ORIGINAL ARTICLE

# Least Squares Auto-Tuning

Shane T. Barratt and Stephen P. Boyd

Electrical Engineering Department, Stanford University, Stanford, CA, 94306.

**ARTICLE HISTORY**
Compiled March 6, 2020

**ABSTRACT**
Least squares auto-tuning automatically finds hyper-parameters in least squares problems that minimize another (true) objective. The least squares tuning optimization problem is nonconvex, so it cannot be efficiently solved. This article presents a powerful proximal gradient method for least squares auto-tuning, which can be used to find good, if not the best, hyper-parameters for least squares problems. The application of least squares auto-tuning to data fitting is discussed. Numerical experiments on a classification problem using the MNIST dataset demonstrate the effectiveness of the method; it is able to cut the test error of standard least squares in half. This article is accompanied by an open source implementation.

**KEYWORDS**
Least squares; hyper-parameter optimization; proximal gradient method

## 1. Introduction

Since its introduction over 200 years ago by Legendre (1805) and Gauss (1809), the method of least squares has been one of the most widely employed computational techniques in many fields, including machine learning and statistics, signal processing, control, robotics, and finance (Boyd and Vandenberghe 2018). Its wide application primarily comes from the fact that it has a simple closed-form solution, it is easy to understand, and very efficient and stable algorithms for computing its solution have been developed (Lawson and Hanson 1995; Golub and Van Loan 2012).

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, and 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,

CONTACT S. T. Barratt. Email: sbarratt@stanford.edu

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. 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, is referred to as *least squares tuning*. It is very widely used, and can be extremely effective in practice.

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

One of the main contributions of this article 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, least squares auto-tuning can simultaneously adjust hyper-parameters in the feature generation chain. Through several examples, it is shown that ordinary least squares, used for over 200 years, coupled with automated hyper-parameter tuning, can be an effective method for data fitting.

The method described 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. An implementation has been developed 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.

This article has 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 (§3.4); the description is mirrored by an open-source implementation (§3.5). The second contribution is the method of least squares auto-tuning, which can automatically tune hyper-parameters in least squares problems (§3.2). The final contribution is the application of least squares auto-tuning to data fitting (§4).

## 2. Related work

This work mainly falls at the intersection of two fields: automatic differentiation and hyper-parameter optimization. This section reviews related work.

### 2.1. *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 (Wengert 1964; Speelpenning 1980). 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 (Baur and Strassen 1983; Griewank and Walther 2008). This means that an optimization algorithm can obtain derivatives of the function it is optimizing as fast as computing the function itself, which explains the proliferation of gradient-based minimization methods (Baydin et al. 2018; Bottou, Curtis, and Nocedal 2018).

There are many popular implementations of AD, and they generally fall into one of two categories. The first category is trace-based AD systems, which trace computations at runtime as they are executed; popular ones include PyTorch (Paszke et al. 2019), Tensorflow eager (Agrawal

et al. 2019b), and autograd (Maclaurin, Duvenaud, and Adams 2015a). 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 (Abadi et al. 2016), Tangent (van Merriënboer, Moldovan, and Wiltschko 2018), and Zygote (Innes 2018).

### 2.2. *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 one can differentiate the solution map (Dontchev and Rockafellar 2009). For convex optimization problems that satisfy strong duality, the solution map is given by the set of solutions to the Karush-Kuhn-Tucker (KKT) conditions, which can in some cases be differentiated using the implicit function theorem (Barratt 2018; Agrawal et al. 2019a). This idea has been applied to convex quadratic programs (Amos and Kolter 2017), convex optimization problems (Agrawal et al. 2019c), stochastic optimization (Donti, Amos, and Kolter 2017), games (Ling, Fang, and Kolter 2018, 2019), physical systems (de Avila Belbute-Peres et al. 2018), control (Amos et al. 2018; Agrawal et al. 2019d), repairing convex optimization problems (Barratt, Angeris, and Boyd 2020), structured inference (Belanger and McCallum 2016; Belanger, Yang, and McCallum 2017), and black-box optimization (Amos and Yarats 2019). In machine learning, these techniques were originally applied to neural networks (Larsen et al. 1998; Eigenmann and Nossek 1999) and ridge regression (Bengio 2000), and more recently to lasso (Mairal, Bach, and Ponce 2012), support vector machines (Chapelle et al. 2002), and log-linear models (Keerthi, Sindhwani, and Chapelle 2007; Foo, Do, and Ng 2008) Two notable AD implementations of these methods are the PyTorch implementation `qpth`, which can compute derivatives of the solution map of quadratic programs (Amos 2017), and `cvxpylayers`, which can compute derivatives of the solution map of convex optimization problems (Agrawal et al. 2019c).

