Mathematical Programming Computation
https://doi.org/10.1007/s12532-024-00258-8

FULL LENGTH PAPER

Efficient algorithms for implementing incremental proximal-point methods

Alex Shtoff[1]

Received: 20 September 2022 / Accepted: 6 June 2024

Abstract
Model training algorithms which observe a small portion of the training set in each computational step are ubiquitous in practical machine learning, and include both stochastic and online optimization methods. In the vast majority of cases, such algorithms typically observe the training samples via the gradients of the cost functions the samples incur. Thus, these methods exploit are the *slope* of the cost functions via their first-order approximations. To address limitations of gradient-based methods, such as sensitivity to step-size choice in the stochastic setting, or inability to exploit small function variability in the online setting, several streams of research attempt to exploit more information about the cost functions than just their gradients via the well-known proximal operators. However, implementing such methods in practice poses a challenge, since each iteration step boils down to computing the proximal operator, which may not be as easy as computing a gradient. In this work we devise a novel algorithmic framework, which exploits convex duality theory to achieve both *algorithmic efficiency* and *software modularity* of proximal operator implementations, in order to make experimentation with incremental proximal optimization algorithms accessible to a larger audience of researchers and practitioners, by reducing the gap between their theoretical description in research papers and their use in practice. We provide a reference Python implementation for the framework developed in this paper as an open source library at on GitHub (https://github.com/alexshtf/inc_prox_pt/releases/tag/prox_pt_paper) Shtoff (Efficient implementation of incremental proximal point methods arXiv:2205.01457, 2024), along with examples which demonstrate our implementation on a variety of problems, and reproduce the numerical experiments in this paper. The pure Python reference implementation is not necessarily the most efficient, but is a basis for creating efficient implementations by combining Python with a native backend.

Keywords Convex optimization · Duality · Proximal operator · Proximal point

✉ Alex Shtoff
alex.shtoff@yahooinc.com

[1] Yahoo Research, Haifa, Israel

Published online: 01 August 2024

1 Introduction

Incremental optimization is the 'bread and butter' of the theory and practice of modern machine learning, where we aim to minimize a *cost* function of the model's parameters, and can be roughly classified into two major regimes. In the stochastic optimization regime it is assumed that the cost functions are sampled from a stationary distribution, our objective is to design algorithms which minimize the expected cost, the theoretical analysis typically produces bounds on the expected cost, and typical algorithms are variants of the stochastic gradient method [31]. In the online regime cost functions from a pre-defined family are chosen by an adversary, our objective is minimizing the cumulative cost incurred by the sequence of the cost functions we observe, the theoretical analysis tool is the performance relatively to a theoretical optimum called *regret*, and typical algorithms are variants of the online gradient method [38], or follow the leader [21] based methods operating on linearly approximated costs.

Both in the stochastic and online regime, these methods follow the following incremental protocol: (a) observe the cost function incurred by a small subset of training samples; (b) update the model's parameters. Since the focus of this paper is on the *implementation* of the computational steps of each iteration, the exact regime plays no role, and thus we treat both regimes under the same umbrella, and refer to these methods as *incremental* methods, e.g. incremental gradient descent.

To be concrete, the incremental gradient method having observed the cost function f, which is usually interpreted as "make a small step in the descent direction", reads:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla f(\mathbf{x}_t),$$

but can equivalently written using the celebrated *proximal view*:

$$\mathbf{x}_{t+1} = \operatorname*{argmin}_{\mathbf{x}} \left\{ f(\mathbf{x}_t) + \langle \nabla f(\mathbf{x}_t), \mathbf{x} - \mathbf{x}_t \rangle + \frac{1}{2\eta} \|\mathbf{x} - \mathbf{x}_t\|_2^2 \right\},$$

meaning "minimize a linear approximation of f, but stay close to $\mathbf{x}_t$". A different approach is using the cost directly instead of its linear approximation via the well-known *proximal operator* [27, 28]:

$$\mathbf{x}_{t+1} = \operatorname{prox}_{\eta f}(\mathbf{x}_t) := \operatorname*{argmin}_{\mathbf{x}} \left\{ f(\mathbf{x}) + \frac{1}{2\eta} \|\mathbf{x} - \mathbf{x}_t\|_2^2 \right\}, \tag{PROX}$$

The idea is not new, and dates back to the first proximal iteration algorithm of Martinet [26], which was designed as a theoretical method for the non-incremental regime. In the stochastic optimization regime this idea is known as stochastic proximal point, and in the online regime as implicit online learning. Of course, we do not have to use the cost itself, and f in (PROX) may be any approximation which is not necessarily linear. In line with our aim to use the same terminology for stochastic and online algorithms, we call the idea above *incremental proximal iteration*.

In the online regime, incremental proximal iteration was first proposed by Kivinen and Warmuth [23], and later analyzed by Kulis and Bartlett [24] where it was termed

'implicit online learning'. Follow-up, such as Karampatziakis and Langford [22] and Campolongo and Orabona [11], showed improved regret guarantees and robustness in various settings.

In the stochastic regime, incremental proximal iteration was first proposed and analyzed for the finite-sum optimization problem by Bertsekas [9]. However, significant advantages over first order methods were discovered later in the form of improved stability with respect to step-size choices [2, 5, 33]. Consequently, hyper-parameter tuning becomes significantly cheaper when using certain proximal point based methods to train models, and consequently overall the computational cost and energetic footprint of the training process is reduced, even if training for one specific hyper-parameter configuration is more expensive.

Using a finer model instead of a linear approximation bears a cost—since f can be arbitrarily complex, computing the proximal operator may be arbitrarily hard, or even infeasible to do in practice. Thus, there is a trade-off between any advantage a finer approximation may provide, and the difficulty of proximal operator computation.

In contrast to the above-mentioned works, rather than theoretical analysis of novel high-level techniques, our aim is to substantially reduce the difficulty of computing the proximal operator devising efficient algorithms to do so in a variety of setups, and providing a Python reference implementing those algorithms. Our pure Python implementation is not necessarily the most efficient, since its main focus is demonstration and readability, and in terms of speed we aim to be modest: up to a moderate constant factor (10–20 times) slower than a competing gradient method. For some families of functions f, computing $\text{prox}_{\eta f}$ may even be as cheap as a regular gradient step, but others require solving a one-dimensional or a low-dimensional optimization problem, and a naive Python implementation is far from optimal. Our Python implementation is designed to demonstrate that efficient algorithms for computing the proximal step need not be hard to implement, and encourage our readers to adapt or revise our implementation to their needs. It is our aim, and hope, that the above contributions make it easier to apply these methods in practice for researchers, while also motivate more research by making numerical experimentation with incremental proximal point methods easily accessible to the research community.

We'd like to emphasize that the aim of this paper is devising efficient algorithms for *implementing* proximal operators of useful functions in machine learning, rather than *using* proximal operators to derive better converging algorithms, in order to make such implementations accessible to the research community. In addition to efficiency, we use extensive mathematical theory to derive *modular* implementations, so that researchers may compose the functions they desire from atomic building blocks. For example, an L1 regularized logistic regression problem's cost function is composed of a linear function, composed onto the convex logistic function $t \to \ln(1 + \exp(t))$, and with the L1 regularizer. Thus, the code for training such models may look like this:

```
x = torch.zeros(x_star.shape)
# note the modular composition below
optimizer = IncRegularizedConvexOnLinear(x, Logistic(), L1Reg(0.01))
epoch_loss = 0.
for t, (a, b) in enumerate(my_data_set, start=1):
    step_size = 1. / math.sqrt(t)
    epoch_loss += optimizer.step(step_size, a, b)
```

Clearly, a proximal step is more computationally demanding than a gradient step, thus our implementation is slower. We show, empirically, that for a variety of problems, our implementation is a small constant factor slower than (proximal) gradient steps, which is useful for both researchers and practitioners. From a practical perspective, since the works we cited above demonstrate, both theoretically and empirically, that proximal point methods methods are very robust to the step-size choice, hyper-parameter tuning becomes significantly cheaper, and thus such a library of implementations may be useful in practice for *reducing* the computational resources required for training a model. For researchers, an implementation whose run-time is a small constant factor that of gradient descent is useful for conducting numerical experiments for a reasonable computational cost for their new algorithms based on proximal operators. The results in this paper were obtained on a 2019 MacBook Pro with a 2.4 GHz 8-Core Intel Core i9 processor, and 32 GB of RAM. The code for reproducing the results is in our code repository.

Since this paper is also aimed at an audience partially unfamiliar with convex analysis theory, but coming from a more machine-learning oriented background. Hence, we will briefly introduce the concepts we need throughout the paper, and refer to additional literature for an in-depth treatment. Consequently, some derivations and proofs in this paper may appear trivial to readers well versed in convex analysis and optimization.

1.1 Notation

Scalars are denoted by lowercase Latin or Greek letters, e.g., a, α. Vectors are denoted by lowercase boldface letters, e.g. $\mathbf{a}$, and matrices by uppercase boldface letters, e.g. $\mathbf{A}$. Vector or matrix components are denoted like scalars, e.g. v_i, A_{ij}.

1.2 Extended real-valued functions

Optimization problems are occasionally described using extended real-valued functions, which are functions that can take any real value, in addition to the infinite values $-\infty$ and ∞. An extended real-value function $\phi : \mathbb{R}^d \to [-\infty, \infty]$ has an associated effective domain $\mathrm{dom}(\phi) = \{\mathbf{x} \in \mathbb{R}^d : \phi(\mathbf{x}) < \infty\}$, and it's natural to use such functions to encode constrained optimization problems, where $\mathrm{dom}(\phi)$ or $\mathrm{dom}(-\phi)$ are used to encode constraints of minimization or maximization problems. Throughout this paper, we implicitly assume that any extended real-valued function ϕ is:

- *proper*—its $-\infty$ nowhere, and $\mathrm{dom}(\phi) \neq \emptyset$, and
- *closed*—it's epigraph $\mathrm{epi}(\phi) = \left\{(\mathbf{x}, t) \in \mathbb{R}^d \times \mathbb{R} : \phi(\mathbf{x}) \leq t\right\}$ is a closed set.

When ϕ is used in the context of a maximization problem, we make those assumptions about $-\phi$. Any reasonable function of practical interest is proper, and the vast majority of functions useful in practice are closed as well. An extensive introduction to extended real-valued function can be found, for example, in [7, Chapter 2].

1.3 Our contributions

In this work, we consider the following families of functions f when computing $\text{prox}_{\eta f}$. For each family of functions we show examples of machine learning models for whose training it might be useful, devise an algorithm for computing its proximal operator, provide a Python reference implementation, and experimentally measure its efficiency. The families are described below.

A convex onto linear composition:

$$f(\mathbf{x}) = h(\mathbf{a}^T\mathbf{x} + b), \tag{CL}$$

where $h : \mathbb{R} \to (-\infty, \infty]$ is a convex extended real-valued function.
A regularized convex onto linear composition:

$$f(\mathbf{x}) = h(\mathbf{a}^T\mathbf{x} + b) + r(\mathbf{x}), \tag{RCL}$$

where $h : \mathbb{R} \to (-\infty, \infty]$, and $r : \mathbb{R}^d \to (-\infty, \infty]$ are convex extended real-valued functions.
A mini-batch of convex onto linear compositions:

$$f(\mathbf{x}) = \frac{1}{m}\sum_{i=1}^{m} h(\mathbf{a}_i^T\mathbf{x} + b_i), \tag{CL-B}$$

where $h : \mathbb{R} \to (-\infty, \infty]$ is a convex extended real-valued function, and m is small.

For the above families we develop a framework for computing the proximal operator, based on convex duality, to achieve both algorithmic efficiency and software modularity. We provide a pure Python reference implementation that aims to be moderately efficient. For some problems it may be on par with SGD, whereas for others it may be up to a few dozen times slower. Indeed, a pure Python reference implementation is is not necessarily the most efficient one, and a better alternative in terms of efficiency is a combination of C and Python, where C is used for loop-intensive tasks, such as one-dimensional root finding. We chose PyTorch as our array module, due to easier integration with auto-grad based model training code. The code we build in this paper is available on GitHub at https://github.com/alexshtf/inc_prox_pt/ [34].

