Benchopt: Reproducible, efficient and collaborative optimization benchmarks

Thomas Moreau, Mathurin Massias, Alexandre Gramfort, Pierre Ablin, Pierre-Antoine Bannier, Benjamin Charlier, Mathieu Dagréou, Tom Dupré La Tour, Ghislain Durif, Cassio F Dantas, et al.

To cite this version:


HAL Id: hal-03830604
https://hal.archives-ouvertes.fr/hal-03830604
Submitted on 26 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Benchopt: Reproducible, efficient and collaborative optimization benchmarks

Thomas Moreau¹,*, Mathurin Massias²,*, Alexandre Gramfort¹,*, Pierre Ablin³,
Pierre-Antoine Bannier, Benjamin Charlier¹, Mathieu Dagréou¹, Tom Dupré la Tour⁶,
Ghislain Durif⁴, Cassio F. Dantas⁷, Quentin Klopfenstein⁸, Johan Larsson⁹, En Lai¹,
Tanguy Lefort⁴, Benoit Malézieux¹, Badr Moufad², Binh T. Nguyen¹⁰, Alain Rakotomamonjy¹¹,
Zaccharie Ramzi¹², Joseph Salmon⁴,⁵, Samuel Vaiter¹³

¹ Université Paris-Saclay, Inria, CEA, 91120 Palaiseau, France
² Univ Lyon, Inria, CNRS, ENS de Lyon, UCB Lyon 1, LIP UMR 5668, F-69342, Lyon, France
³ Université Paris-Dauphine, PSL University, CNRS, 75016, Paris, France
⁴ IMAG, Univ Montpellier, CNRS, Montpellier, France
⁵ Institut Universitaire de France (IUF)
⁶ University of California, Berkeley, CA 94720, USA
⁷ TETIS, Univ Montpellier, INRAE, Montpellier, France
⁸ University of Luxembourg, LCSB, Esch-sur-Alzette, Luxembourg
⁹ The Department of Statistics, Lund University
¹⁰ LTCI, Télécom Paris, 91120 Palaiseau, France
¹¹ Criteo AI Lab, Paris, France
¹² ENS Ulm, CNRS, UMR 8553, Paris, France
¹³ CNRS & Université Côte d’Azur, Laboratoire J.A. Dieudonné, CNRS, Nice, France

Abstract

Numerical validation is at the core of machine learning research as it allows to assess the actual impact of new methods, and to confirm the agreement between theory and practice. Yet, the rapid development of the field poses several challenges: researchers are confronted with a profusion of methods to compare, limited transparency and consensus on best practices, as well as tedious re-implementation work. As a result, validation is often very partial, which can lead to wrong conclusions that slow down the progress of research. We propose Benchopt, a collaborative framework to automate, reproduce and publish optimization benchmarks in machine learning across programming languages and hardware architectures. Benchopt simplifies benchmarking for the community by providing an off-the-shelf tool for running, sharing and extending experiments. To demonstrate its broad usability, we showcase benchmarks on three standard learning tasks: $\ell_2$-regularized logistic regression, Lasso, and ResNet18 training for image classification. These benchmarks highlight key practical findings that give a more nuanced view of the state-of-the-art for these problems, showing that for practical evaluation, the devil is in the details. We hope that Benchopt will foster collaborative work in the community hence improving the reproducibility of research findings.

1 Introduction

Numerical experiments have become an essential part of statistics and machine learning (ML). It is now commonly accepted that every new method needs to be validated through comparisons with existing approaches on standard problems. Such validation provides insight into the method’s benefits and limitations and thus adds depth to the results. While research aims at advancing knowledge and not just improving the state of the art, experiments ensure that results are reliable and support theoretical claims (Sculley et al., 2018). Practical validation also helps the ever-increasing number of ML users in applied sciences to choose the right method for their task. Performing rigorous and extensive experiments is, however, time-consuming (Raff, 2019), particularly because comparisons
Figure 1: A visual summary of Benchopt. Each Solver is run (in parallel) on each Dataset and each variant of the Objective. Results are exported as a CSV file that is easily shared and can be automatically plotted as interactive HTML visualizations or PDF figures.

against existing methods in new settings often requires reimplementing baseline methods from the literature. In addition, ingredients necessary for a proper reimplementation may be missing, such as important algorithmic details, hyperparameter choices, and preprocessing steps (Pineau et al., 2019).