### 2.3. *Variable projection method*

The variable projection method (Golub and Pereyra 1973, 2003; Chen et al. 2018) is a method for the approximate solution of nonlinear least squares problems whose residual is linear in some of the variables but nonlinear in the rest. It works by partially minimizing the objective with respect to the linear variable since it is a linear least squares problem and writing the objective as a function of the remaining variables. The objective can then be minimized using standard first or second-order optimization methods.

The variable projection method is similar to least squares auto-tuning, but indeed fundamentally different. Least squares auto-tuning parametrizes a least squares problem by hyper-parameters, which is then solved to generate the parameters, and then the parameters are evaluated using a separate loss function. In the variable projection method, the parameters and hyper-parameters are chosen together to minimize a single objective.

### 2.4. *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 (Domke 2012; Baydin and Pearlmutter 2014). 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 (Maclaurin, Duvenaud, and Adams 2015b; Fu et al. 2016; Lorraine and Duvenaud 2018),

and even to learn weights on individual data points (Ren et al. 2018). 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 feasible way.

### 2.5. *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 (Bergstra and Bengio 2012), Bayesian optimization (Močkus 1975; Rasmussen 2004; Snoek, Larochelle, and Adams 2012), and covariance matrix adaptation (Hansen and Ostermeier 1996).

## 3. Least squares auto-tuning

This section describes the least squares tuning problem, the method to automaticcaly solve the problem, and the 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. Throughout this article it is assumed that $A(\omega)$ has linearly independent columns for $\omega \in \Omega$, 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) generalized inverse. Solving a least squares problem for a given hyper-parameter vector corresponds to computing $\theta^{\text{ls}}(\omega)$. The least squares solution $\theta^{\text{ls}}$ will be though of as a function mapping the hyper-parameter $\omega \in \Omega$ to a parameter $\theta^{\text{ls}}(\omega) \in \mathbf{R}^{n\times m}$.

#### *3.1.1. Multi-objective least squares*

In many applications one has multiple least squares objectives (Boyd and Vandenberghe 2004, §4.7). 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

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

The sequel will often write least squares problems in the form (3), and it will be assumed 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 this dependence is omitted.

#### *3.1.2. Solving the least squares problem*

For a given value of $\omega$, there are many ways to solve the least squares problem (1), including dense or sparse QR or other factorizations (Golub 1965; Björck and Duff 1980), iterative methods such as CG or LSQR (Hestenes and Stiefel 1952; Paige and Saunders 1982), and many others. Very efficient libraries for computing the least squares solution that target multiple CPUs or one or more GPUs have also been developed (Dongarra et al. 1990; Anderson et al. 1999). This problem is separable across the columns of $\theta$, *i.e.*, the problem splits into $m$ independent least squares problems with vector variables and a common coefficient matrix.

A few more details are provided here for two of these methods. The first case considered is where $A(\omega)$ and $B(\omega)$ are stored and manipulated as dense matrices. One attractive option for a GPU implementation is to form the Gram matrix $G = A^T A$, along with $H = A^T B$. 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 $\theta^{\mathrm{ls}}$, one 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 $n^2 m$ flops, and then solve the triangular equation $L^T \theta^{\mathrm{ls}} = Y$, which costs order $n^2 m$ flops. Overall, the complexity of solving a dense least squares problem is order $kn(n+m)$. The numerical stability of this method depends heavily on the condition number of $A$ (see, *e.g.*, Higham (2002, §20.5)).

The other case for which more detail is given 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, one can evaluate $A(\omega)u$ for any $u \in \mathbf{R}^n$, and $A(\omega)^T v$ for any $v \in \mathbf{R}^k$. (This is the so-called *matrix-free* representation.) CG or LSQR can be used to solve the least squares problem, in parallel for each column of $\theta$. 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 (Hestenes and Stiefel 1952; Paige and Saunders 1982).

### 3.2. *Least squares tuning problem*

In a *least squares tuning problem*, the goal is to choose the hyper-parameters to achieve some goal. This is formalized as the problem

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

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*. Infinite values of $r$ (and therefore $F$) are used to encode constraints on the hyper-parameter $\omega$, and it will be assumed that $r(\omega)$ is defined as $\infty$ for $\omega \notin \Omega$. A least squares tuning problem is specified by

the functions $A$, $B$, $\psi$, and $r$. Some additional assumptions will be made 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. Parameters that may appear in the hyper-parameter regularization function will be referred to 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, $\theta$ and $\omega$ 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 the above formulation (5).

### 3.3. *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. (One important exception is when $\omega$ is a scalar and $\Omega$ is an interval, in which case one can simply evaluate $F(\omega)$ on a grid of values over $\Omega$.) This means that one needs to resort a local optimization or heuristic method in order to (approximately) solve it.

It will be assumed that $A$ and $B$ are differentiable in $\omega$, which implies that $\theta^{\mathrm{ls}}$ is differentiable in $\omega$, since the mapping from $\omega$ to $\theta^{\mathrm{ls}}$ is differentiable. It will also be assumed that $\psi$ is differentiable, which implies that the true objective $\psi(\theta^{\mathrm{ls}}(\omega))$ is differentiable in the hyper-parameters $\omega$. 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 (Douglas and Rachford 1956; Lions and Mercier 1979; Shor 1985; Boyd et al. 2011).