The remainder of the paper is organized as follows. After discussing previous work below, we develop the initial version for our algorithmic framework for convex onto linear compositions in Sect. 2. In that section, we also provide full code inline to let the readers appreciate how software modularity is facilitated by our framework. Then, we

proceed to extending our framework to regularized convex onto linear compositions in Sect. 3, and to a mini-batch of convex onto linear compositions in Sect. 4. These sections contain only the framework code, whereas the remaining code stemming from these sections is available both on our GitHub repository, and in Appendices A and B in this paper.

1.4 Previous work

There has been a significant body of research into the analysis of incremental proximal iteration algorithms in various settings and approximating models, which in addition to the examples we provided above also include [3, 4, 15–17, 20, 25, 35, 37]. However, to the best of our knowledge, finding generic algorithms for efficient implementation of incremental proximal algorithms received little attention, and the issue has been partially addressed in the papers focused on the analysis.

Exceptions to the above rule are methods based on the proximal gradient approach, original proposed by Passty [30], where the approximating function is of the form:

$$f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + r(\mathbf{x}),$$

where $r(x)$ is some 'simple' function, usually a regularizer, for which a closed form solution for the proximal operator of r is known. See, for example, the works of Parikh and Boyd [29] and Beck [7] and references therein for examples. In this work we consider a significantly broader family of functions, aimed at machine learning applications, by building on existing theory in convex analysis, and functions whose proximal operator is known.

Proximal operators of functions belonging to the (CL) family is covered to some extent in the literature. For example, Kulis and Bartlett [24] shows an explicit formula for the case when the 'outer' function h is the ℓ_2 cost $h(t) = \frac{1}{2}t^2$. The case of $h(t) = \max(t, 0)$ has an explicit formula in the work of Asi and Duchi [5], while $h(t) = |t|$ is treated in [15]. Additional formulas can be found in [25]. In [17] the authors provide an efficient Cython implementation for the Logistic and Hinge loss, but not in a generic framework. While Ryu and Boyd [33] show an explicit method for the entire family, they do not show how the method is derived, and provide neither concrete examples, nor code. We treat this family using a uniform framework based on duality, show its benefits from a software engineering perspective of decoupling concerns, and provide explicit code for examples which are useful in machine learning.

The work of Ryu and Boyd [33] briefly discusses an algorithm for a simplified version (RCL) case where the regularizer is assumed to be separable. Their idea is similar to ours, but we devise an algorithm for a more general case, and provide explicit examples and code which is useful for machine learning practitioners. Moreover, they provide a 'switch to SGD heuristic' for improving computational efficiency by switching to a regular stochastic gradient method. This heuristic is orthogonal to the contributions of our paper.

Finally, the (CL-B) family is tackled in [2] for the special case of $h(t) = \max(0, t)$. We devise a uniform framework for a broader family which covers a wide variety of functions h, and give several examples which may be useful to practitioners.

2 A convex onto linear composition

In this section we first present some applications of the convex onto linear family described in (CL), and then describe a generic framework for designing and implementing algorithms for computing the proximal point of this function family.

2.1 Applications

Compositions of convex onto linear functions appear in a wide variety of classical machine learning problems, but also appear to be useful for other applications as well. This family is useful when training on *one* training sample in each iteration.

Linear regression In the simplest case, least-squares regression, we aim to minimize a sum, or an expectation, of costs of the form

$$f(\mathbf{x}) = \frac{1}{2}(\mathbf{a}^T\mathbf{x} + b)^2,$$

where $(\mathbf{a}, b) \in \mathbb{R}^n \times \mathbb{R}$ are the data of a training sample, and $\mathbf{x}$ is the model parameters vector. In this case we have $h(t) = \frac{1}{2}t^2$. Different regression variants are obtained by choosing a different function h, for example, using $h(t) = |t|$ we obtain robust regression:

$$f(\mathbf{x}) = |\mathbf{a}^T\mathbf{x} + b|,$$

and using the $h(t) = \max((p-1)t, pt)$ for some $p \in (0, 1)$, we obtain linear *quantile regression*.

Logistic regression For the binary classification problem, we are given an input features $\mathbf{a} \in \mathbb{R}^n$ and a label $y \in \{-1, 1\}$. The probability of $y = 1$ is modeled using

$$\sigma(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{a}^T\mathbf{x})},$$

where $\mathbf{x}$ is the model's parameter vector. The cost incurred by each training sample is computed using the *binary cross-entropy loss*:

$$f(\mathbf{x}) = \begin{cases} -\ln(\sigma(\mathbf{x})) & y = -1 \\ -\ln(1 - \sigma(\mathbf{x})) & y = 1, \end{cases}$$

which after some algebraic manipulation can be written as:

$$f(\mathbf{x}) = \ln(1 + \exp(-y\mathbf{a}^T\mathbf{x})).$$

Defining $h(t) = \ln(1 + \exp(t))$, we obtain the convex-onto-linear form.
The APROX model [5] A cost function ϕ which is bounded below, where w.l.o.g we assume that it is bounded below by zero, is approximated by:

$$f(\mathbf{x}) = \max(\phi(\mathbf{x}_t) + \langle \nabla\phi(\mathbf{x}_t), \mathbf{x} - \mathbf{x}_t \rangle, 0).$$

It's similar to a linear approximation, but it also incorporates knowledge about the function's lower bound: if a cost function is non-negative, its approximation should also be. Taking $h(t) = \max(t, 0)$, $\mathbf{a} = \nabla\phi(\mathbf{x}_t)$, and $b = \phi(\mathbf{x}_t) - \langle \phi(\mathbf{x}_t), \mathbf{x}_t \rangle$, we obtain the convex-onto-linear form (CL).
The prox-linear approximation Assume we are given a cost function of the form

$$\phi(\mathbf{x}) = h(g(\mathbf{x})),$$

where the *outer* function h is convex, and the *inner* function g is an arbirary function, such as a deep neural network. For instance, suppose we're solving a classification problem using a neural network u, whose output is fed to the sigmoid, and then to the binary cross-entropy loss. Similarly to the logistic regression setup above, given a label $y \in \{-1, 1\}$, and an input $\mathbf{w}$, we obtain:

$$\phi(\mathbf{x}) = \ln(1 + \exp(-yu(\mathbf{x}, \mathbf{w})).$$

Taking $h(t) = \ln(1 + \exp(t))$ and $g(\mathbf{x}) = -yu(\mathbf{x}, \mathbf{w})$ we have the desired form.

The idea of the prox-linear method is to linearly approximate the inner function around the current iterate, while leaving the outer function as is. Thus, we obtain the following approximation of the cost ϕ

$$f(\mathbf{x}) = h(g(\mathbf{x}_t) + \langle \nabla g(\mathbf{x}_t), \mathbf{x} - \mathbf{x}_t \rangle),$$

and taking $\mathbf{a} = \nabla g(\mathbf{x})$, and $b = g(\mathbf{x}_t) - \langle \nabla g(\mathbf{x}_t), \mathbf{x}_t \rangle$ we obtain the desired convex-onto-linear form (CL). See the recent work of Drusvyatskiy and Paquette [19] and references therein for extensive analysis and origins of the method.

2.2 The proximal operator

Having shown a variety of potential applications, let's explore the proximal operator of convex-onto-linear functions. Its computation amounts to solving the following problem:

$$\min_{\mathbf{x}} \quad h(\mathbf{a}^T\mathbf{x} + b) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2. \tag{1}$$

We'll tackle this problem, and most of the remaining cost families, using the well known *convex duality* framework, which we briefly introduce here for completeness. An extensive introduction can be found in many optimization textbooks, such as [7].

2.2.1 Convex duality

We make a brief introduction to a subset of convex duality theory for unfamiliar readers who would like to understand how we built our framework and how to extend it for their own purposes. Since we don't need most generic convex duality theory this paper, we introduce a simplification. Suppose we're given an optimization problem:

$$\min_{\mathbf{x}} \quad f(\mathbf{x}) \quad \text{s.t.} \quad \mathbf{A}\mathbf{x} = \mathbf{b}, \tag{Q}$$

where $\mathbf{A} \in \mathbb{R}^m \times \mathbb{R}^n$ is a matrix, and $f : \mathbb{R}^n \to (-\infty, \infty]$ is a closed and convex extended real-valued function. Define

$$q(\mathbf{s}) = \inf_{\mathbf{x}} \left\{ \mathcal{L}(\mathbf{x}, \mathbf{s}) : = f(\mathbf{x}) + \mathbf{s}^T (\mathbf{A}\mathbf{x} - \mathbf{b}) \right\},$$

namely, we replace the j^{th} linear constraint by a "price" s_j for its violation, and define q to be the optimal value as a function of these prices. The modified cost function $\mathcal{L}$ is called the *Lagrangian* associated with the constrained problem (Q).

First, it's apparent that $q(\mathbf{s})$ is *concave*, since it is a minimum of linear functions of $\mathbf{s}$. Moreover, it's easy to see that $q(\mathbf{s})$ is a lower bound for the optimal value of (Q), using the simple observation that minimizing over a subset of $\mathbb{R}^n$ produces a value that is higher or equal to the minimization over the entire space:

$$\begin{aligned} q(\mathbf{s}) &= \inf_{\mathbf{x}} \left\{ f(\mathbf{x}) + \mathbf{s}^T (\mathbf{A}\mathbf{x} - \mathbf{b}) \right\} \\ &\leq \inf_{\mathbf{x}} \left\{ f(\mathbf{x}) + \mathbf{s}^T (\mathbf{A}\mathbf{x} - \mathbf{b}) : \mathbf{A}\mathbf{x} = \mathbf{b} \right\} \\ &= \inf_{\mathbf{x}} \{ f(\mathbf{x}) : \mathbf{A}\mathbf{x} = \mathbf{b} \} \end{aligned}$$

The problem of finding the "best" lower bound is called the *dual problem* associated with (Q), namely:

$$\max_{\mathbf{s}} \quad q(\mathbf{s}) \quad \text{s.t.} \quad q(\mathbf{s}) > -\infty, \tag{D}$$

whereas the original problem (Q) is called the *primal* problem. A well known result in convex analysis is that with slight technical conditions, the optimal values of the primal and the dual problems coincide:

Theorem 1 (Strong duality) *Suppose that f is a convex closed extended real-valued function, that the optimal value of* (Q) *is finite, namely,*

$$f_{\text{opt}} = \inf_{\mathbf{x}} \{ f(\mathbf{x}) : \mathbf{A}\mathbf{x} = \mathbf{b} \} > -\infty,$$

and that there exists some feasible solution $\hat{\mathbf{x}}$. Then,

(i) *The optimal value of the dual problem* (D) *is attained at some optimal solution $\mathbf{s}^*$, and it is equal f_{opt}.*

(ii) *The optimal solutions of* (Q) *are* $\mathbf{x}^* \in \operatorname{argmin}_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \mathbf{s}^*)$ *which are feasible. In particular, if* $\mathcal{L}(\mathbf{x}, \mathbf{s}^*)$ *has a unique minimizer, then it must be an optimal solution of* (Q).

Proof (i) is a special case of Theorem A.1 and (ii) is a special case of Theorem (ii) in [7]. □

The strong duality theorem has an important consequence for the case when the dual problem (D) significantly easier to solve than the primal problem (Q). Having obtained its optimal solution $\mathbf{s}^*$, we can recover the optimal solution of the primal problem by minimizing the Lagrangian function $\mathcal{L}(\mathbf{x}, \mathbf{s}^*)$ over $\mathbf{x}$.

