^k))),$$

where k denotes the iteration number, $t^k > 0$ is a step size, and the proximal operator $\mathbf{prox}_{tr} : \mathbf{R}^p \to \Omega$ is given by

$$\mathbf{prox}_{tr}(\nu) = \operatorname*{argmin}_{\omega\in\Omega} \left(tr(\omega) + (1/2)\|\omega - \nu\|_2^2 \right).$$

We assume here that the argmin exists; when it is not unique, we choose any minimizer. In order to use the proximal gradient method, we need the proximal operator of tr to be relatively easy to evaluate.

The proximal gradient method reduces to the ordinary gradient method when $r = 0$ and $\Omega = \mathbf{R}^p$. Another special case is when $\Omega \subset \mathbf{R}^p$, and $r(\omega) = 0$ for $\omega \in \Omega$. In this case r is the indicator function of the set Ω, the proximal operator of tr is Euclidean projection onto Ω, and the proximal gradient method coincides with the projected gradient method.

Choosing the step size. There are many ways to choose the step size. We adopt the following simple adaptive scheme, borrowed from [LB17]. The method begins with an initial step size t^1. If the function value decreases or stays the same from iteration k to $k+1$, or $F(\omega^{k+1}) \leq F(\omega^k)$, then we increase the step size a bit and accept the update. If, on the

other hand, the function value increases from iteration k to $k+1$, or $F(\omega^{k+1}) > F(\omega^k)$, we decrease the step size substantially and reject the update, *i.e.*, $\omega^{k+1} = \omega^k$. A simple rule for increasing the step size is $t^{k+1} = (1.2)t^k$, and a simple rule for decreasing it is $t^{k+1} = (1/2)t^k$.

We note that more sophisticated step size selection methods exist (see, *e.g.*, the line search methods for the Goldstein or Armijo conditions [NW06]).

Stopping criterion. By default, we run the method for a fixed maximum number of iterations. If $F(\omega^{k+1}) \leq F(\omega^k)$, then a reasonable stopping criterion at iteration $k+1$ is

$$\|(\omega^k - \omega^{k+1})/t^k + (g^{k+1} - g^k)\|_2 \leq \epsilon, \tag{7}$$

where $g^k = \nabla_{\omega^k} \psi(\theta^{\mathrm{ls}}(\omega^k))$, for some small tolerance $\epsilon > 0$. (For more justification of this stopping criterion, see Appendix B.) When $r = 0$ and $\Omega = \mathbf{R}^p$ (*i.e.*, the proximal gradient method coincides with the ordinary gradient method), this stopping criterion reduces to

$$\|g^{k+1}\|_2 \leq \epsilon,$$

which is the standard stopping criterion in the ordinary gradient method. The full algorithm for least squares auto-tuning via the proximal gradient method is summarized in Algorithm 3.1.

Algorithm 3.1 *Least squares auto-tuning via proximal gradient.*

given initial hyper-parameter vector $\omega^1 \in \Omega$, initial step size t^1, number of iterations n_{iter}, tolerance ϵ.

for $k = 1, \ldots, n_{\mathrm{iter}}$

1. *Solve the least squares problem.* $\theta^{\mathrm{ls}}(\omega^k) = (A^T(\omega^k)A(\omega^k))^{-1}A^T(\omega^k)B(\omega^k)$.
2. *Compute the gradient.* $g^k = \nabla_\omega \psi(\theta^{\mathrm{ls}}(\omega^k))$.
3. *Compute the gradient step.* $\omega^{k+1/2} = \omega^k - t^k g^k$.
4. *Compute the proximal operator.* $\omega^{\mathrm{tent}} = \mathbf{prox}_{t^k r}(\omega^{k+1/2})$.
5. **if** $F(\omega^{\mathrm{tent}}) \leq F(\omega^k)$,
   Increase step size and accept update. $t^{k+1} = (1.2)t^k$; $\omega^{k+1} = \omega^{\mathrm{tent}}$.
   Stopping criterion. **quit** if $\|(\omega^k - \omega^{k+1})/t^k + (g^{k+1} - g^k)\|_2 \leq \epsilon$.
6. **else** *Decrease step size and reject update.* $t^{k+1} = (1/2)t^k$; $\omega^{k+1} = \omega^k$.

end for

We emphasize that many other methods can be used to (approximately) solve the least squares tuning problem (5); we have described the proximal gradient method here only for completeness.

3.3 Computing the gradient

In principle, one could calculate the gradient by directly differentiating the linear algebra routines used to solve the least squares problem [Smi95]. We, however, work out formulas for computing the gradient analytically, that work in the case of sparse A and allow for a more efficient implementation.

To compute $g = \nabla_\omega \psi(\theta^{\mathrm{ls}}(\omega)) \in \mathbf{R}^p$ we can make use of the chain rule for the composition $f = \psi \circ \theta^{\mathrm{ls}}$. We assume that $\theta = \theta^{\mathrm{ls}}(\omega)$ has been computed. We first compute $\nabla_\theta \psi(\theta) \in \mathbf{R}^{n\times m}$, and then form

$$C = (A^T A)^{-1} \nabla_\theta \psi(\theta) \in \mathbf{R}^{n\times m}.$$

(Here we have dropped the dependence on ω, *i.e.*, $A = A(\omega)$.) If A is stored as a dense matrix, we observe that $G = A^T A$ and its factorization have already been computed (to evaluate θ), so this step involves a back-solve. If A is represented as an abstract operator, we can evaluate each column of C (in parallel) using an iterative method.

It can be shown (see Appendix A) that the gradients of ψ with respect to A and B are given by

$$\nabla_A \psi = (B - A\theta)C^T - AC\theta^T \in \mathbf{R}^{k\times n}, \qquad \nabla_B \psi = AC \in \mathbf{R}^{k\times m}. \tag{8}$$

(Again, the dependence on ω has been dropped.)

In the case of A dense, we can explicitly form $\nabla_A \psi$ and $\nabla_B \psi$, since they are the same size as A and B, which we already have stored. The overall complexity of computing $\nabla_A \psi$ and $\nabla_B \psi$ is $kn(n+m)$ in the dense case, which is the same cost as solving the least squares problem.