For completeness, the proximal gradient method (which stems from the proximal point method (Martinet 1970); for a modern reference see Nesterov (2013)) is described. The proximal gradient method is 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

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

It is assumed that the argmin exists; when it is not unique, any minimize can be chosen In order to use the proximal gradient method, one needs 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 $\Omega$, the proximal operator of $tr$ is Euclidean projection onto $\Omega$, and the proximal gradient method coincides with the projected gradient method.

There are many ways to choose the step size. The following simple adaptive scheme, borrowed

from Lall and Boyd (2017), is adopted. 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 the step size is increased a bit and accept the update. If, on the other hand, the function value increases from iteration $k$ to $k+1$, or $F(\omega^{k+1}) > F(\omega^k)$, the step size is decreased substantially and the update rejected, *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 $t^{k+1} = (1/2)t^k$. More sophisticated step size selection methods exist (see, *e.g.*, the line search methods for the Goldstein or Armijo conditions (Nocedal and Wright 2006)).

By default, the method is run 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 1 below.

---

**Algorithm 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 $n_{\mathrm{iter}}$, tolerance $\epsilon$.

**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**

---

The proximal gradient method is guaranteed to converge to the global minimum when the composition $\psi \circ \theta^{\mathrm{ls}}$ and $r$ are convex. However, since this problem is in general nonconvex, this method is subject to the same convergence properties as the proximal gradient method on nonconvex problems. That is, under certain assumptions, the method converges to a stationary point (Beck 2017, Theorem 10.15).

It is emphasized that many other methods can be used to (approximately) solve the least squares tuning problem (5); the proximal gradient method in particular was described only for completeness.

### 3.4. *Computing the gradient*

*Note.* In principle, one could calculate the gradient by directly differentiating the linear algebra routines used to solve the least squares problem (Smith 1995). We, however, work out formulas

for computing the gradient in closed-form, that work in the case of sparse $A$ and allow for a more efficient implementation.

To compute $g = \nabla_\omega \psi(\theta^{\mathrm{ls}}(\omega)) \in \mathbf{R}^p$ one can make use of the chain rule for $f = \psi \circ \theta^{\mathrm{ls}}$. Assume that $\theta = \theta^{\mathrm{ls}}(\omega)$ has been computed. First, one computes $\nabla_\theta \psi(\theta) \in \mathbf{R}^{n\times m}$, and then forms

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

(Here the dependence on $\omega$ has been dropped, *i.e.*, $A = A(\omega)$.) If $A$ is stored as a dense matrix, observe that $G = A^T A$ and its factorization have already been computed (to evaluate $\theta$; see §3.1.2), so this step involves a back-solve. If $A$ is represented as an abstract operator, one can evaluate each column of $C$ (in parallel) using an iterative method.

It can be shown (see Appendix A) that the gradients of $\psi$ with respect to $A$ and $B$ are

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

(Again, the dependence on $\omega$ has been dropped.)

In the case of $A$ dense, one can explicitly form $\nabla_A \psi$ and $\nabla_B \psi$, since they are the same size as $A$ and $B$, which are already 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, one can explicitly form the matrix $\nabla_B \psi$, but can not form $\nabla_A \psi$, which by assumption is too large to store, since it is the size of $A$. Instead, it is assumed that $\omega$ only affects $A$ at a subset of its entries, $\Gamma$, *i.e.*, $A_{ij}(\omega) = 0$ for all $i, j \notin \Gamma$, and for all $\omega \in \Omega$. By doing this, $\nabla_A \psi$ is restricted to have the same sparsity pattern as $A$, meaning only $(\nabla_A \psi)_{ij}$ for $i, j \in \Gamma$, needs to be computed:

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

where $b_i$ is the $i$th row of $B$, $a_i$ is the $i$th row of $A$, $c_j$ is the $j$th column of $C$, and $(C\theta^T)_j$ is the $j$th 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 $\nabla_B \psi$. When $A$ is stored as a dense matrix, one first evaluates $\nabla_\omega A_{ij} \in \mathbf{R}^p$ and $\nabla_\omega B_{ij} \in \mathbf{R}^p$, the gradients of the problem data entries with respect to $\omega$. If these gradients are all dense, one needs to store $k(n+m)$ vectors in $\mathbf{R}^p$; but generally, they are quite sparse. (There is an explanation below of how to take advantage of the sparsity in §3.5.) Finally,

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

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

When $A$ is large and sparse, one only need to compute the inner product at the entries of $A$ that are affected by $\omega$, *i.e.*,

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

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

Table 1.: GPU timings.

| $k$ | $n$ | $m$ | $p$ | Compute $\psi$ | 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. *Implementation*