2.2.2 Employing duality

Duality requires a constrained optimization problem, whereas the proximal operator in Eq. (1) aims to solve an unconstrained problem. However, constraints are easily added with the help of an auxiliary variable, and we can equivalently solve:

$$\min_{\mathbf{x},z} \quad h(z) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 \quad \text{s.t.} \quad z = \mathbf{a}^T\mathbf{x} + b$$

The dual objective is therefore:

$$\begin{aligned} q(s) &= \inf_{\mathbf{x},z} \left\{ h(z) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 + s(\mathbf{a}^T\mathbf{x} + b - z) \right\} \\ &= \underbrace{\min_{\mathbf{x}} \left\{ \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 + s\mathbf{a}^T\mathbf{x} \right\}}_{A} + \underbrace{\inf_{z}\{h(z) - sz\}}_{B} + sb \end{aligned}$$

The term denoted by A is a strictly convex quadratic function, whose minimizer is

$$\mathbf{x}^* = \mathbf{x}_t - \eta s\mathbf{a}, \tag{2}$$

and the minimum itself is

$$A = -\frac{\eta\|\mathbf{a}\|_2^2}{2}s^2 + (\mathbf{a}^T\mathbf{x}_t)s.$$

The term denoted by B can be alternatively written as

$$B = -\sup_{z}\{sz - h(z)\} = -h^*(s),$$

where h^* is a well-known object in optimization called the *convex conjugate* of h. A catalogue of pairs of convex conjugate functions is available in a variety of standard textbooks on optimization, e.g. [7]. For completeness, in Table 1 we show conjugate pairs which are useful for the machine-learning oriented examples in this paper.

Table 1 Example convex conjugate pairs

$h(z)$	$\text{dom}(h)$	$h^*(s)$	$\text{dom}(h^*)$	Useful for
$\frac{1}{2}z^2$	$\mathbb{R}$	$\frac{1}{2}s^2$	$\mathbb{R}$	Linear least squares
$\ln(1+\exp(z))$	$\mathbb{R}$	$s\ln(s)+(1-s)\ln(1-s)$ where $0\ln(0)\equiv 0$	$[0,1]$	Logistic regression
$\max(z,0)$	$\mathbb{R}$	$\begin{cases}0 & s\in[0,1]\\ \infty & \text{else}\end{cases}$	$[0,1]$	AProx model, Hinge loss

Convex conjugates possess two important properties. First, under mild technical conditions, we have $(h^*)^* = h$, i.e. the bi-conjugate of a convex function is the function itself. These conditions hold for most functions we care about in practice, including the functions in Table 1. Second, the conjugate is always convex.

Summarizing the above, the dual problem aims to solve the following *one dimensional* and *strongly concave* maximization problem:

$$\max_s \quad q(s) \equiv -\underbrace{\frac{\eta\|\mathbf{a}\|_2^2}{2}}_{\frac{\alpha}{2}} s^2 + \underbrace{(\mathbf{a}^T\mathbf{x}_t + b)}_{\beta} s - h^*(s). \tag{3}$$

Strong concavity implies the existence of a unique maximizer s^*, while the strong duality theorem implies we can recover the proximal operator we seek by substituting the above maximizer into Eq. (2).

We implement the above idea the `IncConvexOnLinear` class below using the `PyTorch` library.

```
import torch

class IncConvexOnLinear:
    def __init__(self, x, h):
        self._h = h
        self._x = x

    def step(self, eta, a, b):
        """
        Performs the optimizer's step, and returns the loss incurred.
        """
        h = self._h
        x = self._x

        # compute the dual problem's coefficients
        alpha = eta * torch.sum(a**2)
        beta = torch.dot(a, x) + b

        # solve the dual problem
        s_star = h.solve_dual(alpha.item(), beta.item())

        # update x
        x.sub_(eta * s_star * a)

        return h.eval(beta.item())
```

```
    def x(self):
        return self._x
```

Note, that from a software engineering perspective, we encode the function h using an object which has two operations: compute the value of h, and solve the dual problem. In the next sub-sections we will implement three such objects: `HalfSquared` for $h(z) = \frac{1}{2}z^2$, `Logistic` for $h(z) = \ln(1+\exp(z))$, and `Hinge` for $h(z) = \max(z, 0)$.

As an example, applying the implicit online learning idea of Kulis and Bartlett [24] to the linear least squares problem looks like this:

```
x = torch.zeros(d)
optimizer = IncConvexOnLinear(x, HalfSquared())
for t, (a, b) in enumerate(my_data_set):
    eta = get_step_size(t)
    optimizer.step(eta, a, b)

print('The parameters are: ' + str(x))
```

2.2.3 The half-squared function

For $h(z) = \frac{1}{2}z^2$, according to Table 1, we have $h^*(s) = \frac{1}{2}s^2$, and thus the dual problem in Eq. (3) amounts to maximizing

$$q(s) = -\frac{\alpha}{2}s^2 + \beta s - \frac{1}{2}s^2 = -\frac{1+\alpha}{2}s^2 + \beta s.$$

Hence, in this case $q(s)$ is a simple concave parabola, maximized at

$$s^* = \frac{\beta}{1+\alpha}.$$

Consequently, our `HalfSquared` class is:

```
import torch
import math

class HalfSquared:
    def solve_dual(self, alpha, beta):
        return beta / ( 1 + alpha)

    def eval(self, z):
        return 0.5 * (z ** 2)
```

2.2.4 The logistic function

For $h(z) = \ln(1+\exp(z))$, according to Table 1, the dual problem in Eq. (3) amounts to maximizing

$$q(s) = -\frac{\alpha}{2}s^2 + \beta s - s\ln(s) - (1-s)\ln(1-s). \tag{4}$$

The following simple result paves the way towards maximizing q.

Proposition 1 *The function q defined in Eq. (4) has a unique maximizer inside the open interval $(0, 1)$.*

Proof Note that $\mathrm{dom}(q) = [0, 1]$, thus maximizing q is done over a compact interval. Its continuity with the Weirstrass theorem ensures that it has a maximizer in $[0, 1]$, and its strict concavity ensures that the maximizer is unique. The derivative $q'(s) = -\alpha s + \beta - \ln(s) + \ln(1 + s)$ is continuous, decreasing, and satisfies:

$$\lim_{s \to 0} q'(s) = \infty, \quad \lim_{s \to 1} q'(s) = -\infty.$$

Hence, there must be a unique point in the open interval $(0, 1)$ where $q'(s) = 0$. Since q is concave, that point must be the maximizer. □

Proposition 1 implies that we can maximize q by employing any root finding algorithm to find a zero of its derivative. Its initial interval $[l, u]$ can be found by:

- $l = 2^{-k}$ for the smallest positive integer k such that $q'(2^{-k}) > 0$.
- $u = 1 - 2^{-k}$ for the smallest positive integer k such that $q'(1 - 2^{-k}) < 0$.

For our reference implementation, we chose to rely on Brent's method [10, Chapter 5] readily available in the `scipy` package. The method finds a ε-approximate root in $O(\ln(\frac{1}{\varepsilon}))$ iterations in the worst case, but is typically significantly faster than naive bisection. The resulting `Logistic` class is listed below:

```
from scipy.optimize import brentq

class Logistic:
    def solve_dual(self, alpha, beta, tol = 1e-16):
        def qprime(s):
            return -alpha * s + beta + math.log(1-s) - math.log(s)

        # compute [l,u] containing a point with zero qprime
        l = 0.5
        while qprime(l) <= 0:
            l /= 2

        u = 0.5
        while qprime(1 - u) >= 0:
            u /= 2
        u = 1 - u

        solution = brentq(qprime, l, u)
        return solution

    def eval(self, z):
        return math.log(1 + math.exp(z))
```

2.2.5 The Hinge function

For $h(t) = \max(0, t)$, according to Table 1, the dual problem in Eq. (3) amounts to solving the following constrained problem:

$$\max_s \quad q(s) = -\frac{\alpha}{2} s^2 + \beta s \quad \text{s.t.} \quad s \in [0, 1].$$

The above is a maximum of a concave parabola over an interval, and it's easy to see that its maximizer is

$$s^* = \max\left(0, \min\left(1, \frac{\beta}{\alpha}\right)\right).$$

The resulting `Hinge` class is therefore:

```
class Hinge:
    def solve_dual(self, alpha, beta):
        return max(0, min(1, beta / alpha))

    def eval(self, z):
        return max(0, z)
```

2.3 Empirical evaluation

To evaluate the efficiency of our implementation, we compare it against a regular incremental gradient method. Our algorithms in this section are meant to work on one training sample at a time, but incremental gradient methods are usually used with mini-batches of samples. Thus, we'd like to see how competitive is our implementation against incremental gradient methods with various mini-batch sizes, including a mini-batch size of one sample, to appreciate the usefulness of our implementation as a tool by the research community.

We applied our evaluation on the Logistic regression and Least Squares problems, one uses the `Logistic` class whereas the other uses the `HalfSquared` class. We measured the total time to make one epoch over data-sets of various dimensions and of various lengths, and plotted a regression line, whose slope allows us to convince ourselves than our implementation is slower than a regular incremental gradient method by a modest constant factor. The results are plotted in Fig. 1. To summarize, the incremental proximal point method we devised here is roughly 2–8 times slower, per iteration, than an incremental gradient method when mini-batches are used, but faster than an incremental gradient method without mini-batching. It's also interesting to point out that without mini-batching, our implementation is *faster* than PyTorch, mainly due to an overhead of its AutoGrad mechanism with small mini-batches. Therefore, we do not believe it to be a "fair" comparison, but merely a part of the demonstration that our implementation is good enough to be useful.

To convince ourselves that our implementation is indeed correct, we reproduce the results seen in [3] which show that proximal-point methods are much more robust to step-size choice. For randomly generated logistic regression and least-squares problems, we solve both problems using a range of step-sizes by performing one epoch over a random shuffle of the data, and measure the training loss of that epoch. We repeat the experiment for each step-size 30 times to obtain a confidence band around the results. The problems are of dimension 100, and the data-set size is 100,000. So ideally, one epoch should simulate a large random sample from a stationary distribution. The results are plotted in Fig. 2. Indeed, the results look similar to what the authors of Asi and Duchi [3] have obtained.

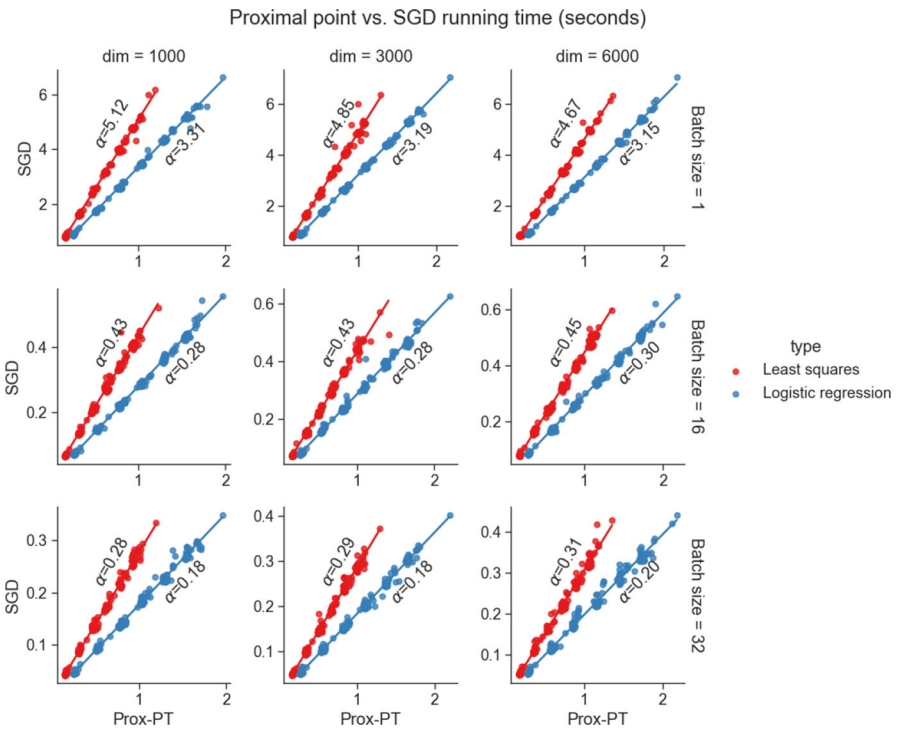


Fig. 1 Execution speed evaluation of incremental proximal point. Each point is a timing of a pair of experiments on the same problem, where the x coordinate is the execution time of one SGD epoch, whereas the y coordinate is the execution time of one proximal point epoch. The corresponding line is a linear regression line, with its slope α labeled, to appreciate the ratio between the SGD and proximal point execution times, on average. The columns are various problem dimensions, from 1000 to 6000, and the rows are various mini-batch sizes for the incremental gradient method. We can see by the first row, for example, that without mini-batching the proximal-point method is actually faster than an incremental gradient method ($\alpha > 0$) based on PyTorch's automatic differentiation. In the last row, for example, we can see that for batch sizes of 32 samples the incremental proximal point method is roughly 4 times slower for least-squares problems and 6 times slower for logistic regression problems compared to an incremental gradient method

2.4 Summary

Typical optimizers for machine learning rely on a well-established tool from analysis—the gradient. Using duality, we were able to design an optimizer for the convex onto linear setup using another well established tool—the convex conjugate. Moreover, duality allowed us to decouple the complexity associated with the convex function h into a separate class whose only purpose is using the conjugate to solve a one dimensional optimization problem. Thus the framework is extensible with additional functions h, and designing a class for such h comprises of two steps: obtaining the conjugate h^*, preferably from a textbook, and figuring out how to solve the resulting dual problem.