In the case of A sparse, we can explicitly form the matrix $\nabla_B \psi$, but we can not form $\nabla_A \psi$, since it is the size of A, which by assumption is too large to store. Instead, we assume that ω only affects A at a subset of its entries, Γ, *i.e.*, $A_{ij}(\omega) = 0$ for all $i, j \notin \Gamma$, and for all $\omega \in \Omega$. By doing this, we have restricted $\nabla_A \psi$ to have the same sparsity pattern as A, meaning we only need to compute $(\nabla_A \psi)_{ij}$ for $i, j \in \Gamma$. That is, we compute

$$(\nabla_A \psi)_{ij} = \begin{cases} (b_i - \theta a_i) c_j^T - a_i^T (C\theta^T)_j & i, j \in \Gamma \\ 0 & \text{otherwise,} \end{cases}$$

where b_i is the ith row of B, a_i is the ith row of A, c_j is the jth column of C, and $(C\theta^T)_j$ is the jth column of $C\theta^T$. (This computation can be done in parallel.)

The next step is to compute $g = \nabla_\omega \psi$ given $\nabla_A \psi$ and $\nabla_B \psi$. We first describe how g can be computed in the case of dense A, and then in the case of sparse A.

Dense A. We first evaluate $\nabla_\omega A_{ij} \in \mathbf{R}^p$ and $\nabla_\omega B_{ij} \in \mathbf{R}^p$, the gradients of the problem data entries with respect to ω. If these gradients are all dense, we need to store $k(n+m)$ vectors in $\mathbf{R}^p$; but generally, they are quite sparse. (We explain how to take advantage of the sparsity in §3.4.) Finally, we have

$$g = \sum_{i,j} (\nabla_A \psi)_{ij} (\nabla_\omega A)_{ij} + \sum_{i,j} (\nabla_B \psi)_{ij} (\nabla_\omega B)_{ij}.$$

Assuming these are all dense, this requires order $knp + kmp = kp(n+m)$ flops. The overall complexity of evaluating the gradient g is order $k(n+m)(n+p)$.

Sparse A. When A is large and sparse, we only need to compute the inner product at the entries of A that are affected by ω, *i.e.*, we compute

$$g = \sum_{i,j \in \Gamma} (\nabla_A \psi)_{ij} (\nabla_\omega A)_{ij} + \sum_{i,j} (\nabla_B \psi)_{ij} (\nabla_\omega B)_{ij}.$$

If $|\Gamma| \ll kn$, then this can be much faster than treating A as dense.

3.4 Implementation

The equations in §3.3 for computing g do not directly lend themselves to an implementation. For example, we need to compute $\nabla_\theta \psi$, $\nabla_\omega A_{ij}$ and $\nabla_\omega B_{ij}$, which depend on the form of ψ, A, and B. Also, we would like to take advantage of the (potential) sparsity of these gradients. We can use libraries for automatic differentiation, *e.g.*, PyTorch [PGC+17] and Tensorflow [ABC+16], to automatically (and efficiently) compute g given ψ, A, and B.

These libraries generally work by representing functions as differentiable computation graphs, allowing one to evaluate the function as well as its gradient. In our case, we represent ψ as a function of ω, defined by a differentiable computation graph, and use these libraries to automatically compute $g = \nabla_\omega \psi$. In order to use these libraries to compute g, we need to implement an operation that solves the least squares problem (2) and computes its gradients (8). We have implemented an operation `lstsq(A,B)` that does exactly this, in both PyTorch and Tensorflow, in both the dense and sparse case. (The code can be found in the Appendix.)

There are several advantages of using these libraries. First, they automatically exploit parallelism and gradient sparsity. Second, they utilize BLAS level 3 operations, which are very efficient on modern hardware. Third, they make it easy to represent the functions ψ, A, and B, since they can be represented as compositions of (the many) pre-defined operations in these libraries.

We also provide a generic PyTorch implementation of the adaptive proximal gradient algorithm that we used in this paper.

GPU timings. Table 1 gives timings of computing $\psi(\theta^{\mathrm{ls}}(\omega)) = \mathbf{tr}(\mathbf{1}\mathbf{1}^T \theta^{\mathrm{ls}}(\omega))$ and its gradient g for a random problem (where ω simply scales the rows of a fixed A and B) and various problem dimensions, where A is dense. The timings given are for the PyTorch implementation, on an unloaded GeForce GTX 1080 Ti Nvidia GPU using 32-bit floating point numbers (floats). The timings are about ten times longer using 64-bit floating point numbers (doubles). (For most applications, including data fitting, we only need floats.) We also give the percentage of time spent on the Cholesky factorization of the Gram matrix.

Table 1: GPU timings.

k	n	m	p	Compute ψ	Compute g	Cholesky
20000	10000	10000	20000	1.32 s	2.49 s	15.6 %
20000	10000	1	20000	446 ms	614 ms	16.1 %
100000	1000	100	100000	28 ms	28 ms	10.8 %

3.5 Equality constrained extension

One can easily extend the ideas described in this article to the equality-constrained least squares problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|A(\omega)\theta - B(\omega)\|_F^2 \\ \text{subject to} & C(\omega)\theta = D(\omega), \end{array}$$

with variable $\theta \in \mathbf{R}^{n\times m}$, where $C : \Omega \to \mathbf{R}^{d\times n}$ and $D : \Omega \to \mathbf{R}^{d\times m}$. The pair of primal and dual variables $(\theta, \nu) \in \mathbf{R}^{n\times m} \times \mathbf{R}^{d\times m}$ are optimal if and only if they satisfy the KKT conditions [BV18, Chapter 16]

$$\begin{bmatrix} 0 & A(\omega)^T & C(\omega)^T \\ A(\omega) & -I & 0 \\ C(\omega) & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ r \\ \nu \end{bmatrix} = \begin{bmatrix} 0 \\ B(\omega) \\ D(\omega) \end{bmatrix},$$

where $r = A(\omega)\theta - B(\omega)$ is the residual. From here on, we let

$$M(\omega) = \begin{bmatrix} 0 & A(\omega)^T & C(\omega)^T \\ A(\omega) & -I & 0 \\ C(\omega) & 0 & 0 \end{bmatrix}.$$

When A and C are dense, one can factorize M directly, using an LDL^T factorization [LH95]. When A and C are sparse matrices, one can solve this system directly by using a sparse LDL^T solver, or iteratively (*i.e.*, without forming the matrix M) using, *e.g.*, MINRES [PS75].