The equations in §3.4 for computing $g$ do not directly lend themselves to an implementation. For example, it is necessary to compute $\nabla_\theta \psi$, $\nabla_\omega A_{ij}$ and $\nabla_\omega B_{ij}$, which depend on the form of $\psi$, $A$, and $B$. Also, it would be nice to take advantage of the (potential) sparsity of these derivatives. Libraries for automatic differentiation, *e.g.*, PyTorch (Paszke et al. 2019) and Tensorflow (Abadi et al. 2016), can be used to automatically (and efficiently) compute $g$ given $\psi$, $A$, 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 the context of this article, $\psi$ is represented as a function of $\omega$, 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 $g$, it is necessary to implement an operation that solves the least squares problem (2) and computes its gradients (8). An operation `lstsq(A,B)` that does exactly this has been implemented in both PyTorch and Tensorflow, in both the dense and sparse case.

There are several advantages to 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 $\psi$, $A$, and $B$, since they can be represented as compositions of (the many) pre-defined operations in these libraries.

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 $\omega$ 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, only floats are needed.) The percentage of time spent on the Cholesky factorization of the Gram matrix is also shown.

### 3.6. *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

(Boyd and Vandenberghe 2018, Chapter 16)

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

where $q = A(\omega)\theta - B(\omega)$ is the residual. From here on, 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 (Lawson and Hanson 1995). 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 (Paige and Saunders 1975).

The true objective function becomes a function of both $\theta$ and $\nu$. Suppose the gradients $\nabla_\theta \psi$ and $\nabla_\nu \psi$ are known. 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 $\psi$ 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, their gradients can be computed only at the nonzero elements, in a similar fashion to the procedure described in §3.4.

## 4. Least squares data fitting

The previous section described the general idea of least squares auto-tuning. In this section, and for the remainder of the article, least squares auto-tuning is applied to data fitting.

### 4.1. *Least squares data fitting*

In a data fitting problem, one has *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*, one fits 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 ($\Omega^{\text{feat}}$ is the set of allowed 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). This predictor is linear in the output of the featurizer.

To select the model parameters, a least squares problem is solved 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 ($\Omega^{\text{data}}$ is the set of allowed 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 ($\Omega^{\text{reg}}$ is the set of allowed regularization hyper-parameters).

The overall hyper-parameter is denoted by

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

The roles of each hyper-parameter are described in more detail below; but here note that $\omega^{\text{data}}$ scales the individual training data examples, $\omega^{\text{feat}}$ are hyper-parameters in the featurizer, and $\omega^{\text{reg}}$ are hyper-parameters that scale each of the regularizers. Assume that the hyper-parameter regularization function is separable, meaning it has the form

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

This means that the proximal operator is separable (Parikh and Boyd 2014). The functions $r^{\text{feat}}$, $r^{\text{data}}$, and $r^{\text{reg}}$ are the regularization functions for the featurizer, data weighting, and regularization hyper-parameters.

### 4.2. *True objective function*

Suppose there is also *validation data* composed of inputs $u_1^{\text{val}}, \ldots, u_{N_{\text{val}}}^{\text{val}} \in \mathcal{U}$ and outputs $y_1^{\text{val}}, \ldots, y_{N_{\text{val}}}^{\text{val}} \in \mathbf{R}^m$. First, the predictions are formed:

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

where the featurizer is fixed, or $\phi(u) = \phi(u, \omega^{\text{data}})$. The true objective $\psi$ in least squares data fitting corresponds to the average loss of the 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). More sophisticated true objective functions are possible (see, *e.g.*, the $K$-fold cross validation loss in §4.6). The focus in this article is on simple out-of-sample validation for simplicity.

The setting 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 $i$th 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 $i$th unit vector in $\mathbf{R}^m$), and the output represents a class label.

#### *4.2.1. 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 $\pi$ 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:

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

  where $M$ is a hyper-hyper-parameter.
- *Bisquare*. The bisquare penalty is a robust penalty that is constant for large residuals:

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

  where $M$ is a hyper-hyper-parameter.

#### *4.2.2. Classification penalty function*

For classification, the prediction $\hat{y} \in \mathbf{R}^m$ is associated with the probability distribution on the $m$ label values given by

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

The prediction $\hat{y}$ can be interpreted as a distribution on the labels of $y$, given $x$.

The *cross-entropy loss* will be used 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.

### 4.3. *Data weighting*

This section describes the role of the data weighing hyper-parameter $\omega^{\text{data}}$. The $i$th entry $\omega_i^{\text{data}}$ *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 $\omega_i$ is small, then the $i$th data point has little effect on the model parameter, and vice versa.