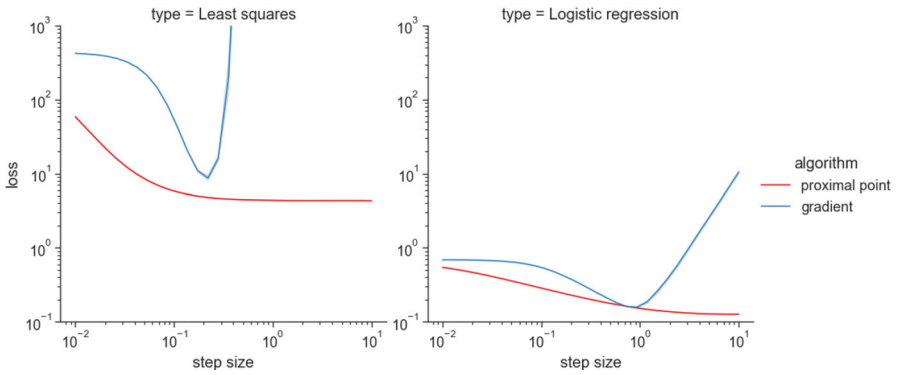


Fig. 2 Reproduction of the stability results of Asi and Duchi [3]. Indeed, for both logistic regression and least squares problems, the training loss of an incremental proximal-point method is significantly more stable w.r.t the step-size choice, reinforcing the claim that with an incremental proximal-point method we can just "make an few educated guesses" instead of performing an extensive hyper-parameter search for the best step-size

3 Regularized convex onto linear composition

Often, machine learning models are trained by using a regularized loss function. Thus, the applications of the regularized convex-onto-linear model (RCL) are identical to the described in Sect. 2.1, e.g. regularized linear least squares, or regularized logistic regression. It's also interesting to note that, using $h(z) = \max(0, z)$ and $r(\mathbf{x}) = \mu\|\mathbf{x}\|_2^2$ we can also formulate the problem of training an SVM [14]. Since the applications are obvious, we dive directly into the computation of the proximal operator, by solving

$$\min_{\mathbf{x}} \quad h(\mathbf{a}^T\mathbf{x} + b) + r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2, \tag{5}$$

where $h : \mathbb{R} \to (-\infty, \infty]$, $r : \mathbb{R}^d \to (-\infty, \infty]$, are convex extended real-valued functions. We also assume that the regularizer r is "simple", meaning that we can efficiently compute its proximal operator $\text{prox}_r(\mathbf{x})$. Explicit formulae for the proximal operators can be found in a variety of textbooks on optimization, e.g. [7, Chapt. 6]. Examples include the squared Euclidean norm $r(\mathbf{x}) = \frac{\mu}{2}\|\mathbf{x}\|_2^2$, the Euclidean norm $r(\mathbf{x}) = \mu\|\mathbf{x}\|_2$, or the ℓ_1 norm $r(\mathbf{x}) = \mu\|\mathbf{x}\|_1$. For completeness, we include a short table of proximal operators of the above-mentioned functions in Table 2.

3.1 A dual problem

Following the path paved in Sect. 2, we begin from deriving a dual to the problem in Eq. (5). Adding the auxiliary variable $z = \mathbf{a}^T\mathbf{x} + b$, we obtain the equivalent optimization problem

$$\min_{\mathbf{x},z} \quad h(z) + r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 \quad \text{s.t.} \quad z = \mathbf{a}^T\mathbf{x} + b$$

Table 2 Proximal operators of commonly used regularizers

$r(\mathbf{x})$	$\text{prox}_{\mu r}(\mathbf{x})$	Remarks
$\|\mathbf{x}\|_1$	$\max(\|\mathbf{x}\| - \mu\mathbf{1}, \mathbf{0}) \cdot \text{sgn}(\mathbf{x})$	Absolute value, max, sgn, and $\cdot$ are component-wise operations. Available as the `softshrink` function in PyTorch
$\frac{1}{2}\|\mathbf{x}\|_2^2$	$\frac{1}{1+\mu}\mathbf{x}$	
$\|\mathbf{x}\|_2$	$\left(1 - \frac{\mu}{\max(\mu, \|\mathbf{x}\|_2)}\right)\mathbf{x}$	

Minimizing the Lagrangian, we obtain:

$$\begin{aligned} q(s) &= \inf_{\mathbf{x},z} \left\{ \mathcal{L}(\mathbf{x}, z, s) \equiv h(z) + r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 + s(\mathbf{a}^T\mathbf{x} + b - z) \right\} \\ &= \inf_{\mathbf{x}} \left\{ r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 + s\mathbf{a}^T\mathbf{x} \right\} + \inf_{z}\{h(z) - sz\} + sb \end{aligned} \tag{6}$$

The remaining challenge is the computing the first infimum. To that end, we need to introduce another well-known concept in optimization—a close relative of the proximal operator.

Definition 1 (*Moreau envelope* [28]) Let $\phi : \mathbb{R}^n \to (-\infty, \infty]$ be a convex extended real-valued function. The Moreau envelope of ϕ with parameter η, denoted by $M_\eta\phi$, is the function:

$$M_\eta\phi(\mathbf{x}) = \min_{\mathbf{u}} \left\{ \phi(\mathbf{u}) + \frac{1}{2\eta}\|\mathbf{u} - \mathbf{x}\|_2^2 \right\}.$$

Since the proximal operator is the minimizer of the minimum in the definition above, which is Moreau proved that is always attained, an alternative way to write the Moreau envelope of a function is obtained by replacing $\mathbf{u} = \text{prox}_{\eta\phi}(\mathbf{x})$ inside the minimization objective above:

$$M_\eta\phi(\mathbf{x}) = \phi(\text{prox}_{\eta\phi}(\mathbf{x})) + \frac{1}{2\eta}\|\text{prox}_{\eta\phi}(\mathbf{x}) - \mathbf{x}\|_2^2. \tag{7}$$

Thus, whenever we have an explicit formula of the proximal operator, we also have an explicit formula of the Moreau envelope. How does it help us with our challenge of minimizing $\mathcal{Q}$ over $\mathbf{x}$? The following proposition provides the answer.

Proposition 2 *Let $q(s)$ and the Lagrangian $\mathcal{L}(\mathbf{x}, z, s)$ be as defined in Eq. (6). Then,*

$$q(s) = M_\eta r(\mathbf{x}_t - \eta s\mathbf{a}) + (\mathbf{a}^T\mathbf{x}_t + b)s - \frac{\eta\|\mathbf{a}\|_2^2}{2}s^2 - h^*(s).$$

Moreover, the unique minimizer of the Lagrangian $\mathcal{L}$ w.r.t $\mathbf{x}$ is

$$\mathbf{x}^* = \text{prox}_{\eta r}(\mathbf{x}_t - \eta s\mathbf{a}).$$

Proof Recall, that for any $\mathbf{x}, \mathbf{y}$ we can open the squared Euclidean norm using the formula

$$\frac{1}{2}\|\mathbf{x}+\mathbf{y}\|_2^2 = \frac{1}{2}\|\mathbf{x}\|_2^2 + \mathbf{x}^T\mathbf{y} + \frac{1}{2}\|\mathbf{y}\|_2^2,$$

and re-arranging the above leads to the *square completion* formula

$$\frac{1}{2}\|\mathbf{x}\|_2^2 + \mathbf{x}^T\mathbf{y} = \frac{1}{2}\|\mathbf{x}+\mathbf{y}\|_2^2 - \frac{1}{2}\|\mathbf{y}\|_2^2.$$

Using the above two formulas, we compute:

$$\begin{aligned}
&r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x}-\mathbf{x}_t\|_2^2 + s\mathbf{a}^T\mathbf{x} \\
&= \frac{1}{\eta}\left[\eta r(\mathbf{x}) + \frac{1}{2}\|\mathbf{x}-\mathbf{x}_t\|_2^2 + \eta s\mathbf{a}^T\mathbf{x}\right] && \leftarrow \text{Factoring out } \frac{1}{\eta} \\
&= \frac{1}{\eta}\left[\eta r(\mathbf{x}) + \frac{1}{2}\|\mathbf{x}\|_2^2 - (\mathbf{x}_t - \eta s\mathbf{a})^T\mathbf{x} + \frac{1}{2}\|\mathbf{x}_t\|_2^2\right] && \leftarrow \text{opening } \frac{1}{2}\|\mathbf{x}-\mathbf{x}_t\|_2^2 \\
&= \frac{1}{\eta}\left[\eta r(\mathbf{x}) + \frac{1}{2}\|\mathbf{x}-(\mathbf{x}_t - \eta s\mathbf{a})\|_2^2 - \frac{1}{2}\|\mathbf{x}_t - \eta s\mathbf{a}\|_2^2 + \frac{1}{2}\|\mathbf{x}_t\|_2^2\right] && \leftarrow \text{square completion} \\
&= \left[r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x}-(\mathbf{x}_t - \eta s\mathbf{a})\|_2^2\right] - \frac{1}{2\eta}\|\mathbf{x}_t - \eta s\mathbf{a}\|_2^2 + \frac{1}{2\eta}\|\mathbf{x}_t\|_2^2 && \leftarrow \text{Multiplying by } \frac{1}{\eta} \\
&= \left[r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x}-(\mathbf{x}_t - \eta s\mathbf{a})\|_2^2\right] + (\mathbf{a}^T\mathbf{x}_t)s - \frac{\eta\|\mathbf{a}\|_2^2}{2}s^2 && \leftarrow \text{re-arranging}
\end{aligned}$$

Plugging the above expression into the formula of $q(s)$, we obtain:

$$\begin{aligned}
q(s) &= \inf_{\mathbf{x}}\left\{r(\mathbf{x}) + \frac{1}{2\eta}\|\mathbf{x}-(\mathbf{x}_t - \eta s\mathbf{a})\|_2^2\right\} + (\mathbf{a}^T\mathbf{x}_t + b)s - \frac{\eta\|\mathbf{a}\|_2^2}{2}s^2 - h^*(s) \\
&= M_\eta r(\mathbf{x}_t - \eta s\mathbf{a}) + (\mathbf{a}^T\mathbf{x}_t + b)s - \frac{\eta\|\mathbf{a}\|_2^2}{2}s^2 - h^*(s)
\end{aligned}$$

Moreover, since the infimum over $\mathbf{x}$ above is attained, we can replace it with a minimum, and by definition the minimizer is

$$\mathbf{x}^* = \text{prox}_{\eta r}(\mathbf{x}_t - \eta s\mathbf{a})$$

The significance of Proposition 2 is due to the fact that we can design an algorithm for computing the proximal operator of regularized convex onto linear losses using three textbook concepts: the Moreau envelope of r, the convex conjugate of h, and the proximal operator of r. The only thing we need to manually derive ourselves is a way to maximize the dual objective q. The basic method, directly applying Proposition 2 and the strong duality theorem (Theorem 1) consists of the following three steps:

- Form $q(s) = M_\eta r(\mathbf{x}_t - \eta s\mathbf{a}) + (\mathbf{a}^T\mathbf{x}_t + b)s - \frac{\eta\|\mathbf{a}\|_2^2}{2}s^2 - h^*(s)$.