Our true objective function becomes a function of both θ and ν. Suppose we have the gradients $\nabla_\theta \psi$ and $\nabla_\nu \psi$. We first compute

$$\begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} = M(\omega)^{-1} \begin{bmatrix} \nabla_\theta \psi \\ 0 \\ \nabla_\nu \psi \end{bmatrix}.$$

Then the gradients of ψ with respect to A and B are given by

$$\nabla_A \psi = -(r g_1^T + g_2 \theta^T), \quad \nabla_B \psi = g_2,$$

and with respect to C and D are given by

$$\nabla_C \psi = -(\nu g_1^T + g_3 \theta^T), \quad \nabla_D \psi = g_3.$$

Computing the gradients of the solution map requires the solution of one linear system, and thus has roughly the same complexity of computing the solution itself (and much less when a factorization is cached). When A and C are sparse, we can compute their gradients at only the nonzero elements, in a similar fashion to the procedure described in §3.3.

4 Least squares data fitting

In the previous section we described the general idea of least squares auto-tuning. In this section, and for the remainder of the paper, we apply least squares auto-tuning to data fitting.

4.1 Least squares data fitting

In a data fitting problem, we have *training data* consisting of *inputs* $u_1, \ldots, u_N \in \mathcal{U}$ and *outputs* $y_1, \ldots, y_N \in \mathbf{R}^m$. In *least squares data fitting*, we fit the parameters of a predictor

$$\hat{y} = \phi(u, \omega^{\text{feat}})^T \theta,$$

where $\theta \in \mathbf{R}^{n \times m}$ is the model variable, $\omega^{\text{feat}} \in \Omega^{\text{feat}} \subseteq \mathbf{R}^{p^{\text{feat}}}$ are feature engineering hyper-parameters, and $\phi : \mathcal{U} \times \Omega^{\text{feat}} \to \mathbf{R}^n$ is a featurizer (assumed to be differentiable in its second argument). We note that this predictor is linear in the output of the featurizer.

To select the model parameters, we solve a least squares problem with data given by

$$A(\omega) = \begin{bmatrix} e^{\omega_1^{\text{data}}} \phi(u_1, \omega^{\text{feat}})^T \\ \vdots \\ e^{\omega_N^{\text{data}}} \phi(u_N, \omega^{\text{feat}})^T \\ e^{\omega_1^{\text{reg}}} R_1 \\ \vdots \\ e^{\omega_d^{\text{reg}}} R_d \end{bmatrix}, \qquad B(\omega) = \begin{bmatrix} e^{\omega_1^{\text{data}}} y_1 \\ \vdots \\ e^{\omega_N^{\text{data}}} y_N \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

where $\omega^{\text{data}} \in \Omega^{\text{data}} \subseteq \mathbf{R}^N$ are data weighting hyper-parameters, $R_1, \ldots, R_d$ are regularization matrices with appropriate sizes, and $\omega^{\text{reg}} \in \Omega^{\text{reg}} \subseteq \mathbf{R}^d$ are regularization hyper-parameters.

The overall hyper-parameter is denoted

$$\omega = (\omega^{\text{feat}}, \omega^{\text{data}}, \omega^{\text{reg}}) \in \Omega^{\text{feat}} \times \Omega^{\text{data}} \times \Omega^{\text{reg}}.$$

We will describe the roles of each hyper-parameter in more detail below; but here we note that ω^{data} scales the individual training data examples, ω^{feat} are hyper-parameters in our featurizer, and ω^{reg} are hyper-parameters that scale each of our regularizers. We assume that the hyper-parameter regularization function is separable, meaning it has the form

$$r(\omega) = r^{\text{feat}}(\omega^{\text{feat}}) + r^{\text{data}}(\omega^{\text{data}}) + r^{\text{reg}}(\omega^{\text{reg}}),$$

leading to a separable proximal operator [PB14].

4.2 True objective function

We also have *validation data* composed of inputs $u_1^{\text{val}}, \ldots, u_{N_{\text{val}}}^{\text{val}} \in \mathcal{U}$ and outputs $y_1^{\text{val}}, \ldots, y_{N_{\text{val}}}^{\text{val}} \in \mathbf{R}^m$. We form predictions

$$\hat{y}_i^{\text{val}} = \phi(u_i^{\text{val}})^T \theta^{\text{ls}}(\omega),$$

where the featurizer is fixed, or $\phi(u) = \phi(u, \omega^{\text{data}})$. The true objective ψ in least squares data fitting corresponds to the average loss of our predictions of the validation outputs, which has the form

$$\psi(\theta) = \frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} l(\hat{y}_i^{\text{val}}, y_i^{\text{val}}),$$

where $l : \mathbf{R}^m \times \mathbf{R}^m \to \mathbf{R}$ is a penalty function (assumed to be differentiable in its first argument).

Regression and classification. The setting that we have described encompasses many problems in data fitting, including both regression and classification. In regression, the output is a scalar, *i.e.*, $y \in \mathbf{R}$. In multi-task regression, the output is a vector, *i.e.*, $y \in \mathbf{R}^m$. In boolean classification, $y \in \{e_1, e_2\}$ (e_i is the ith unit vector in $\mathbf{R}^2$), and the output represents a boolean class. In multi-class classification, $y \in \{e_i \mid i = 1, \ldots, m\}$ (e_i is the ith unit vector in $\mathbf{R}^m$), and the output represents a class label.

Regression penalty function. The penalty function in regression (and multi-task regression) problems often has the form

$$l(\hat{y}, y) = \pi(r),$$

where $r = \hat{y} - y$ is the residual and $\pi : \mathbf{R}^m \to \mathbf{R}$ is a penalty function applied to the residual. Some common forms for π are listed below.

- *Square.* The square penalty is given by $\pi(r) = \|r\|_2^2$.
- *Huber.* The Huber penalty is a robust penalty that has the form of the square penalty for small residuals, and the 2-norm penalty for large residuals:

$$\pi(r) = \begin{cases} \|r\|_2^2 & \|r\|_2 \leq M \\ M(2\|r\|_2 - M) & \|r\|_2 > M. \end{cases}$$

  We can consider M as a hyper-hyper-parameter.