Separately weighing the loss values of each data point has the same effect as using a non-quadratic loss function. However, instead of having to decide on which loss function to use, the weights are automatically selected in a weighted square loss. Auto-tuning of data weights is not new; in 1961, Lawson considered the problem of $\ell_\infty$ approximation by iteratively updating the weights of data points in least squares problems (Lawson 1961). Later, this idea was extended by Rice and Usow to data fitting problems with $p$-norms, and coined as the iteratively re-weighted least squares algorithm (Rice and Usow 1968).

The constraint set $\Omega^{\text{data}} = \{x \mid \mathbf{1}^T x = 0\}$ is considered, meaning the geometric mean of $\exp(\omega^{\text{data}})$ is constrainted to be equal to one. The hyper-parameter can be regularized towards equal weighting ($\omega^{\text{data}} = 0$), *e.g.*, by using $r^{\text{data}}(\omega) = (\lambda/2)\|\omega\|_2^2$ or $r^{\text{data}}(\omega) = \lambda\|\omega\|_1$, where $\lambda > 0$. (Here $\lambda$ is a hyper-hyper-parameter, since it scales a regularizer on the hyper-parameters.)

The details of a particular proximal operator is given below that is needed later in the article. Evaluating the proximal operator of $r^{\text{data}}(\omega) = (\lambda/2)\|\omega\|_2^2$ with $\Omega = \Omega^{\text{data}}$ at $\nu$ with step size $t$ corresponds to solving the optimization problem

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

with variable $\omega$. This optimization problem has the analytical solution

$$\omega^\star = \frac{1}{1 + t\lambda}(\nu - \mathbf{1}^T\nu/p).$$

### 4.4. *Regularization*

The next hyper-parameter subvector is the regularization hyper-parameter $\omega^{\text{reg}}$. 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 $\theta$. For example, if $R_i = I$, then the $i$th term is the sum of the squares of the singular values of $\theta$. 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 two examples are given.

Separate diagonal regularization has the form

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

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

The $R_i$s could correspond to incidence matrices of graphs between the elements in each column of $\theta$. 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 $\omega^{\text{feat}}$, which parametrizes the featurizer. The goal is to select a $\omega^{\text{feat}}$ which makes the output $y$ roughly linear in $\phi(u, \omega^{\text{feat}})$. The input set $\mathcal{U}$ is assumed to be a vector space in these examples.

Often $\phi$ 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*

This section describes some scalar feature engineering functions $\phi : \mathbf{R} \to \mathbf{R}$, with the assumption that they could be applied elementwise (with different hyper-parameters) to vector inputs.

*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 $a = 1/\sigma$ and $b = -\mu/\sigma$, where $\mu$ is the mean of $x$ and $\sigma$ is the standard deviation of $x$. Instead, with least squares auto tuning, $a$ and $b$ are selected 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 $\gamma$ 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 $\gamma = 0$. The power transform has also found use in computational photography, where it is known as gamma correction (Hurter and Driffield 1890). 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 $r^{\text{feat}}$ and $\Omega^{\text{feat}}$ can be used to enforce continuity (and differentiability) at $z_1, \ldots, z_{k+1}$.

#### *4.5.2. Multi-dimensional feature engineering functions*

This section describes some multidimensional feature engineering functions.

*Low rank.* The featurizer $\phi$ can be a low rank transformation, given by

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

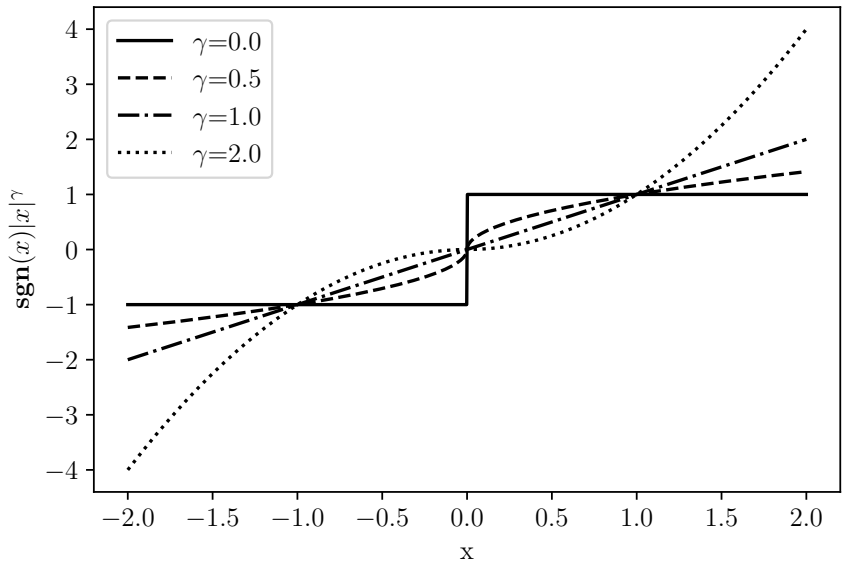


Figure 1.: Examples of the power transform. Here the center $c = 0$.