- Solve the dual problem: find a maximizer s^* of $q(s)$
- Compute $\text{prox}_f(\mathbf{x}_t) = \text{prox}_{\eta r}(\mathbf{x}_t - \eta s^* \mathbf{a})$

3.2 Computing the proximal operator

An important aspect of solving the dual problem is figuring out $\text{dom}(-q)$, since this set encodes the constraints of the dual problem. It turns out that Moreau envelopes are finite everywhere [7, Theorem 6.55], and thus $\text{dom}(-q) = \text{dom}(h^*)$. Moreover, the strong duality theorem (Theorem 1) ensures that q attains its maximum, so a maximizer $s^* \in \text{dom}(h^*)$ *must* exist.

The dual problem is *one dimensional*, and there is a variety of reliable algorithms and their implementations for maximizing such functions. Indeed, this time we will not bother implementing a procedure to maximize q, but use a readily available implementations in the `SciPy` package. When $\text{dom}(-q)$ is a compact interval, we will employ the `scipy.optimize.fminbound` function on $-q$, which uses Brent's method [10, Chapter 5], and requires us to provide a function, and a compact interval where its maximizer must lie.

When the interval $\text{dom}(-q)$ is not compact, we will have to locate a compact interval which contains a maximizer of q. To that end, we will require h^* to be continuously differentiable, strictly convex, and $\text{dom}(h^*)$ to be open. The object representing h needs to provide two sequences $l_1 > l_2 > \ldots$ converging to the left endpoint of $\text{dom}(-q)$, and $u_1 < u_2 < \ldots$ converging to the right endpoint of $\text{dom}(-q)$. We will shortly see the details, but the general idea of using these sequences is similar to our search for an initial interval in case of the `Logistic` class we implemented in Sect. 2.2.4, where the sequences were $2^{-1}, 2^{-2}, \ldots$, and $1-2^{-1}, 1-2^{-2}, \ldots$. Having found an interval containing a maximizer, we will, again, employ the `scipy.optimize.fminbound` on $-q$ function to find our maximizer.

Remark Note, that the Scipy's `scipy.optimize.minimize_scalar` function also supports finding the initial *bracket* for Brent's method when we don't possess a compact interval containing a minimizer. However, as of the time of writing of this paper, the `minimize_scalar` function is not able to handle half-infinite, such as $[0, \infty)$. It supports either a compact domain, or the entire real line. Thus, to be as generic as possible, we resort to finding the initial interval ourselves using the above-mentioned pairs of sequences.

The case of a compact domain To employ Brent's method for finding a maximizer s^* of q, we need to be able to evaluate the function q. To that end, we require an oracle for evaluating the Moreau envelope of r, and the convex conjugate h^*. Moreover, to recover the solution of the primal problem, we need an oracle for computing the proximal operator of r.

The case of a non-compact domain This case requires us to employ an additional result about Moreau envelopes.

Proposition 3 *Let $\phi : \mathbb{R}^d \to (-\infty, \infty]$ be a convex extended real-valued function, and let $M_\mu \phi$ be its Moreau envelope. Then $M_\mu \phi$ is continuously differentiable with gradient*

$$\nabla M_{\mu}\phi(\mathbf{x}) = \frac{1}{\mu}\left(\mathbf{x} - \text{prox}_{\mu\phi}(\mathbf{x})\right) \tag{8}$$

Proof See [7, Theorem 6.55]. □

Recall, that we require h^* to be continuously differentiable, strictly convex, and $\text{dom}(h^*)$ to be open. Using (8) and the chain rule to compute the derivative of $q(s)$, and then simplifying, we obtain:

$$q'(s) = \mathbf{a}^T \text{prox}_{\eta r}(\mathbf{x}_t - \eta s\mathbf{a}) - h^{*\prime}(s) + b \tag{9}$$

For the non-compact domain case, we will require that there is a unique maximizer s^* in the *interior* of $\text{dom}(h)$, which also implies that $q'(s^*) = 0$. The above is satisfied when, for example, when h^* is strictly convex and $\text{dom}(h^*)$ is open, or when h^* is a *convex function of Legendre type* [32, Sect. 26], which is a condition satisfied by, for example, by $h^*(s) = \frac{1}{2}s^2$ in the least-squares setup. Strict convexity of h^* implies that q' is strictly decreasing, and thus any point $l < s^*$ has a positive derivative, whereas any $u > s^*$ has a negative derivative. Thus, we can find an interval $[l, u]$ containing s^* just by scanning to the left until we find a point whose derivative is positive, and to the right until we find a point whose derivative is negative. And that's exactly why we need the sequences $\{l_k\}_{k=1}^{\infty}$ and $\{u_k\}_{k=1}^{\infty}$ for - to tell us how to scan towards the left and right boundaries of $\text{dom}(-q) = \text{dom}(h^*)$. For example, if $\text{dom}(h^*) = [1, \infty)$, we may scan towards the left end-point using the sequence $\ell_k = 1 + 2^{-k}$, and towards the right-endpoint, which is infinity, using the sequence $u_k = 1 + 2^k$.

It is important to note that computing q' requires evaluating the proximal operator of the regularizer r. Therefore, each iteration in the search for the optimal s^* *depends on the dimension* d, and hence is typically significantly slower than the convex onto linear setting without regularization, that was discussed in Sect. 2. Of course, it would be possible to construct a dedicated procedure for each combination of outer function h and regularizer r, which may occasionally be fast, but it defeats one of the main purposes of our framework - its modularity.

The fact that the computational complexity of each iteration is linear in the dimension suggests that we could have employed an accelerated first-order method, such as [8], directly on the primal proximal sub-problem. To see why we chose *not* to pursue this direction, note that Brent's method [10] used by `fminbound` in our code enjoys *superlinear convergence* (of degree 1.67) in the vicinity of the optimum. Accelerated first-order methods have a convergence rate that is linear, at best, when their objective is strongly convex. This suggests that Brent's method, in addition to fitting well into our dual decomposition approach, is also potentially more efficient due to reduced iteration count. We chose to rely on the fact that for one-dimensional problems we have specialized methods that are often faster than methods that work in arbitrary dimensions.

The implementation Combining the two cases above, here is an implementation of our optimizer.

```
from scipy.optimize import fminbound
import torch

class IncRegularizedConvexOnLinear:
    def __init__(self, x, h, r):
        self._x = x
        self._h = h
        self._r = r

    def step(self, eta, a, b):
        x = self._x
        h = self._h
        r = self._r

        if torch.is_tensor(b):
            b = b.item()

        lin_coef = (torch.dot(a, x) + b).item()
        quad_coef = (eta / 2.) * a.square().sum().item()
        loss = h.eval(lin_coef) + r.eval(x).item()

        def qprime(s):
            prox = r.prox(eta, x - eta * s * a)
            return torch.dot(a, prox).item() \
                   - h.conjugate_prime(s) \
                   + b

        def q(s):
            return r.envelope(eta, x - eta * s * a) \
                   + lin_coef * s \
                   - quad_coef * (s ** 2) \
                   - h.conjugate(s)

        if h.conjugate_has_compact_domain():
            l, u = h.domain()
        else:
            # scan left until a positive derivative is found
            l = next(s for s in h.lower_bound_sequence() if qprime(s) > 0)

            # scan right until a negative derivative is found
            u = next(s for s in h.upper_bound_sequence() if qprime(s) < 0)

        min_result = fminbound(lambda s: -q(s), l, u)
        s_prime = min_result.x
        x.set_(r.prox(eta, x - eta * s_prime * a))

        return loss
```

Let's look at a usage example. Assuming that $h(z) = \frac{1}{2}z^2$ is represented using the `HalfSquared` class, and $r(\mathbf{x}) = \|\mathbf{x}\|_1$ is represented using the `L1Reg` class, we can perform an epoch of training an L1 regularized least-squares model (Lasso) with regularization parameter 0.1 using the following code:

```
x = torch.zeros(d)

optimizer = IncRegularizedConvexOnLinear(x, HalfSquared(), L1Reg(0.1))
epoch_loss = 0.
for t, (a, b) in enumerate(get_training_data()):
    eta = get_step_size(t)
    epoch_loss += optimizer.step(eta, a, b)

print('Model parameters = ', x)
print('Average epoch loss = ', epoch_loss / t)
```

We believe that Sect. 2 demonstrated how to modularity is achieved, and therefore the implementation of the `HalfSquared`, `Logistic`, and `Hinge` classes for the functions h representing various losses, and of the `L1Reg` and `L2Reg`, and `L2NormReg` classes for regularization with $\|\cdot\|_1$, $\|\cdot\|_2^2$, and $\|\cdot\|_2$ are in Appendix A.

3.3 Empirical evaluation

Before evaluating the efficiency empirically, a word about the computational complexity is in place. Computing the proximal step requires solving a one-dimensional dual problem using an optimization algorithm over the real line. The algorithms employed by the `fminbound` function typically achieve ε accuracy in terms of distance to a minimize using $O(\ln(\frac{1}{\varepsilon}))$ function evaluations. By default, SciPy's implementation achieves $\varepsilon = 10^{-10}$, meaning that $\ln(\varepsilon^{-1}) \approx 23$. However, this time, evaluating $q(s)$ requires computing the proximal operator of our regularizer, which entails a computational complexity of $O(n)$ for $\mathbf{x} \in \mathbb{R}^n$. This is in contrast to the convex-onto-linear case we saw in Sect. 2, where evaluating $q(s)$ was independent of the problem's dimension. The computational cost per iteration is dominated by few dozen evaluations of $\text{prox}_{\eta r}$, and thus we expect it to be a few dozen times slower than a regular proximal-gradient step (without mini-batches).

As was the case with the convex-onto-linear setting, We applied our speed evaluation on the Logistic regression and Least Squares problems, both with L1 and L2 regularization. We measured the total time to make one epoch over data-sets of various dimensions and of various lengths, and plotted a regression line, so that we can indeed convince ourselves than our implementation is slower than a regular incremental gradient method by a small-enough constant factor. The results are plotted in Fig. 3. As a summary, without mini-batching, the proximal-point method is roughly 5–10 times slower than its proximal-gradient counterpart. However, mini-batches may make an incremental gradient method computational faster by a factor of 20–150, depending on the problem. As we pointed out, this slowdown is mainly caused by the significantly costlier solution of the dual problem. Even though it appears to be a one-dimensional optimization problem, the computational complexity of evaluating the derivative of the one-dimensional objective does depend on the problem dimension.

Regarding correctness, we again adopt a similar strategy to the convex-onto-linear setting, and solve various problem types with a variety of step-sizes, and see the achieved training loss of one epoch over a data-set of 10,000 samples in $\mathbb{R}^{100}$. The results are plotted in Fig. 4.