- *Bisquare.* The bisquare penalty is a robust penalty that is constant for large residuals:

$$\pi(r) = \begin{cases} \frac{M^2}{6}\left(1 - [1 - (\frac{\|r\|_2^2}{M^2})]^3\right) & \|r\|_2 \leq M \\ M^2/6 & \|r\|_2 > M. \end{cases}$$

  We can consider M as a hyper-hyper-parameter.

Classification penalty function. For classification, we associate with our prediction $\hat{y} \in \mathbf{R}^m$ the probability distribution on the m label values given by

$$\mathbf{Prob}(y = e_i) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^m e^{\hat{y}_j}}, \quad i = 1, \ldots, m.$$

We interpret our prediction $\hat{y}$ as giving us a distribution on the labels of y, given x.

We will use the *cross-entropy loss* as the penalty function in classification. It has the form

$$l(\hat{y}, y = e_i) = -\hat{y}_i + \log(\sum_{j=1}^m e^{\hat{y}_j}), \quad i = 1, \ldots, m.$$

The true loss is then average negative log probability of y under the predicted distribution, over the test set.

We now describe the role of each of the three (vector) components of our hyper-parameter vector.

4.3 Data weighting

We first describe the role of the data weighting hyper-parameter ω^{data}. The ith entry ω_i^{data} *weights* the squared error of the (u_i, y_i) data point in the loss by $e^{2\omega_i^{\text{data}}}$ (a positive number). If ω_i is small, then the ith data point has little effect on the model parameter, and vice versa.

By separately weighting the loss values of each data point, we can get the same effect as using a non-quadratic loss function. However, instead of having to decide on which loss function to use, we can automatically select the weights in a weighted square loss.

We let $\Omega^{\text{data}} = \{x \mid \mathbf{1}^T x = 0\}$, meaning we constrain the geometric mean of $\exp(\omega^{\text{data}})$ to be one. We describe some forms for the hyper-parameter regularization function r^{data}. We can regularize the hyper-parameter towards each data point being weighted equally ($\omega^{\text{data}} = 0$), *e.g.*, by using $r^{\text{data}}(\omega) = \lambda\|\omega\|_2^2$ or $r^{\text{data}}(\omega) = \lambda\|\omega\|_1$, where $\lambda > 0$. (Here λ is a hyper-hyper-parameter, since it scales a regularizer on the hyper-parameters.)

Proximal operator. We give details on a particular proximal operator that is needed later in the paper. Evaluating the proximal operator of $r^{\text{data}}(\omega) = \lambda\|\omega\|_2^2$ with $\Omega = \Omega^{\text{data}}$ at ν with step size t corresponds to solving the optimization problem

$$\begin{array}{ll} \text{minimize} & t\lambda\|\omega\|_2^2 + (1/2)\|\omega - \nu\|_2^2 \\ \text{subject to} & \mathbf{1}^T\omega = 0, \end{array}$$

with variable ω. The (linear) KKT system for this optimization problem is

$$\begin{bmatrix} (1+\lambda t)I & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \omega \\ y \end{bmatrix} = \begin{bmatrix} \nu \\ 0 \end{bmatrix},$$

with dual variable y. This linear system can be solved efficiently using block elimination [BV04, Appendix C].

4.4 Regularization

The next hyper-parameter subvector that we consider is the regularization hyper-parameter ω^{reg}. The regularization hyper-parameter affects the

$$\sum_{i=1}^{d} \exp(2\omega_i^{\text{reg}}) \|R_i\theta\|_F^2$$

term in the least squares objective. Each $\|R_i\theta\|_F^2$ term is meant to correspond to a measure of the complexity of θ. For example, if $R_i = I$, then the ith term is the sum of the squares of the singular values of θ. The entries of the regularization hyper-parameter correspond to the log of the weight on each regularization term. The regularization matrices can have many forms; here we give two examples.

Diagonal regularization. Separate diagonal regularization has the form

$$R_i = \mathbf{diag}(e_i), \quad i = 1, \ldots, n,$$

where e_i is the ith unit vector in $\mathbf{R}^n$. The ith regularization term corresponds to the sum of squares of the ith row of θ.

Graph regularization. The R_i can correspond to incidence matrices of graphs between the elements in each column of θ. Here the regularization hyper-parameter determines the relative importance of the regularization graphs.

4.5 Feature engineering

The final hyper-parameter is the feature engineering hyper-parameter ω^{feat}, which parametrizes the featurizer. The goal is to select a ω^{feat} which makes the output y roughly linear in $\phi(u, \omega^{\text{feat}})$. We assume that the input set $\mathcal{U}$ is a vector space in these examples.

Composition. Often ϕ is constructed as a feature generation chain, meaning it can be expressed as the composition of individual feature engineering functions $\phi_1, \ldots, \phi_l$, or

$$\phi = \phi_l \circ \cdots \circ \phi_1.$$

Often the last feature engineering function adds a constant, or $\phi_l(x) = (x, 1)$, so that the resulting predictor is affine.

4.5.1 Scalar feature engineering functions

We describe some scalar feature engineering functions $\phi : \mathbf{R} \to \mathbf{R}$, with the assumption that we could apply them elementwise (with different hyper-parameters) to vector inputs.

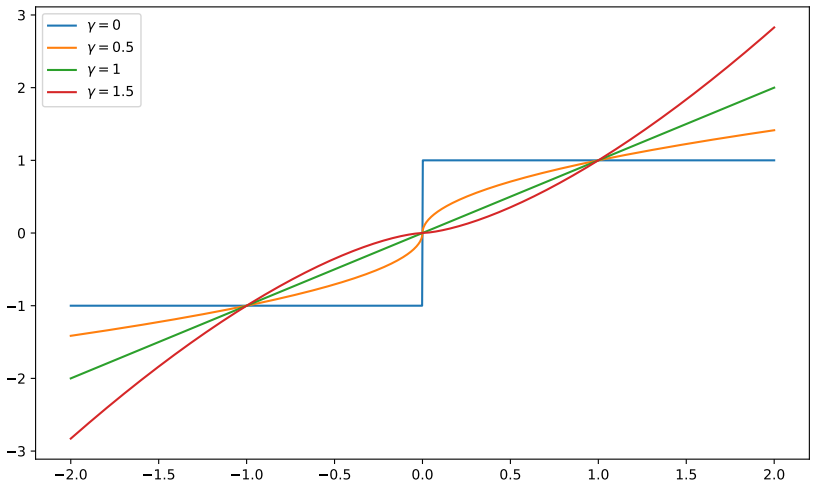

Figure 1: Examples of the power transform. Here the center $c = 0$.