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

*Neural networks.* The featurizer $\phi$ can be a neural network; in this case $\omega^{\text{feat}}$ corresponds to the neural network's parameters.

*Feature selection.* A fraction $f$ of the features can be selected with

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

where $\Omega^{\text{feat}} = \{a \mid a \in \{0,1\}^{n_1}, \mathbf{1}^T a = \lfloor f n_1 \rfloor\}$ and $\lfloor \cdot \rfloor$ is the floor function. 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 least squares auto-tuning directly minimizes the validation loss. To detect this, a third dataset, the *test dataset*, can be introduced fitted model can be evaluated 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.

As a slight variation, to combat overfitting, the loss of the fitted model on the test set can be calculated at each iteration, and the algorithm halted when the test loss begins to increase. This technique is sometimes referred to as early stopping (Prechelt 1998). When performing early stopping, it is important to have a fourth dataset, the *final test dataset*, and evaluate on this dataset one time when the algorithm terminates. This technique has been observed to work very well in practice. However, in the numerical example, no early stopping is used and the algorithm is run until convergence.

Another way to create more robust models is to use the loss averaged over multiple training and validation datasets as the true objective. Mltiple training and validation dataset pairs from

Table 2.: Results of MNIST experiment on small dataset.

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

a single dataset can be constructed via $K$-fold cross validation, *i.e.*, by partitioning the data into $K$ equally sized groups, and letting each training dataset be all groups but one and each validation dataset be the data points not in the respective training set (Hastie, Tibshirani, and Friedman 2009, §7.10). A separate model is then fit to each training set, evaluate it on the validation dataset, and then compute the gradient of the validation loss with respect to the hyper-parameters. Having done this for each fold, the gradients can be averaged to recover the gradient of the cross-validation loss with respect to the hyper-parameters. Since the exact same techniques described in this manuscript can be applied to $K$-fold cross validation, this idea will not be pursued further in this manuscript.

## 5. Numerical example

In this section the method of automatic least squares data fitting is applied to the well-studied MNIST handwritten digit classification dataset (LeCun et al. 1998). In the machine learning community, this task is considered 'solved', *i.e.*, *i.e.*, one can achieve arbitrarily low test error, by, *e.g.*, deep convolutional neural networks (Ciregan, Meier, and Schmidhuber 2012). The ideas described in this article are applied 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 noting that no hyper-hyper-parameter optimization was performed.

### 5.1. *MNIST*

The MNIST dataset is composed of 50,000 training data points, where each data point is a 784-vector (a $28 \times 28$ 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, the cross-entropy loss is used 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.

Two MNIST datasets are created 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. The four methods are evaluated by tuning their hyper-parameters and then calculating the final validation loss and test error. The results are summarized in tables 2 and 3. Each method is described below, in order.

Table 3.: Results of MNIST experiment on full dataset.

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

### 5.2. *Base model*

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

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

Here no hyper-parameter tuning is performed. This model is referred to as *LS* in the tables.

### 5.3. *Regularization*

To this simple model, a graph regularization term is added, and optimize the two regularization hyper-parameters. A graph on the length 784 feature vector is constructed, connecting two nodes if the pixels they correspond to are adjacent to each other in the original image. The incidence matrix of this graph was formed, as described in §4.4, and denoted by $A \in \mathbf{R}^{1512\times 784}$ (it has 1512 edges). The two regularization matrices used were:

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

The matrix $R_1$ corresponds to standard ridge regularization, and $R_2$ measures the smoothness of the feature vector according to the graph described above. This introduces 2 hyper-parameters, which separately weight $R_1$ and $R_2$. Here, no hyper-parameter regularization function is used, and $\omega^{\text{reg}}$ is initialized to $(-2, -2)$. These two regularization hyper-parameters are optimized to minimize validation loss. This model is referred to as *LS + reg × 2* in the tables.

### 5.4. *Feature engineering*

For each label, the $k$-means algorithm was run with $k = 5$ on the training data points that have that label. This results in $km = 50$ centers, referred to as *archetypes*, and denoted 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

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

The following feature engineering function is considered:

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

(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.

where $\sigma$ 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

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

WSeparate ridge regularization is introduced for the pixel features and the $k$-means features, and still use the graph regularization on the pixel features; the regularization matrices are

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

No hyper-parameter regularization function is used for $\omega^{\text{feat}}$. Here $\sigma$ is initialized to 3 and $\omega^{\text{reg}}$ to $(0, 0, 0)$, and optimize these four hyper-parameters. This model is referred to as *LS + reg × 3 + feat*.

### 5.5. *Data weighting*

To LS + reg × 3 + feat, data weighting is added, as described in §4.3. The constraint set employed is $\Omega^{\text{reg}} = \{\omega \mid \mathbf{1}^T\omega = 0\}$ 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. The initialization was $\omega = 0$. This model is referred to 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. Training examples with the lowest data weights are shown in figure 2 and the training examples with the highest weights in figure 3, both on the small dataset. The training data points with low weights seem harder to classify (for example, (b) and (c) in figure 2 could be interpreted as nines).