3.4 Summary

Throughout this paper, our aim is to design implementations of proximal point algorithms with two objectives in mind: (a) make the algorithms extensible by decoupling concerns into separate modules, and (b) use existing text-book concepts and software libraries to help build these modules. In the case of convex onto linear composition, we were able to decouple the function h and the regularizer r into separate modules

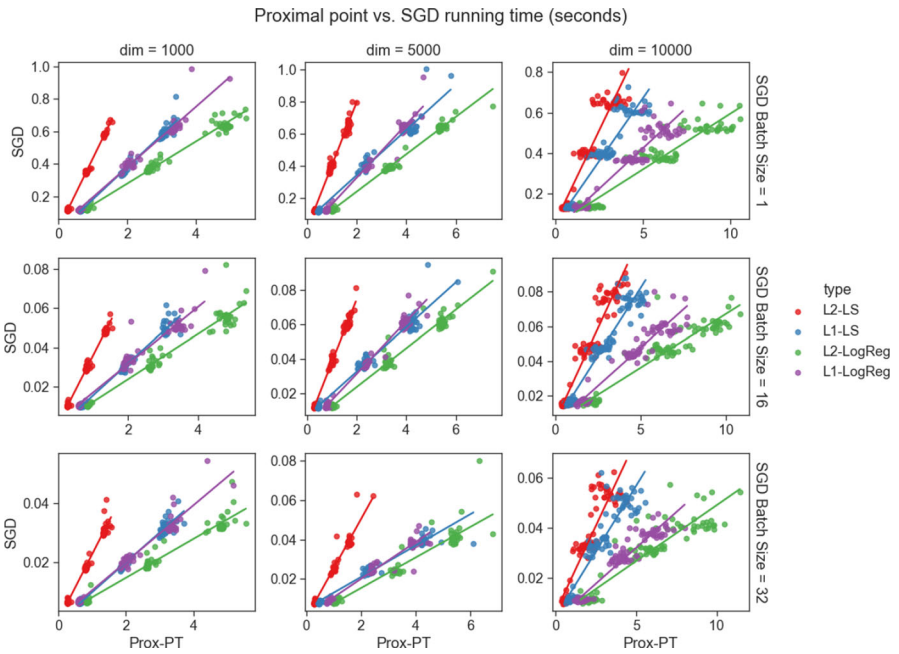


Fig. 3 Execution speed evaluation of incremental proximal point. Each point is a timing of a pair of experiments on the same problem, where the x coordinate is the execution time of one SGD epoch, whereas the y coordinate is the execution time of one proximal point epoch. The columns are various problem dimensions, and the rows are various mini-batch sizes for the incremental proximal gradient method. The columns are various problem dimensions, and the rows are various mini-batch sizes for the incremental proximal gradient method. We can see by the first row, for example, that without mini-batching the proximal-point method is 5–10 times slower than its proximal-gradient variant. In the last row, for example, we can see that for batch sizes of 32 samples the incremental proximal point method is roughly 20–150 times slower, depending on the problem and the dimension

which provide oracles to a central algorithm. The oracles are implemented using three textbook concepts: the convex conjugate of h, the Moreau envelope of r, and the proximal operator of r. Using textbook concepts allows easier extension of our library with additional functions h and r, since we stand on the shoulders of the giants who already developed tables and calculus rules of convex conjugates and proximal operators for a variety of useful functions.

An alternative approach to using duality could be using a generic solver package, such as `CVXPY` [18], to directly solve the proximal sub-problem. The numerical experiments of the next section show that this approach is significantly slower in practice for the setting at hand, where we aim to learn from one sample at a time, rather than from a mini-batch of samples. Indeed, our solver is roughly 5–10 times slower than SGD, whereas the CVXPY approach for small batch sizes appears to be at least 30 times slower (see Fig. 5b).

Another alternatives for solving the proximal sub-problem could be primal-dual hybrid gradient methods, such as the celebrated Chambolle–Pock method [12], and their accelerated variants [13]. However, this would result in a radically different

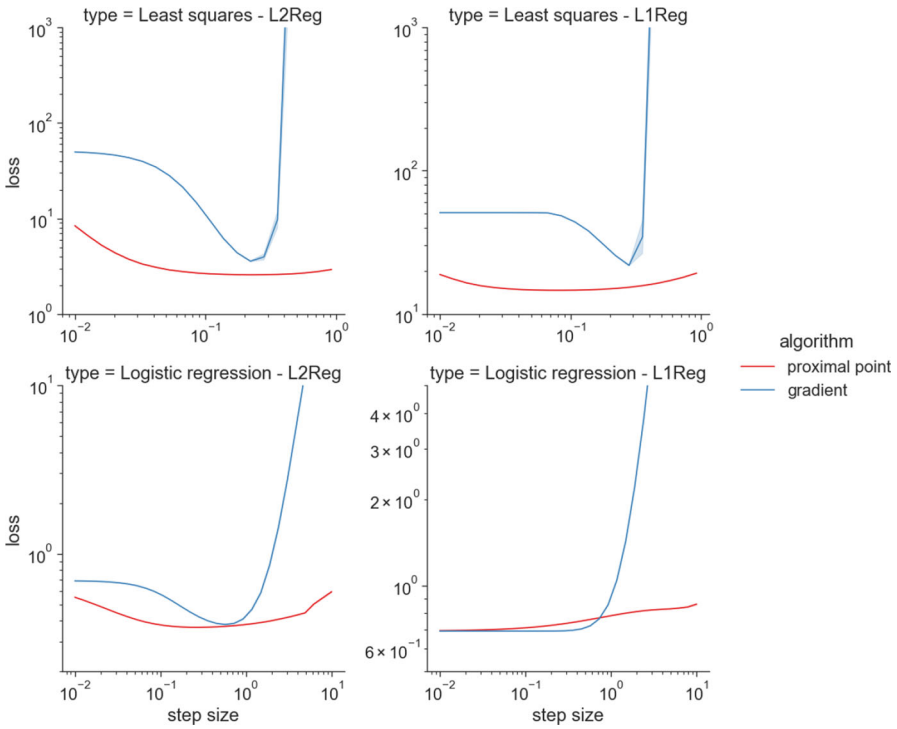


Fig. 4 Reproduction of the stability results of Asi and Duchi [3]. Indeed, for both logistic regression and least squares problems, with both L1 and L2 regularization, the training loss of an incremental proximal-point method is significantly more stable w.r.t the step-size choice than the incremental proximal-gradient method, reinforcing the claim that with an incremental proximal-point method we can just "make an few educated guesses" instead of performing an extensive hyper-parameter search for the best step-size

decomposition into software components than our duality-based approach. Therefore, these approaches are out of the scope of this paper, but are worth exploring in a future work.

4 Mini-batch convex onto linear composition

The mini-batch convex onto linear compositions are useful when extending the scenarios described in Sect. 2.1 to handle mini-batches of data, instead of individual data points. For example, suppose we're aiming to minimize

$$F(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\mathbf{x}),$$

and that the size of the data-set N is huge. A standard practice with incremental optimization methods, such as SGD, is to use mini-batches of items: a mini-batch $B \subseteq \{1, \ldots, N\}$ is chosen in each iteration, and the model's parameters are updated using the gradient of the function

$$F_B(\mathbf{x}) = \frac{1}{|B|}\sum_{j\in B} f_i(\mathbf{x}).$$

The idea above is equivalent to using a linear approximation of F_B in each iteration. For example, the steps of mini-batch SGD are:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\nabla F_B(\mathbf{x}_t) = \operatorname*{argmin}_{\mathbf{x}} \left\{ F_B(\mathbf{x}_t) + \langle \nabla F_B(\mathbf{x}_t), \mathbf{x} - \mathbf{x}_t\rangle + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 \right\}.$$

If our functions f_i are of the form $h(\mathbf{a}_i^T\mathbf{x} + b_i)$ with h being a convex extended real-valued function, or if we chose to approximate f_i using such functions, then by replacing the linear approximation $\mathbf{x}_{t+1}$ is computed by solving

$$\mathbf{x}_{t+1} = \operatorname*{argmin}_{x} \left\{ \frac{1}{|B|}\sum_{i\in B} h(\mathbf{a}_i^T\mathbf{x} + b_i) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 \right\}.$$

Assuming w.l.o.g that $B = \{1, \ldots, m\}$, the above is exactly of the form in Eq. (CL-B). As was the case with our previous derivations, convex duality plays a central role in computing the above proximal operator. We note that an alternative formulation for mini-batching could be using the proximal average [6, 36], rather than the arithmetic average of functions. In such a formulation, the proximal operator of the mini-batch reduces to the average of the proximal operators of the individual functions, and can be computed using the tools developed in Sect. 2. However, we also note that the interplay of such a formulation with the stochastic optimization setting is not clear - what is the bias, if any, of the proximal average as an estimator for the mean loss?

4.1 A dual problem

Embedding the vectors $\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T$ into the rows of the matrix $\mathbf{A}$, and the scalars $b_1, \ldots, b_m$ into the vector $\mathbf{b}$, and introducing the auxiliary variable $\mathbf{z} = \mathbf{A}\mathbf{x} + \mathbf{b}$, our optimization problem can be equivalently written as

$$\min_{\mathbf{x},\mathbf{z}} \quad \frac{1}{m}\sum_{i=1}^{m} h(z_i) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 \quad \text{s.t.} \quad \mathbf{z} = \mathbf{A}\mathbf{x} + \mathbf{b}.$$

The corresponding dual problem aims to maximize

$$\begin{aligned} q(\mathbf{s}) &= \inf_{\mathbf{x},\mathbf{z}} \left\{ \frac{1}{m}\sum_{i=1}^{m} h(z_i) + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 + \mathbf{s}^T(\mathbf{A}\mathbf{x} + \mathbf{b} - \mathbf{z}) \right\} \\ &= \underbrace{\inf_{\mathbf{x}} \left\{ (\mathbf{A}^T\mathbf{s})^T\mathbf{x} + \frac{1}{2\eta}\|\mathbf{x} - \mathbf{x}_t\|_2^2 \right\}}_{(*)} + \sum_{z=1}^{m} \underbrace{\inf_{z_i} \left\{ \frac{1}{m}h(z_i) - z_i s_i \right\}}_{(**)} + \mathbf{s}^T\mathbf{b} \end{aligned}$$

The term denoted by (*) above, despite its "hairy" appearance, is the minimum of a simple convex quadratic function, which is computed by comparing the gradient of the term inside the inf with zero, which leads to

$$\mathbf{x} = \mathbf{x}_t - \eta \mathbf{A}^T \mathbf{s}, \tag{10}$$

and the corresponding minimum is

$$(*) = -\frac{\eta}{2}\|\mathbf{A}^T\mathbf{s}\|_2^2 + (\mathbf{A}\mathbf{x}_t)^T\mathbf{s}.$$

The terms denoted by (**) can be equivalently written as

$$\inf_{z_i}\left\{\frac{1}{m}h(z_i) - z_i s_i\right\} = -\frac{1}{m}\sup_{z_i}\{(ms_i)z_i - h(z_i)\} = -\frac{1}{m}h^*(ms_i),$$

where, again, h^* denotes the convex conjugate of h. To summarize, the dual aims to solve:

$$\max_{\mathbf{s}} \quad q(\mathbf{s}) = -\frac{\eta}{2}\|\mathbf{A}^T\mathbf{s}\|_2^2 + (\mathbf{A}\mathbf{x}_t + \mathbf{b})^T\mathbf{s} - \frac{1}{m}\sum_{i=1}^{m} h^*(ms_i).$$

4.2 Computing the proximal operator

Recall, that the strong duality theorem (Theorem 1) ensures that q has a maximizer, and that Eq. (10) recovers the primal minimizer, which is what we aim to compute. Thus, computing our proximal operator amounts to the following three steps:

1. Form $q(\mathbf{s}) = -\frac{\eta}{2}\|\mathbf{A}^T\mathbf{s}\|_2^2 + (\mathbf{A}\mathbf{x}_t + \mathbf{b})^T\mathbf{s} - \frac{1}{m}\sum_{i=1}^{m} h^*(ms_i)$
2. Solve the dual problem: find a maximizer $\mathbf{s}^*$ of $q(\mathbf{s})$
3. Recover the desired solution: $\mathbf{x}_t - \eta\mathbf{A}^T\mathbf{s}^*$

Note, that in the mini-batch setting, the dual problem is no longer *one dimensional*, but when the size of the mini-batch m is small, it is still a *low dimensional* problem, whose solution should be pretty quick. As was the case in Sect. 2, our representation of the function h amounts to a class which provides two oracles: (a) evaluate the function h, and (b) maximize functions of the form $\frac{1}{2}\|\mathbf{P}\mathbf{s}\|_2^2 + \mathbf{c}^T\mathbf{s} - \frac{1}{m}\sum_{i=1}^{m} h^*(ms_i)$. Let's first implement a generic optimizer, and then dive into the implementation of concrete functions h.

```
import torch

class MiniBatchConvLinOptimizer:
    def __init__(self, x, phi):
        self._x = x
        self._phi = phi

    def step(self, step_size, A_batch, b_batch):
        # helper variables
        x = self._x
```

```
        phi = self._phi

        # compute dual problem coefficients
        P = math.sqrt(step_size) * A_batch.t()
        c = torch.addmv(b_batch, A_batch, x)

        # solve dual problem
        s_star = phi.solve_dual(P, c)

        # perform step
        step_dir = torch.mm(A_batch.t(), s_star)
        x.sub_(step_size * step_dir.reshape(x.shape))

        # return the mini-batch losses w.r.t the params before making the step
        return phi.eval(c)
```