Scaling. One of the simplest feature engineering functions is affine scaling, given by

$$\phi(x, (a, b)) = ax + b.$$

It is common practice in data fitting to standardize or whiten the data, by scaling each dimension with $a = 1/\mathbf{std}(x)$ and $b = -\mathbf{E}[x]/\mathbf{std}(x)$. Instead, with least squares auto tuning, we can select a and b based on the data.

Power transform. The *power transform* is given by

$$\phi(x, (c, \gamma)) = \mathbf{sgn}(x - c)|x - c|^{\gamma},$$

where $\mathbf{sgn}(x)$ is 1 if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$, the center $c \in \mathbf{R}$, and the scale $\gamma \in \mathbf{R}$. Here the hyper-parameters are $\gamma \in \mathbf{R}$ and the center $c \in \mathbf{R}$. For various values of γ and c, this function defines different transformations. For example, if $\gamma = 1$ and $c = 0$, this transform is the identity. If $\gamma = 0$, this transform determines whether x is to the right or left of the center, c. If $\gamma = 1/2$ and $c = 0$, this transform performs a symmetric square root. This transform is differentiable everywhere except when $\gamma = 0$. See figure 1 for some examples.

Polynomial splines. A spline is a piecewise polynomial function. Given a monotonically increasing knot vector $z \in \mathbf{R}^{k+1}$, a degree d, and polynomial coefficients $f_0, \ldots, f_{k+1} \in \mathbf{R}^{d+1}$, a spline is given by

$$\phi(x, (z, f_1, \ldots, f_{k+1})) = \begin{cases} \sum_{i=0}^{d} (f_0)_i x^i & x \in (-\infty, z_1) \\ \sum_{i=0}^{d} (f_j)_i x^i & x \in [z_j, z_{j+1}), \quad j = 1, \ldots, k, \\ \sum_{i=0}^{d} (f_{k+1})_i x^i & x \in [z_{k+1}, +\infty). \end{cases}$$

Here r^{feat} and Ω^{feat} can be used to enforce continuity (and differentiability) at $z_1, \ldots, z_{k+1}$.

4.5.2 Multi-dimensional feature engineering functions

Next we describe some multidimensional feature engineering functions.

Low rank. The featurizer can be a low rank transformation, given by

$$\phi(x, T) = Tx$$

where $T \in \mathbf{R}^{r \times n}$, and $r < n$. In practice, a common choice for T is the first few eigenvectors of the singular value decomposition of the data matrix. With least squares auto tuning, we can select T directly.

Neural networks. The featurizer ϕ can be a neural network; in this case ω^{feat} corresponds to the neural network's parameters.

Feature selection. We can select a fraction f of the features with

$$\phi(x) = \mathbf{diag}(a)x,$$

where $\Omega^{\text{feat}} = \{a \mid a \in \{0,1\}^{n_1}, \mathbf{1}^T a = \lfloor f n_1 \rfloor\}$. Here f is a hyper-hyper-parameter.

4.6 Test set and early stopping

There is a risk of overfitting to the validation set when there are a large number of hyper-parameters, since we are (almost) directly minimizing the validation loss. To detect this, we introduce a third dataset, the *test dataset*, and evaluate the fitted model on it *once* at the end of the algorithm. In particular, the validation loss throughout the algorithm need not be an accurate measure of the model's performance on new unseen data, especially when there are many hyper-parameters.

Early stopping. As a slight variation, to combat overfitting, we can calculate the loss of the fitted model on the test at each iteration, and halt the algorithm when the test loss begins to increase. This technique is sometimes referred to as early stopping [Pre98]. When performing early stopping, it is important to have a fourth dataset, the *final test dataset*, and evaluate on this set one time when the algorithm terminates. We have observed that this technique works very well in practice. However, we do not use early stopping in our numerical example, and instead run the algorithm until convergence.

Table 2: Small dataset.

Method	Hyper-parameters	Validation loss	Test error (%)
LS	0	1.77	13.0
LS + reg $\times$ 2	2	1.76	11.6
LS + reg $\times$ 3 + feat	4	1.54	6.1
LS + reg $\times$ 3 + feat + weighting	3504	1.54	6.0

Table 3: Full dataset.

Method	Hyper-parameters	Validation loss	Test error (%)
LS	0	1.74	10.3
LS + reg $\times$ 2	2	1.74	10.3
LS + reg $\times$ 3 + feat	4	1.53	4.7
LS + reg $\times$ 3 + feat + weighting	35004	1.53	4.8

5 Numerical example

In this section we apply our method of automatic least squares data fitting to the well-studied MNIST handwritten digit classification dataset [LBBH98]. We note that in the machine learning community, this task is considered "solved" by, *e.g.*, deep convolutional neural networks (*i.e.*, one can achieve arbitrarily low test error). We apply the ideas described in this paper to a large and small version of MNIST in order to show that standard least squares coupled with automatic tuning of additional hyper-parameters can achieve relatively high test accuracy, and can drastically improve the performance of standard least squares. In this example, it is also worth nothing that we do not perform any hyper-hyper-parameter optimization.

MNIST. The MNIST dataset is composed of 50,000 training data points, where each data point is a 784-vector (a 28×28 grayscale image flattened in row-order). There are $m = 10$ classes, corresponding to the digits 0–9. MNIST also comes with a test set, composed of 10,000 training points and labels. Since the task here is classification, we use the cross-entropy loss as the true objective function. The code used to produce these results has been made freely available at `www.github.com/sbarratt/lsat`. All experiments were performed on an unloaded Nvidia 1080 TI GPU using floats.