In the past years, the ML community has actively sought to overcome this “reproducibility crisis” (Hutson, 2018) through collaborative initiatives such as open datasets (OpenML, Vanschoren et al. 2013), standardized code sharing (Forde et al., 2018), benchmarks (MLPerf, Mattson et al. 2020), the NeurIPS and ICLR reproducibility challenges (Pineau et al., 2019; Pineau et al., 2021) and new journals (e.g., Rougier and Hinsen 2018). As useful as these endeavors may be, they do not, however, fully address the problems in optimization for ML since, in this area, there are no clear community guidelines on how to perform, share, and publish benchmarks.

Optimization algorithms pervade almost every area of ML, from empirical risk minimization, variational inference to reinforcement learning (Sra et al., 2012). It is thus crucial to know which methods to use depending on the task and setting (Bartz-Beielstein et al., 2020). While some papers in optimization for ML provide extensive validations (Lueckmann et al., 2021), many others fall short in this regard due to lack of time and resources, and in turn feature results that are hard to reproduce by other researchers. In addition, both performance and hardware evolve over time, which eventually makes static benchmarks obsolete. An illustration of this is the recent work by Schmidt et al. (2021), which extensively evaluates the performances of 15 optimizers across 8 deep-learning tasks. While their benchmark gives an overall assessment of the considered solvers, this assessment is bound to become out-of-date if it is not updated with new solvers and new architectures. Moreover, the benchmark does not reproduce state-of-the-art results on the different datasets, potentially indicating that the considered architectures and optimizers could be improved.

We firmly believe that this critical task of maintaining an up-to-date benchmark in a field cannot be solved without a collective effort. We want to empower the community to take up this challenge and build a living, reproducible and standardized state of the art that can serve as a foundation for future research.

Benchopt provides the tools to structure the optimization for machine learning (Opt-ML) community around standardized benchmarks, and to aggregate individual efforts for reproducibility and results sharing. Benchopt can handle algorithms written in Python, R, Julia or C/C++ via binaries. It offers built-in functionalities to ease the execution of benchmarks: parallel runs, caching, and automatical results archiving. Benchmarks are meant to evolve over time, which is why Benchopt offers a modular structure through which a benchmark can be easily extended with new objective functions, datasets, and solvers by the addition of a single file of code.

The paper is organized as follows. We first detail the design and usage of Benchopt, before presenting results on three canonical problems:

- $\ell_2$-regularized logistic regression: a convex and smooth problem which is central to the evaluation of many algorithms in the Opt-ML community, and remains of high relevance for practitioners;
- the Lasso: the prototypical example of non-smooth convex problem in ML;
- training of ResNet18 architecture for image classification: a large scale non-convex deep learning problem central in the field of computer vision.
The reported benchmarks, involving dozens of implementations and datasets, shed light on the current state-of-the-art solvers for each problem, across various settings, highlighting that the best algorithm largely depends on the dataset properties (e.g., size, sparsity), the hyperparameters, as well as hardware. A variety of other benchmarks (e.g., MCP, TV1D, etc.) are also presented in Appendix, with the goal to facilitate contributions from the community.

By the open source and collaborative design of Benchopt (BSD 3-clause license), we aim to open the way towards community-endorsed and peer-reviewed benchmarks that will improve the tracking of progress in optimization for ML.

2 The Benchopt library

The Benchopt library aims to provide a standard toolset and structure to implement benchmarks for optimization in ML, where the problems depend on some input dataset \( D \). The considered problems are of the form

\[
\theta^* \in \arg \min_{\theta \in \Theta} f(\theta; D, \Lambda),
\]

where \( f \) is the objective function, \( \Lambda \) are its hyperparameters, and \( \Theta \) is the feasible set for \( \theta \). The scope of the library is to evaluate optimization methods in their wide sense by considering the sequence \( \{\theta^t\} \) produced to approximate \( \theta^* \). We emphasize than Benchopt does not provide a fixed set of benchmarks, but a framework to create, extend and share benchmarks on any problem of the form (1). To provide a flexible and extendable coding standard, benchmarks are defined as the association of three types of object classes:

**Objective**: It defines the function \( f \) to be minimized as well as the hyperparameters \( \Lambda \) or the set \( \Theta \), and the metrics to track along the iterations (e.g., objective value, gradient norm for smooth problems, or validation loss). Multiple metrics can be registered for each \( \theta^t \).