We will shortly implement $h(z) = \frac{1}{2}z^2$ in the `HalfSquared` class, $h(z) = \ln(1 + \exp(z))$ in the `Logistic` class, and $h(z) = \max(0, z)$ in the `Hinge` class. With the above we can now, for example, solve a linear least-squares problem using mini-batches of training samples:

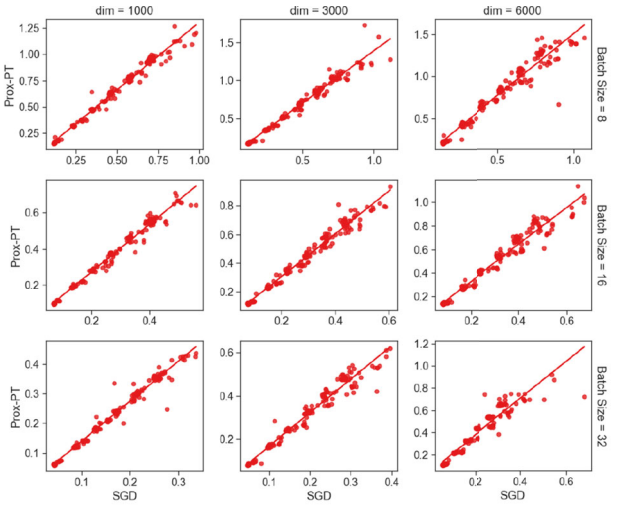
```
x = torch.zeros(dim)
optimizer = MiniBatchConvLinOptimizer(x, HalfSquared())
dataset = get_my_dataset()
for t, (A_batch, b_batch) in enumerate(DataLoader(dataset, batch_size=32)):
    step_size = get_step_size(t)
    optimizer.step(step_size, A_batch, b_batch)

print('The model parameter vector is', x)
```

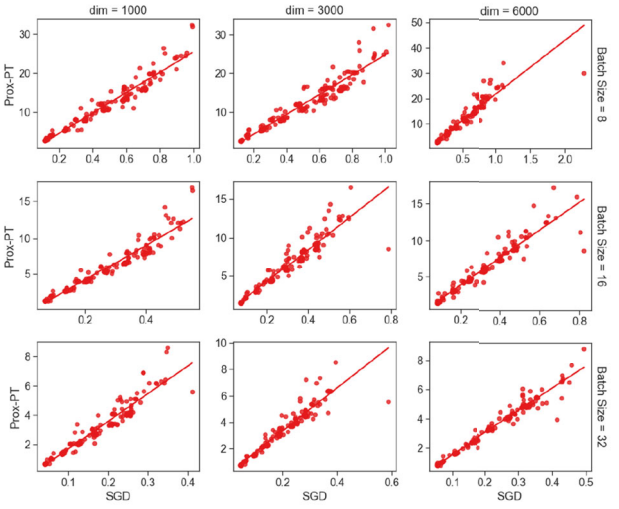
The concrete implementation of various functions h can be found in Appendix B. We note that the dual problem for $h(z) = \frac{1}{2}z^2$ can be computed analytically, but for the hinge and logistic functions it cannot, and therefore we employ `CVXPY` [1, 18], which in fact is an interface to a variety of lower-level convex optimization solvers. Since the dual problem's dimension is proportional to the mini-batch size, which is typically quite small, solving it should still be moderately fast. This is, again, the case when a combined C/Python implementation could be more efficient than pure Python code, since we could use C to implement a dedicated efficient convex solver for the dual problem of each loss.

4.3 Empirical evaluation

In contrast to the previous problem families, this time our proximal-point solver can handle mini-batches of data. Thus, when comparing execution speed, we compare the same mini-batch size for both our method and the incremental gradient method. Figure 5 contains the plots for various mini-batch sizes and two problem types - linear least-squares and logistic regression. We see that for least-squares problems, the mini-batched proximal point method is on par with the incremental gradient method. However, logistic regression problems employ the `Logistic` class which incurs the overhead of the `CVXPY` framework, and is approximately 15 times slower for mini-batches of 32 samples, and approximately 100 times slower for a mini-batch of 8 samples. The overhead of `CVXPY` is apparent, however, it isn't our aim to design and develop dedicated solvers for high-dimensional convex optimization problems.



(a) Least squares proximal point vs. SGD running time (seconds)



(b) Logistic regression proximal point vs. SGD running time (seconds)

Fig. 5 Execution speed evaluation of mini-batch incremental proximal point. Each point is a timing of a pair of experiments on the same problem, where the x coordinate is the execution time of one SGD epoch, whereas the y coordinate is the execution time of one mini-batch proximal point epoch. Points differ in running times due to generated data-set size. The columns are various problem dimensions, and the rows are various mini-batch sizes for the incremental gradient method. The corresponding line is a least-squares regression line, whose slope allows to appreciate the ratio between the SGD and proximal point running times. For least-squares problems, the mini-batched proximal point method is on par with the incremental gradient method. However, logistic regression problems employ the `Logistic` class which incurs the overhead of the `CVXPY` framework and of a generic conic solver, and is approximately 15 times slower for mini-batches of 32 samples, and approximately 100 times slower for a mini-batch of 8 samples, where CVXPY's overhead is more significant

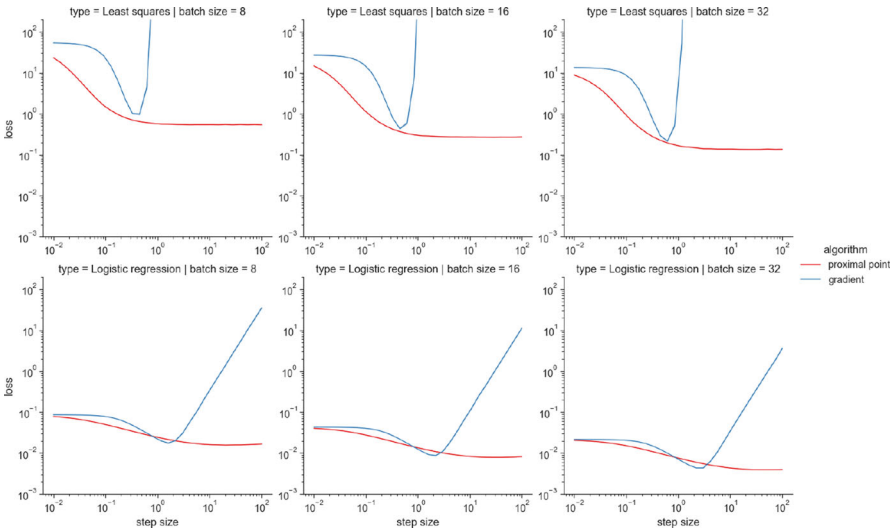


Fig. 6 Results of solving logistic regression and linear least-squares problems using an incremental gradient method and our proximal-point with mini-batches implementation

However, we encourage our readers to do so, or find a faster third-party solver, if they desire a production-grade optimizer for their model.

And again, to see that we indeed harvest the fruits of a proximal-point algorithm, we conduct a stability experiment, where we solve logistic regression and least-squares problems, and expect to see the resulting algorithm being much more stable, with respect to the step-size choice, than an incremental gradient method. The results are plotted in Fig. 6, where the stability is apparent - we indeed obtain a low training loss value for a large range of step-sizes, in contrast to the incremental gradient method, where we need to "pinpoint" the correct step-size to obtain good performance.

4.4 Summary

Duality played a central role here as well, but instead of reducing the proximal operator problem to a one-dimensional problem, we reduced the proximal operator to a, hopefully, low dimensional problem whose dimension is the size of the mini-batch. In practice, mini-batches typically have less than 128 samples, and thus our dual problems are of very low dimensions and can be solved extremely quickly. There is, of course, an additional overhead incurred by using a commodity optimization package such as CVXPY, and a dedicated solver for each dual problem could be substantially faster. However, writing robust and efficient convex optimziation solvers is out of the scope of this paper.

A Code for regularized convex onto linear compositions

A.1 The `HalfSquared` class

For $h(z) = \frac{1}{2}z^2$, looking at the Table 1, we see that $h^*(s) = \frac{1}{2}s^2$ with $\mathrm{dom}(h^*) = \mathbb{R}$, i.e. its conjugate is itself. Since the domain is non-compact and open, and h^* is strictly convex, we perfectly fit the non-compact case. We will use the sequence $-1, -2, -2^2, -2^3, \ldots$ for the lower bounds, and $1, 2, 2^2, 2^3, \ldots$ for upper bounds. The implementation is below:

```
from itertools import count

class HalfSquared:
    def eval(self, z):
        return (z ** 2) / 2

    def conjugate_has_compact_domain(self):
        return False

    def lower_bound_sequence(self):
        return (-(2 ** j) for j in count())

    def upper_bound_sequence(self):
        return ((2 ** j) for j in count())

    def conjugate(self, s):
        return (s ** 2) / 2

    def conjugate_prime(self, s):
        return s
```

A.2 The `Logistic` class

For $h(z) = \ln(1 + \exp(z))$, looking at Table 1, we see that $h^*(s) = s\ln(s) + (1 - s)\ln(1 - s)$ with the convention that $0\ln(0) = 0$, and that the domain of h^* is the compact interval $[0, 1]$. Based on the above, we implement the `Logistic` class below:

```
import math

class Logistic:
    def eval(self, z):
        return math.log1p(math.exp(z))

    def conjugate_has_compact_domain(self):
        return True

    def domain(self):
        return (0, 1)

    def conjugate(self, s):
        def entr(u):
            if u == 0:
                return 0
            else:
                return u * math.log(u)

        return entr(s) + entr(1 - s)
```

Note that since we're not going to be searching for an initial interval, we don't need to implement the methods returning upper bound and lower bound sequences.

A.3 The `Hinge` class

For $h(z) = \max(0, z)$, looking at Table 1, we see that $h^*(s)$ is just the indicator of the interval $[0, 1]$, and we again fall into the compact domain case. Below is the implementation:

```
import math

class Hinge:
    def eval(self, z):
        return max(0, z)

    def conjugate_has_compact_domain(self):
        return True

    def domain(self):
        return (0, 1)

    def conjugate(self, s):
        if s < 0 or s > 1:
            return math.inf
        else:
            return 0
```

A.4 The `L2Reg` class

First, note that for all regularizers we need to be able to compute their Moreau envelope and their proximal operator. Hence, we first define a common base class:

```
from abc import ABC, abstractmethod

class Regularizer(ABC):
    @abstractmethod
    def prox(self, eta, x):
        pass

    @abstractmethod
    def eval(self, x):
        pass

    def envelope(self, eta, x):
        prox = self.prox(eta, x)
        result = self.eval(prox) + 0.5 * (prox - x).square().sum() / eta
        return result.item()
```

Now we can use it to implement our `L2Reg` class, which represents $r(\mathbf{x}) = \frac{\mu}{2}\|\mathbf{x}\|_2^2$, using the proximal operator in Table 2.

```
class L2Reg(Regularizer):
    def __init__(self, mu):
        self._mu = mu

    def prox(self, eta, x):
```

```
        return x / (1 + self._mu * eta)

    def eval(self, x):
        return self._mu * x.square().sum() / 2.
```

A.5 The `L1Reg` class

Using the `Regularizer` base class above, and on Table 2, we can also implement the `L1Reg` class for representing $r(\mathbf{x}) = \mu\|\mathbf{x}\|_1$.

```
from torch.nn.functional import softshrink

class L1Reg(Regularizer):
    def __init__(self, mu):
        self._mu = mu

    def prox(self, eta, x):
        softshrink(x, eta * self._mu)

    def eval(self, x):
        return self._mu * x.abs().sum()
```

A.6 The `L2NormReg` class

The following class represents the $r(\mathbf{x}) = \mu\|\mathbf{x}\|_2$ class.

```
from torch.linalg import norm

class L2NormReg(Regularizer):
    def __init__(self, mu):
        self._mu = mu

    def prox(self, eta, x):
        nrm = norm(x)
        eta = eta * self._mu
        return (1 - eta / max(eta, nrm)) * x

    def eval(self, x):
        return self._mu * norm(x)
```