We create two MNIST datasets by randomly selecting data points. The *small dataset* has 3,500 training data points and 1,500 validation data points. The *full dataset* has 35,000 training data points and 15,000 validation data points. We evaluate four methods by tuning their hyper-parameters and then calculating the final validation loss and test error. The results are summarized in table 2 and table 3. We describe each method below, in order.

Base model. The simplest model is standard least squares, using the $n = 784$ image pixels as the feature vector. That is, we solve the optimization problem

$$\text{minimize} \quad \|X\theta - Y\|_F^2 + \|\theta\|_F^2.$$

Here we do no hyper-parameter tuning. We refer to this model as *LS* in the tables.

Regularization. To this simple model, we add a graph regularization term, and optimize the two regularization hyper-parameters. We define a graph on the length 784 feature vector, connecting two nodes if the pixels they correspond to are adjacent to each other in the original image. We then formed the incidence matrix of this graph, as described in §4.4, and denote it by $A \in \mathbf{R}^{1512\times 784}$ (it has 1512 edges). We then use the two regularization matrices:

$$R_1 = I, \qquad R_2 = A.$$

The matrix R_1 corresponds to standard ridge regularization, and R_2 measures the smoothness of the feature vector according to the graph we defined. This introduces 2 hyper-parameters, which separately weight R_1 and R_2. We do not use a hyper-parameter regularization function, and initialize $\omega^{\text{reg}} = (-2, -2)$. We optimize these two regularization hyper-parameters to minimize validation loss. We refer to this model as *LS + reg × 2* in the tables.

Feature engineering. For each label, we run the k-means algorithm with $k = 5$ on the training data points that have that label. From this, we get $km = 50$ centers, which we call *archetypes* and denote by $a_1, \ldots, a_{50} \in \mathbf{R}^{784}$. Define the function d such that it calculates how far x is from each of the archetypes, or

$$d(x)_i = \|x - a_i\|_2, \quad i = 1, \ldots, 50.$$

We use the feature engineering function

$$\phi(x) = (x, s(-d(x)/\exp(\sigma)), 1),$$

where σ is a feature engineering hyper-parameter, and s is the softmax function, which transforms a vector $z \in \mathbf{R}^n$ to a vector in the probability simplex, defined as

$$s(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

We introduce separate ridge regularization for the pixel features and the k-means features, *i.e.*, and still use the graph regularization on the pixel features; the regularization matrices are

$$R_1 = \begin{bmatrix} I & 0 & 0 \end{bmatrix}, \quad R_2 = \begin{bmatrix} 0 & I & 0 \end{bmatrix}, \quad R_3 = \begin{bmatrix} A & 0 & 0 \end{bmatrix}.$$

We do not use a hyper-parameter regularization function for ω^{feat}. We initialize σ to 3 and ω^{reg} to $(0, 0, 0)$, and optimize these four hyper-parameters. We refer to this model as *LS + reg × 3 + feat*.

(a) 7 (b) 8 (c) 4 (d) 4 (e) 5 (f) 9

Figure 2: Training data with the lowest weights. Captions correspond to labels.

(a) 8 (b) 4 (c) 4 (d) 4 (e) 9 (f) 3

Figure 3: Training data with the highest weights. Captions correspond to labels.

Data weighting. To LS + reg × 3 + feat, we add data weighting, as described in §4.3. We use $\Omega^{\text{reg}} = \{\omega \mid \mathbf{1}^T\omega = 0\}$ and $r^{\text{reg}}(\omega^{\text{reg}}) = (0.01)\|\omega^{\text{reg}}\|_2^2$. This introduces 3,500 hyper-parameters in the case of the small dataset, and 35,000 hyper-parameters for the large dataset. We use the initialization $\omega = 0$. We refer to this model as *LS + reg × 3 + feat + weighting*. This method performs the best on the small dataset, in terms of test error. On the full dataset, it performs slightly worse than the model without data weighting, likely because of the overfitting phenomenon discussed in §4.6. We show the training examples with the lowest data weights and the training examples with the highest weights in figure 2 and figure 3 respectively, on the small dataset. The training data points with low weights seem harder to classify (for example, (c) and (d) in figure 2 could be interpreted as nines).

6 Conclusion

The authors are currently writing a second paper, *Auto-Tuning Linear Quadratic Control and Estimation*, which will detail an application of the methods described in this paper to linear quadratic control and estimation.

Acknowledgements

Shane Barratt is supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1656518.

References

[ABB+99] E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and A. McKenney. *LAPACK Users' guide*. SIAM, 1999.

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard. Tensorflow: a system for large-scale machine learning. In *Proc. Operating Systems Design and Implementation*, volume 16, pages 265–283, 2016.

[AJS+18] B. Amos, I. Jimenez, J. Sacks, B. Boots, and Z. Kolter. Differentiable mpc for end-to-end planning and control. In *Proc. Advances in Neural Information Processing Systems*, pages 8299–8310, 2018.

[AK17] B. Amos and Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *Proc. International Conference on Machine Learning*, pages 136–145, 2017.

[Amo17] B. Amos. A fast and differentiable qp solver for pytorch. `https://github.com/locuslab/qpth`, 2017.

[AMP+19] A. Agrawal, A. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, A. Ganichev, J. Levenberg, M. Hong, R. Monga, and S. Cai. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. In *Proc. SysML Conf.*, 2019.

[Bar18] S. Barratt. On the differentiability of the solution to convex optimization problems. *arXiv preprint arXiv:1804.05098*, 2018.

[BB12] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, pages 281–305, 2012.

[BCN18] L. Bottou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[BD80] A. Björck and I. Duff. A direct method for the solution of sparse linear least squares problems. *Linear Algebra and its Applications*, 34:43–67, 1980.

[Ben00] Y. Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, pages 1889–1900, 2000.

[BM16] D. Belanger and A. McCallum. Structured prediction energy networks. In *Proc. International Conference on Machine Learning*, pages 983–992, 2016.

[BP14] A. Baydin and B. Pearlmutter. Automatic differentiation of algorithms for machine learning. *arXiv preprint arXiv:1404.7456*, 2014.

[BPC+11] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[BPRS18] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1–43, 2018.