### 5.6. *Discussion*

With least squares auto-tuning, the test error was cut in half for both the small and full dataset. This experiment demonstrates that even though there are thousands of hyper-parameters, the method does not overfit. This example is highly simplified in order to illustrate the usage of least squares auto-tuning; better models can be devised using the very same techniques illustrated.

## 6. Conclusion

This article presented a general framework for tuning hyper-parameters in least squares problems. The algorithm proposed for least squares auto-tuning is simple and scales well. It was shown how this framework can be used for data fitting. Finally, least squares auto-tuning was applied to variations of a digit classification dataset, cutting the test error of standard least squares in half. In a follow-up article, the authors have applied least squares auto-tuning to Kalman smoothing (Barratt and Boyd 2020).

## Disclosure statement

The authors declare that they have no conflicts of interest.


## Funding

This work was supoprted by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1656518.


## References


Abadi, Martin, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. "Tensorflow: a system for large-scale machine learning." In *Proc. Operating Systems Design and Implementation*, Vol. 16, 265–283.

Agrawal, A., S. Barratt, S. Boyd, E. Busseti, and W. Moursi. 2019a. "Differentiating through a cone program." *Journal of Applied and Numerical Optimization* 1 (2): 107–115.

Agrawal, A., A. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, A. Ganichev, et al. 2019b. "TensorFlow Eager: a Multi-Stage, Python-Embedded DSL for Machine Learning." *Proceedings SysML Conference* .

Agrawal, Akshay, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and Zico Kolter. 2019c. "Differentiable convex optimization layers." In *Proc. Advances in Neural Information Processing Systems*, 9558–9570.

Agrawal, Akshay, Shane Barratt, Stephen Boyd, and Bartolomeo Stellato. 2019d. "Learning Convex Optimization Control Policies." In *Conf. Learning for Decision and Control 2020*, To appear.

Amos, B. 2017. "A fast and differentiable QP solver for pytorch." `https://github.com/locuslab/qpth`.

Amos, B., I. Jimenez, J. Sacks, B. Boots, and Z. Kolter. 2018. "Differentiable MPC for End-to-end Planning and Control." In *Proc. Advances in Neural Information Processing Systems*, 8299–8310.

Amos, B., and Z. Kolter. 2017. "Optnet: Differentiable optimization as a layer in neural networks." In *Proc. Intl. Conf. on Machine Learning*, 136–145.

Amos, Brandon, and Denis Yarats. 2019. "The Differentiable Cross-Entropy Method." *arXiv preprint arXiv:1909.12830* .

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

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

Barratt, Shane, Guillermo Angeris, and Stephen Boyd. 2020. "Automatic Repair of Convex Optimization Problems." *arXiv preprint arXiv:2001.11010* .

Barratt, Shane, and Stephen Boyd. 2020. "Fitting a Kalman smoother to data." In *American Control Conf. 2020*, To appear.

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

Baydin, A., and B. Pearlmutter. 2014. "Automatic differentiation of algorithms for machine learning." In *ICML 2014 AutoML Workshop*, .

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

Beck, A. 2017. *First-order methods in optimization*. SIAM.

Belanger, D., and A. McCallum. 2016. "Structured prediction energy networks." In *Proc. Intl. Conf. on Machine Learning*, 983–992.

Belanger, D., B. Yang, and A. McCallum. 2017. "End-to-end learning for structured prediction energy networks." In *Proc. Intl. Conf. on Machine Learning*, 429–439.

Bengio, Y. 2000. "Gradient-based optimization of hyperparameters." *Neural Computation* 1889–1900.

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

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

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

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

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

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

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

Chen, G., M. Gan, C. Chen, and H. Li. 2018. "A regularized variable projection algorithm for separable nonlinear least-squares problems." *IEEE Transactions on Automatic Control* 64 (2): 526–537.

Ciregan, D., U. Meier, and J. Schmidhuber. 2012. "Multi-column deep neural networks for image classification." In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, June, 3642–3649.

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

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

Dongarra, J., J. Cruz, S. Hammarling, and I. Duff. 1990. "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.

Dontchev, A., and T. Rockafellar. 2009. *Implicit functions and solution mappings*. Springer.

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

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

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

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

Fu, J., H. Luo, J. Feng, and T. Chua. 2016. "Distilling Reverse-Mode Automatic Differentiation (DrMAD) for Optimizing Hyperparameters of Deep Neural Networks." *arXiv preprint arXiv:1601.00917* .

Gauss, C. 1809. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. Vol. 7. Perthes et Besser.

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

Golub, G., and V. Pereyra. 1973. "The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate." *SIAM Journal on Numerical Analysis* 10 (2): 413–432.