**Datasets**: The Dataset objects provide the data \( D \) to be passed to the Objective class. They control how data is loaded and preprocessed. Datasets are separated from the Objective, making it easy to add new ones, provided they are coherent with the Objective.

**Solvers**: The Solver objects define how to run the algorithm. They are provided with the Objective and Dataset objects and output a sequence \( \{\theta^t\} \). This sequence can be obtained using a single run of the method, or with multiple runs in case the method only returns its final iterate.

Each of these objects can have parameters that change their behavior, e.g., the regularization parameters for the Objective, the choice of preprocessing for the Datasets, or the step size for the Solvers. By exposing these parameters in the different objects, Benchopt can evaluate their influence on the benchmark results. The Benchopt library defines an application programming interface (API) for each of these concepts and provides a command line interface (CLI) to make them work together. A benchmark is defined as a folder that contains an Objective as well as subfolders containing the Solvers and Datasets. Appendix B presents a concrete example on Ridge regression of how to construct a Benchopt benchmark while additional design design choices of Benchopt are discussed in Appendix C.

For each Dataset and Solver, and for each set of parameters, Benchopt retrieves a sequence \( \{\theta^t\} \), and evaluates the metrics defined in the Objective for each \( \theta^t \). To ensure fair evaluation, the computation of these metrics is done off-line. The metrics are gathered in a CSV file that can be used to display the benchmark results, either locally or as HTML files published on a website that reference the benchmarks run with Benchopt. This workflow is described in Figure 1.

This modular and standardized organization for benchmarks empowers the optimization community by making numerical experiments easily reproducible, shareable, flexible and extendable. The benchmark can be shared as a git repository or a folder containing the different definitions for the Objective, Datasets and Solvers and it can be run with the Benchopt CLI, hence becoming a convenient reference for future comparisons. This ensures fair evaluation of baselines in follow-up experiments, as implementations validated by the community are available. Moreover, benchmarks can be extended easily as one can add a Dataset or a Solver to the comparison by adding a single
Finally, by supporting multiple metrics — e.g., training and testing losses, error on parameter estimates, sparsity of the estimate — the Objective class offers the flexibility to define the concurrent evaluation, which can be extended to track extra metrics on a per benchmark basis, depending on the problem at hand.

As one of the goal of Benchopt is to make benchmarks as simple as possible, it also provides a set of features to make them easy to develop and run. Benchopt is written in Python, but Solvers run with implementations in different languages (e.g., R and Julia, as in Section 4) and frameworks (e.g., PyTorch and TensorFlow, as in Section 5). Moreover, benchmarks can be run in parallel with checkpointing of the results, enabling large scale evaluations on many CPU or GPU nodes. Benchopt also makes it possible to run solvers with many different hyperparameters’ values, allowing to assess their sensitivity on the method performance. Benchmark results are also automatically exported as interactive visualizations, helping with the exploration of the many different settings.

Benchmarks All presented benchmarks are run on 10 cores of an Intel Xeon Gold 6248 CPUs @ 2.50GHz and NVIDIA V100 GPUs (16GB). The results’ interactive plots and data are available at https://benchopt.github.io/results/preprint_results.html.

3 First example: $\ell_2$-regularized logistic regression

Logistic regression is a very popular method for binary classification. From a design matrix $X \in \mathbb{R}^{n \times p}$ with rows $X_i$ and a vector of labels $y \in \{-1, 1\}^n$ with corresponding element $y_i$, $\ell_2$-regularized logistic regression provides a generalized linear model indexed by $\theta^* \in \mathbb{R}^p$ to discriminate the classes by solving

$$
\theta^* = \arg \min_{\theta \in \mathbb{R}^p} \sum_{i=1}^{n} \log \left(1 + \exp(\theta^T X_i)\right) + \frac{\lambda}{2} \|	heta\|_2^2.
$$

where $\lambda > 0$ is the regularization hyperparameter. Thanks to the regularization part, Problem (2) is strongly convex with a Lipschitz gradient, and thus its solution can be estimated efficiently using many iterative optimization schemes.