[BS83] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22(3):317–330, 1983.

[BV04] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[BV18] S. Boyd and L. Vandenberghe. *Introduction to applied linear algebra: vectors, matrices, and least squares*. Cambridge University Press, 2018.

[BYM17] D. Belanger, B. Yang, and A. McCallum. End-to-end learning for structured prediction energy networks. In *Proc. International Conference on Machine Learning*, pages 429–439, 2017.

[CVBM02] O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46:131–159, 2002.

[dABPSA+18] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and Z. Kolter. End-to-end differentiable physics for learning and control. In *Proc. Advances in Neural Information Processing Systems*, pages 7178–7189, 2018.

[DAK17] P. Donti, B. Amos, and Z. Kolter. Task-based end-to-end model learning in stochastic optimization. In *Proc. Advances in Neural Information Processing Systems*, pages 5484–5494, 2017.

[DCHD90] J. Dongarra, J. Cruz, S. Hammarling, and I. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software*, 16(1):18–28, 1990.

[Dom12] J. Domke. Generic methods for optimization-based modeling. In *Proc. Intl. Conf. Artificial Intelligence and Statistics*, pages 318–326, 2012.

[DR56] J. Douglas and H. Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society*, 82(2):421–439, 1956.

[DR09] A. Dontchev and T. Rockafellar. Implicit functions and solution mappings. *Springer Monographs in Mathematics. Springer*, 208, 2009.

[EN99] R. Eigenmann and J. Nossek. Gradient based adaptive regularization. In *Proc. Neural Networks for Signal Processing*, pages 87–94, 1999.

[FDN08] C. Foo, C. Do, and A. Ng. Efficient multiple hyperparameter learning for log-linear models. In *Proc. Advances in Neural Information Processing Systems*, pages 377–384, 2008.

[FLFC16] J. Fu, H. Luo, J. Feng, and T. Chua. Distilling reverse-mode automatic differentiation (DrMAD) for optimizing hyperparameters of deep neural networks. *arXiv preprint arXiv:1601.00917*, 2016.

[Gau09] C. Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*, volume 7. Perthes et Besser, 1809.

[Gol65] G. Golub. Numerical methods for solving linear least squares problems. *Numerische Mathematik*, 7(3):206–216, 1965.

[GVL12] G. Golub and C. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[GW08] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[HO96] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proc. IEEE Intl. Conf. on Evolutionary Computation*, pages 312–317. IEEE, 1996.

[HS52] M. Hestenes and E. Stiefel. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS Washington, DC, 1952.

[Inn18] M. Innes. Don't unroll adjoint: Differentiating ssa-form programs. *arXiv preprint arXiv:1810.07951*, 2018.

[KSC07] S. Keerthi, V. Sindhwani, and O. Chapelle. An efficient method for gradient-based adaptation of hyperparameters in SVM models. In *Proc. Advances in Neural Information Processing Systems*, pages 673–680, 2007.

[LB17] S. Lall and S. Boyd. Lecture 11 notes for ee104, 2017.

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[LBGR16] J. Luketina, M. Berglund, K. Greff, and T. Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. In *Proc. International Conference on Machine Learning*, pages 2952–2960, 2016.

[LD18] J. Lorraine and D. Duvenaud. Stochastic hyperparameter optimization through hypernetworks. *arXiv preprint arXiv:1802.09419*, 2018.

[Leg05] A. Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes*. 1805.

[LFK18] C. Ling, F. Fang, and Z. Kolter. What game are we playing? end-to-end learning in normal and extensive form games. *arXiv preprint arXiv:1805.02777*, 2018.

[LFK19] C. Ling, F. Fang, and Z. Kolter. Large scale learning of agent rationality in two-player zero-sum games. 2019.

[LH95] C. Lawson and R. Hanson. *Solving least squares problems*, volume 15. SIAM, 1995.

[LM79] P. Lions and B. Mercier. Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis*, 16(6):964–979, 1979.

[LSAH98] J. Larsen, C. Svarer, L. Andersen, and L. Hansen. Adaptive regularization in neural network modeling. In *Prob. Neural Networks: Tricks of the Trade*, pages 113–132, 1998.

[Mar70] B. Martinet. Brève communication. régularisation d'inéquations variationnelles par approximations successives. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 4(R3):154–158, 1970.

[MBP12] J. Mairal, F. Bach, and J. Ponce. Task-driven dictionary learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):791–804, 2012.

[MDA15a] D. Maclaurin, D. Duvenaud, and R. Adams. Autograd: Effortless gradients in numpy. In *Proc. ICML 2015 AutoML Workshop*, 2015.

[MDA15b] D. Maclaurin, D. Duvenaud, and R. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proc. International Conference on Machine Learning*, pages 2113–2122, 2015.

[Moč75] J. Močkus. On Bayesian methods for seeking the extremum. In *Proc. Optimization Techniques Conf.*, pages 400–404, 1975.

[Nes13a] Y. Nesterov. Gradient methods for minimizing composite functions. *Mathematical Programming*, 140(1):125–161, 2013.

[Nes13b] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[NW06] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[PB14] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends® in Optimization*, 1(3):127–239, 2014.

[Ped16] F. Pedregosa. Hyperparameter optimization with approximate gradient. In *Proc. International Conference on Machine Learning*, pages 737–746, 2016.

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[Pol87] B. Polyak. *Introduction to optimization*. Optimization Software, 1987.

[Pre98] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the Trade*, pages 55–69. Springer, 1998.

[PS75] C. Paige and M. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM journal on Numerical Analysis*, 12(4):617–629, 1975.

[PS82] C. Paige and M. Saunders. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, 1982.

[Ras04] C. Rasmussen. *Gaussian Processes in Machine Learning*. Springer, 2004.

[RZYU18] M. Ren, W. Zeng, B. Yang, and R. Urtasun. Learning to reweight examples for robust deep learning. *arXiv preprint arXiv:1803.09050*, 2018.

[Sho85] N. Shor. *Minimization methods for non-differentiable functions*, volume 3. Springer Science & Business Media, 1985.

[SK10] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[SLA12] J. Snoek, H. Larochelle, and R. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proc. Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[Smi95] S. Smith. Differentiation of the cholesky algorithm. *Journal of Computational and Graphical Statistics*, 4(2):134–147, 1995.