B Code for mini-batch of convex onto linear compositions

B.1 The `HalfSquared` class

For $h(z) = \frac{1}{2}z^2$, we have $h^*(s) = \frac{1}{2}s^2$. Hence, our dual problem is of the form

$$\begin{aligned} q(\mathbf{s}) &= -\frac{1}{2}\|\mathbf{P}\mathbf{s}\|_2^2 + \mathbf{c}^T\mathbf{s} - \frac{m}{2}\|\mathbf{s}\|_2^2 \\ &= -\frac{1}{2}\mathbf{s}^T\left(\mathbf{P}^T\mathbf{P} + m\mathbf{I}\right)\mathbf{s} + \mathbf{c}^T\mathbf{s}, \end{aligned}$$

where $\mathbf{I}$ is the identity matrix of the appropriate size. It's a simple strictly concave quadratic function, which can be minimized by equating its gradient with zero:

$$\nabla q(\mathbf{s}) = -(\mathbf{P}^T\mathbf{P} + m\mathbf{I})\mathbf{s} + \mathbf{c} = 0.$$

Re-arranging the above equation leads to the maximizer

$$\mathbf{s} = (\mathbf{P}^T\mathbf{P} + m\mathbf{I})^{-1}\mathbf{c}.$$

It's also easy to see that $\mathbf{P}^T\mathbf{P}+m\mathbf{I}$ is a symmetric positive-definite matrix, and thus $\mathbf{s}$ can be obtained using the well-known Cholesky decomposition. Fortunately, `PyTorch` has all the necessary machinery to do exactly that.

```
class HalfSquared:
    def solve_dual(self, P, c):
        m = P.shape[1]  # number of columns = batch size

        # construct lhs matrix P* P + m I
        lhs_mat = torch.mm(P.t(), P)
        lhs_mat.diagonal().add_(m)

        # solve positive-definite linear system using Cholesky factorization
        lhs_factor = torch.cholesky(lhs_mat)
        rhs_col = c.unsqueeze(1)  # make rhs a column vector, so that cholesky_solve works
        return torch.cholesky_solve(rhs_col, lhs_factor)

    def eval(self, lin):
        return 0.5 * (lin ** 2)
```

B.2 The `Logistic` class

In direct contrast to the case of the half-squared function, for $h(z) = \ln(1 + \exp(z))$ with $h^*(s) = s\ln(s) + (1 - s)\ln(1 - s)$ we don't have a formula for computing a maximizer s^*. However, convex optimization is a mature technology, and a variety of extremely fast and efficient software packages exists to do exactly that - minimize convex functions, or equivalently, maximize concave functions. In this paper we'll use one such package, `CVXPY` [1, 18], which in fact is an interface to a variety of lower-level convex optimization solvers.

```
import torch
import cvxpy as cp

class Logistic:
    def solve_dual(self, P, c):
        # extract information and convert tensors to numpy. CVXPY
        # works with numpy arrays
        dtype = P.dtype
        m = P.shape[1]
        P = P.data.numpy()
        c = c.data.numpy()

        # define the dual optimization problem using CVXPY
        s = cp.Variable(m)
        objective = 0.5 * cp.sum_squares(P @ s) - \
```

```
            cp.sum(cp.multiply(c, s)) - \
            (cp.sum(cp.entr(m * s)) + cp.sum(cp.entr(1 - m * s))) / m

        prob = cp.Problem(cp.Minimize(objective))

        # solve the problem, and extract the optimal solution
        prob.solve()

        # recover optimal solution, and ensure it's cast to the same type as
        # the input data.
        return torch.tensor(s.value).to(dtype=dtype).unsqueeze(1)

    def eval(self, lin):
        return torch.log1p(torch.exp(lin))
```

B.3 The `Hinge` class

As was the case with the `Logistic` class, there is no closed-form solution for solving the dual. The conjugate of $h(z) = \max(0, z)$ is the indicator of the interval $[0, 1]$, and thus the dual problem aims to solve

$$\max_{\mathbf{s}} \quad \frac{1}{2}\|\mathbf{P}\mathbf{s}\|_2^2 + \mathbf{c}^T\mathbf{s} \quad \text{s.t.} \quad 0 \le s_i \le \frac{1}{m}$$

The corresponding Python code using CVXPY is below.

```
import torch
import torch.nn.functional
import cvxpy as cp

class Hinge:
    def solve_dual(self, P, c):
        # extract information and convert tensors to numpy. CVXPY
        # works with numpy arrays
        dtype = P.dtype
        m = P.shape[1]
        P = P.data.numpy()
        c = c.data.numpy()

        # define the dual optimization problem using CVXPY
        s = cp.Variable(m)
        objective = 0.5 * cp.sum_squares(P @ s) - cp.sum(cp.multiply(c , s))

        constraints = [s >= 0, s <= 1. / m]
        prob = cp.Problem(cp.Minimize(objective), constraints)

        # solve the problem, and extract the optimal solution
        prob.solve()
        return torch.tensor(s.value).to(dtype=dtype).unsqueeze(1)

    def eval(self, lin):
        return torch.nn.functional.relu(lin)
```

Funding The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Data availability Data sharing not applicable to this article as no datasets were generated or analysed during the current study. All the plots were created using a simulation, and are reproducible using the code repository accompanying this paper.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Agrawal, A., Verschueren, R., Diamond, S., Boyd, S.: A rewriting system for convex optimization problems. J. Control Dec. **5**(1), 42–60 (2018)
2. Asi, H., Chadha, K., Cheng, G., Duchi, J.C.: Minibatch stochastic approximate proximal point methods. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 21958–21968. Curran Associates Inc, New York (2020)
3. Asi, H., Duchi, J.C.: The importance of better models in stochastic optimization. Proc. Natl. Acad. Sci. **116**(46), 22924–22930 (2019). https://doi.org/10.1073/pnas.1908018116.
4. Asi, H., Duchi, J.C.: Modeling simple structures and geometry for better stochastic optimization algorithms. In: Chaudhuri, K., Sugiyama, M. (eds.) Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research, vol. 89, pp. 2425–2434. PMLR (2019). http://proceedings.mlr.press/v89/asi19a.html
5. Asi, H., Duchi, J.C.: Stochastic (approximate) proximal point methods: convergence, optimality, and adaptivity. SIAM J. Optim. **29**(3), 2257–2290 (2019). https://doi.org/10.1137/18M1230323
6. Bauschke, H.H., Goebel, R., Lucet, Y., Wang, X.: The proximal average: basic theory. SIAM J. Optim. **19**(2), 766–785 (2008). https://doi.org/10.1137/070687542
7. Beck, A.: First-Order Methods in Optimization. Society for Industrial and Applied Mathematics, Philadelphia (2017). https://doi.org/10.1137/1.9781611974997
8. Beck, A., Teboulle, M.: A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM J. Imag. Sci. **2**(1), 183–202 (2009)
9. Bertsekas, D.P.: Incremental proximal methods for large scale convex optimization. Math. Program. **129**(2), 163–195 (2011)
10. Brent, R.P.: Algorithms for Minimization Without Derivatives. Courier Corporation, Chelmsford (2013)
11. Campolongo, N., Orabona, F.: Temporal variability in implicit online learning. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 12377–12387. Curran Associates Inc, New York (2020)
12. Chambolle, A., Pock, T.: A first-order primal-dual algorithm for convex problems with applications to imaging. J. Math. Imaging Vis. **40**(1), 120–145 (2011). https://doi.org/10.1007/s10851-010-0251-1
13. Chambolle, A., Pock, T.: On the ergodic convergence rates of a first-order primal-dual algorithm. Math. Program. **159**(1), 253–287 (2016). https://doi.org/10.1007/s10107-015-0957-3
14. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
15. Davis, D., Drusvyatskiy, D.: Stochastic model-based minimization of weakly convex functions. SIAM J. Optim. **29**(1), 207–239 (2019). https://doi.org/10.1137/18M1178244
16. Davis, D., Drusvyatskiy, D., MacPhee, K.J.: Stochastic model-based minimization under high-order growth (2018). https://arxiv.org/abs/1807.00255
17. Defazio, A.: A simple practical accelerated method for finite sums. In: Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 29. Curran Associates, Inc., New York (2016)
18. Diamond, S., Boyd, S.: CVXPY: a python-embedded modeling language for convex optimization. J. Mach. Learn. Res. **17**(83), 1–5 (2016)
19. Drusvyatskiy, D., Paquette, C.: Efficiency of minimizing compositions of convex functions and smooth maps. Math. Program. **178**(1), 503–558 (2019). https://doi.org/10.1007/s10107-018-1311-3
20. Duchi, J.C., Ruan, F.: Stochastic methods for composite and weakly convex optimization problems. SIAM J. Optim. **28**(4), 3229–3259 (2018). https://doi.org/10.1137/17M1135086
21. Kalai, A., Vempala, S.: Efficient algorithms for online decision problems. J. Comput. Syst. Sci. **71**(3), 291–307 (2005). https://doi.org/10.1016/j.jcss.2004.10.016
22. Karampatziakis, N., Langford, J.: Online importance weight aware updates . UAI'11, pp. 392–399. AUAI Press, Arlington, Virginia (2011)

23. Kivinen, J., Warmuth, M.K.: Exponentiated gradient versus gradient descent for linear predictors. Inf. Comput. **132**(1), 1–63 (1997). https://doi.org/10.1006/inco.1996.2612
24. Kulis, B., Bartlett, P.L.: Implicit online learning. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 575–582 (2010)
25. Liu, J., Xu, L., Shen, S., Ling, Q.: An accelerated variance reducing stochastic method with Douglas–Rachford splitting. Mach. Learn. **108**(5), 859–878 (2019). https://doi.org/10.1007/s10994-019-05785-3
26. Martinet, B.: Brève communication. régularisation d'inéquations variationnelles par approximations successives. Revue Française d'informatique et de recherche Opérationnelle Série Rouge **4**(R3), 154–158 (1970)
27. Moreau, J.J.: Fonctions convexes duales et points proximaux dans un espace Hilbertien. C. R. Hebd. Seances Acad. Sci. **255**, 2897–2899 (1962)
28. Moreau, J.J.: Proximité et dualité dans un espace hilbertien. Bull. Soc. Math. France **93**, 273–299 (1965). https://doi.org/10.24033/bsmf.1625
29. Parikh, N., Boyd, S.: Proximal algorithms. Found. Trends Optim. **1**(3), 127–239 (2014). https://doi.org/10.1561/2400000003
30. Passty, G.B.: Ergodic convergence to a zero of the sum of monotone operators in Hilbert space. J. Math. Anal. Appl. **72**(2), 383–390 (1979). https://doi.org/10.1016/0022-247X(79)90234-8
31. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat. **22**(3), 400–407 (1951)
32. Rockafellar, R.T.: Convex Analysis. Princeton University Press, Princeton (1970)
33. Ryu, E.K., Boyd, S.: Stochastic proximal iteration: a non-asymptotic improvement upon stochastic gradient descent. Author website, early draft (2014). https://web.stanford.edu/~boyd/papers/pdf/spi.pdf
34. Shtoff, A.: alexshtf/inc_prox_pt: [Code] Efficient implementation of incremental proximal point methods. Zenodo (2024). https://doi.org/10.5281/zenodo.11500107
35. Tran-Dinh, Q., Pham, N., Nguyen, L.: Stochastic Gauss-Newton algorithms for nonconvex compositional optimization. In: III, H.D., Singh, A. (eds.) Proceedings of the 37th International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 119, pp. 9572–9582. PMLR (2020). http://proceedings.mlr.press/v119/tran-dinh20a.html
36. Yu, Y.L.: Better approximation and faster algorithm using the proximal average. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. (eds.) Advances in Neural Information Processing Systems, vol. 26. Curran Associates Inc, New York (2013)
37. Zhou, K., So, A.M.C., Cheng, J.: Boosting first-order methods by shifting objective: new schemes with faster worst-case rates. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 15405–15416. Curran Associates Inc, New York (2020)
38. Zinkevich, M.: Online convex programming and generalized infinitesimal gradient ascent. In: Proceedings of the 20th International Conference on International Conference on Machine Learning, ICML'03, pp. 928–935. AAAI Press (2003)