Golub, G., and V. Pereyra. 2003. "Separable nonlinear least squares: the variable projection method and its applications." *Inverse problems* 19 (2): R1.

Golub, G., and C. Van Loan. 2012. *Matrix computations*. JHU Press.

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

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

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media.

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

Higham, N. 2002. *Accuracy and stability of numerical algorithms*. Vol. 80. Siam.

Hurter, Ferdinand, and Vero Driffield. 1890. "Photochemical investigations and a new method of determination of the sensitiveness of photographic plates." *Journal of the Society of the Chemical Industry* 9: 455–469.

Innes, M. 2018. "Don't Unroll Adjoint: Differentiating SSA-Form Programs." In *Workshop on Systems for ML at NeurIPS 2018*, .

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

Lall, S., and S. Boyd. 2017. "Lecture 11 Notes for EE104." `http://ee104.stanford.edu/lectures/prox_gradient.pdf`.

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

Lawson, C. 1961. "Contribution to the theory of linear least maximum approximation." *Ph.D. dissertation* .

Lawson, C., and R. Hanson. 1995. *Solving least squares problems*. Vol. 15. SIAM.

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

Legendre, A. 1805. *Nouvelles méthodes pour la détermination des orbites des comètes*.

Ling, C., F. Fang, and Z. Kolter. 2018. "What game are we playing? End-to-end learning in normal and extensive form games." In *Intl. Joint Conf. on Artificial Intelligence*, .

Ling, C., F. Fang, and Z. Kolter. 2019. "Large Scale Learning of Agent Rationality in Two-Player Zero-Sum Games." .

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

Lorraine, J., and D. Duvenaud. 2018. "Stochastic Hyperparameter Optimization through Hypernetworks." *arXiv preprint arXiv:1802.09419* .

Maclaurin, D., D. Duvenaud, and R. Adams. 2015a. "Autograd: Effortless gradients in NumPy." In *Proc. ICML 2015 AutoML Workshop*, .

Maclaurin, D., D. Duvenaud, and R. Adams. 2015b. "Gradient-based hyperparameter optimization through reversible learning." In *Proc. Intl. Conf. on Machine Learning*, 2113–2122.

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

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

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

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

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

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

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

Parikh, N., and S. Boyd. 2014. "Proximal algorithms." *Foundations and Trends® in Optimization* 1 (3): 127–239.
Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "PyTorch: An imperative style, high-performance deep learning library." In *Proc. Advances in Neural Information Processing Systems*, 8024–8035.
Prechelt, L. 1998. "Early stopping-but when?" In *Neural Networks: Tricks of the Trade*, 55–69. Springer.
Rasmussen, C. 2004. *Gaussian Processes in Machine Learning*. Springer.
Ren, M., W. Zeng, B. Yang, and R. Urtasun. 2018. "Learning to reweight examples for robust deep learning." *arXiv preprint arXiv:1803.09050* .
Rice, J., and K. Usow. 1968. "The Lawson algorithm and extensions." *Mathematics of Computation* 22 (101): 118–127.
Shor, N. 1985. *Minimization methods for non-differentiable functions*. Vol. 3. Springer Science & Business Media.
Smith, S. 1995. "Differentiation of the Cholesky algorithm." *Journal of Computational and Graphical Statistics* 4 (2): 134–147.
Snoek, J., H. Larochelle, and R. Adams. 2012. "Practical Bayesian optimization of machine learning algorithms." In *Proc. Advances in Neural Information Processing Systems*, 2951–2959.
Speelpenning, B. 1980. *Compiling fast partial derivatives of functions given by algorithms*. Technical Report.
van Merriënboer, B., D. Moldovan, and A. Wiltschko. 2018. "Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming." In *Proc. Advances in Neural Information Processing Systems*, 6259–6268.
Wengert, Robert Edwin. 1964. "A simple automatic derivative evaluation program." *Communications of the ACM* 7 (8): 463–464.

## Appendix 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

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

The goal is to find the linear operator $D_A\phi(A, B) : \mathbf{R}^{k\times n} \to \mathbf{R}^{n\times m}$, *i.e.*, the derivative of $\phi$ 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 $\phi$ with respect to $B$.

### A.1. *Derivative with respect to A*

Here

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

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 the approximation $(X + Y)^{-1} \approx X^{-1} - X^{-1}YX^{-1}$ for $Y$ small was used, and higher order terms were dropped. 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 $D_A f(A,B)$ is given by

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

from which it can be concluded that $\nabla_A f = (B - A\theta)C^T - AC\theta^T$.

### A.2. *Derivative with respect to B*

Here $D_B\phi(A,B)(\Delta B) = (A^TA)^{-1}A^T\Delta B$, 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 $D_B f(A,B)$ is given by

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

from which it can be concluded that $\nabla_B f = AC$.

## Appendix 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 $r$. It follows from elementary subdifferential calculus 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 $\omega^{k+1}$ is

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

which leads to the stopping criterion (7).