[Spe80] B. Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, 1980.

[vMMW18] B. van Merriënboer, D. Moldovan, and A. Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Proc. Advances in Neural Information Processing Systems*, pages 6259–6268, 2018.

[WCV11] S. Walt, S. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[Wen64] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

A Derivation of gradient of least squares solution

Consider the least squares solution map $\phi : \mathbf{R}^{k\times n} \times \mathbf{R}^{k\times m} \to \mathbf{R}^{n\times m}$, given by

$$\theta = \phi(A, B) = (A^TA)^{-1}A^TB.$$

We are interested in the linear operator $D_A\phi(A, B) : \mathbf{R}^{k\times n} \to \mathbf{R}^{n\times m}$, *i.e.*, the derivative of ϕ with respect to A, and the linear operator $D_B\phi(A, B) : \mathbf{R}^{k\times m} \to \mathbf{R}^{n\times m}$, *i.e.*, the derivative of ϕ with respect to B.

Derivative with respect to A. We have that

$$D_A\phi(A, B)(\Delta A) = (A^TA)^{-1}\Delta A^TB - (A^TA)^{-1}(\Delta A^TA + A^T\Delta A)\theta,$$

since

$$\begin{aligned}
\phi(A + \Delta A, B) &= ((A + \Delta A)^T(A + \Delta A))^{-1}(A^TB + \Delta A^TB) \\
&\approx (A^TA)^{-1}\left(I - \Delta A^TA(A^TA)^{-1} - A^T\Delta A(A^TA)^{-1}\right)(A^TB + \Delta A^TB) \\
&\approx \phi(A, B) + (A^TA)^{-1}\Delta A^TB - (A^TA)^{-1}(\Delta A^TA + A^T\Delta A)\theta,
\end{aligned}$$

where we used the approximation $(X + Y)^{-1} \approx X^{-1} - X^{-1}YX^{-1}$ for Y small, and dropped higher order terms. Suppose $f = \psi \circ \phi$ and $C = (A^TA)^{-1}\nabla_\theta\psi$ for some $\psi : \mathbf{R}^{n\times m} \to \mathbf{R}$. Then the linear map $D_Af(A, B)$ is given by

$$\begin{aligned}
D_Af(A, B)(\Delta A) &= D_{\theta^{\text{ls}}}\psi(D_A\phi(A, B)(\Delta A)) \\
&= \mathbf{tr}\left(\nabla_\theta\psi^T\left((A^TA)^{-1}\Delta A^TB - (A^TA)^{-1}(\Delta A^TA + A^T\Delta A)\theta\right)\right) \\
&= \mathbf{tr}((BC^T - A\theta C^T - AC\theta^T)^T\Delta A),
\end{aligned}$$

from which we conclude that $\nabla_Af = (B - A\theta)C^T - AC\theta^T$.

Derivative with respect to B. We have that $D_B\phi(A, B)(\Delta B) = (A^TA)^{-1}A^T\Delta B$, since

$$\phi(A, B + \Delta B) = \phi(A, B) + (A^TA)^{-1}A^T\Delta B.$$

Suppose $f = \psi \circ \phi$ for some $\psi : \mathbf{R}^{n\times m} \to \mathbf{R}$ and $C = (A^TA)^{-1}\nabla_\theta\psi$. Then the linear map $D_Bf(A, B)$ is given by

$$\begin{aligned}
D_Bf(A, B)(\Delta B) &= D_{\theta^{\text{ls}}}\psi(D_B\phi(A, B)(\Delta B)) \\
&= \mathbf{tr}(\nabla_\theta\psi^T(A^TA)^{-1}A^T\Delta B) \\
&= \mathbf{tr}((AC)^T\Delta B),
\end{aligned}$$

from which we conclude that $\nabla_Bf = AC$.

B Derivation of stopping criterion

The optimality condition for minimizing $\psi + r$ is

$$\nabla_\omega \psi + g = 0,$$

where $g \in \partial_\omega r$, the subdifferential of r. We have that

$$t^k \partial r(\omega^{k+1}) + \omega^{k+1} - \omega^k + t^k \nabla_{\omega^k} \psi = 0,$$

which implies that

$$(\omega^k - \omega^{k+1})/t^k - g^k \in \partial r(\omega^{k+1}).$$

Therefore, the optimality condition for ω^{k+1} is

$$(\omega^k - \omega^{k+1})/t^k + (g^{k+1} - g^k) = 0,$$

which leads to the stopping criterion (7).

C PyTorch implementation

We implemented the forward/backward methods of the least squares in PyTorch. We compute the forward pass using a Cholesky factorization of the Gram matrix $A^T A$, which is cached and reused in the backward pass. The code is simply:

```
import torch

class DenseLeastSquares(torch.autograd.Function):
    @staticmethod
    def forward(ctx, A, B):
        with torch.no_grad():
            u = torch.cholesky(A.t() @ A, upper=True)
            theta = torch.potrs(A.t() @ B, u)
        ctx.save_for_backward(A, B, theta, u)

        return theta

    @staticmethod
    def backward(ctx, dtheta):
        A, B, theta, u = ctx.saved_tensors

        with torch.no_grad():
            C = torch.potrs(dtheta, u)
            Cthetat = C @ theta.t()
            dA = B @ C.t() - A @ (Cthetat + Cthetat.t())
            dB = A @ C

        return dA, dB
```

D Tensorflow implementation

We implemented the forward/backward methods of least squares in Tensorflow. We compute the forward pass using a Cholesky factorization of the Gram matrix A^TA, which is cached and reused in the backward pass. The code is simply:

```
import tensorflow as tf

@tf.custom_gradient
def lstsq(A,B):
    AtA = tf.transpose(A)@A
    chol = tf.linalg.cholesky(AtA)
    theta = tf.linalg.cholesky_solve(chol, tf.transpose(A)@B)
    def grad(dtheta):
        C = tf.linalg.cholesky_solve(chol, dtheta)
        Cthetat = C@tf.transpose(theta)
        dA = B@tf.transpose(C)-A@(Cthetat+tf.transpose(Cthetat))
        dB = A@C
        return dA, dB
    return theta, grad
```