The most classical methods to solve this problem take inspiration from Newton’s method (Wright and Nocedal, 1999). On the one hand, quasi-Newton methods aim at approximating the Hessian of the cost function with cheap to compute operators. Among these methods, L-BFGS (Liu and Nocedal, 1989) stands out for its small memory footprint, its robustness and fast convergence in a variety of settings. On the other hand, truncated Newton methods (Dembo et al., 1982) try to directly approximate Newton’s direction by using e.g., the conjugate gradient method (Fletcher and Reeves, 1964) and Hessian-vector products to solve the associated linear system. Yet, these methods suffer when $n$ is large: each iteration requires a pass on the whole dataset.

In this context, methods based on stochastic estimates of the gradient have become standard (Bottou, 2010), with Stochastic Gradient Descent (SGD) as a main instance. The core idea is to use cheap and noisy estimates of the gradient (Robbins and Monro, 1951; Kiefer and Wolfowitz, 1952). While SGD generally converges either slowly due to decreasing step sizes, or to a neighborhood of the solution for constant step sizes, variance-reduced adaptations such as SAG (Schmidt et al., 2017), SAGA (Defazio et al., 2014) and SVRG (Johnson and Zhang, 2013) make it possible to solve the problem more efficiently and are often considered to be state-of-the-art for large scale problems.

Finally, methods based on coordinate descent (Bertsekas, 1999) have also been proposed to solve Problem (2). While these methods are usually less popular, they can be efficient in the context of sparse datasets, where only few samples have non-zero values for a given feature, or when accelerated on distributed systems or GPU (Dünner et al., 2018).

The code for the benchmark is available at https://github.com/benchopt/benchmark_logreg_l2/. To reflect the diversity of solvers available, we showcase a Benchopt benchmark with 3 datasets, 10 optimization strategies implemented in 5 packages, leveraging GPU hardware when possible. We also consider different scenarios for the objective function: (i) scaling (or not) the features, a recommended data preprocessing step, crucial in practice to have comparable regularization strength on all variables; (ii) fitting (or not) an unregularized intercept term, important in practice and making optimization harder when omitted from the regularization term (Koh et al., 2007); (iii) working (or
Figure 3: Benchmark for the $\ell_2$-regularized logistic regression, on 13 solvers, 4 datasets (rows), and 3 variants of the Objective (columns) with $\lambda = 1$. The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time. The first column corresponds to the objective function detailed in Problem (2). In the second column, datasets were preprocessed by normalizing each feature to unit standard deviation. The third column is for an objective function which includes an unregularized intercept.

Results Figure D.1 presents the results of the benchmarks, in terms of suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, for three datasets and three scenarios. Here, because the problem is convex, $\theta^*$ is approximated by the best iterate across all runs (see Section C.1). Overall, the benchmark shows the benefit of using GPU solvers (cuML and snapML), even for small scale tasks such as ijcnn1. Note that these two accelerated solvers converge to a higher suboptimality level compared to other solvers, due to operating with 32-bit float precision. Another observation is that data scaling can drastically change the picture. In the case of madelon, most solvers have a hard time converging for the scaled data. For the solvers that converge, we note that the convergence time is one order of magnitude smaller with the scaled dataset compared to the raw one. This stems from the fact that in this case, the scaling improves the conditioning of the dataset.¹ For news20.binary, the stochastic solvers such as SAG and SAGA have degraded performances on scaled data. Here, the scaling makes the problem harder.²

¹The condition number of the dataset is divided by 5.9 after scaling.
²The condition number is multiplied by 407 after scaling.
On CPU, quasi-Newton solvers are often the most efficient ones, and provide a reasonable choice in most situations. For large scale news20.binary, stochastic solvers such as SAG, SAGA or SVRG—which are often considered as state of the art for such problem—have worst performances for the presented datasets. While this dataset is often used as a test bed for benchmarking new stochastic solvers, we fail to see an improvement over non-stochastic ones for this experimental setup. In contrast, the last row in Figure D.1 displays an experiment with the larger scale criteo dataset, which demonstrates a regime where variance-reduced stochastic gradient methods outperform quasi-Newton methods. For future benchmarking of stochastic solvers, we therefore recommend using such a large dataset.

Finally, the third column in Figure D.1 illustrates a classical problem when benchmarking different solvers: their specific (and incompatible) definition and resolution of the corresponding optimization problem. Here, the objective function is modified to account for an intercept (bias) in the linear model. In most situations, this intercept is not regularized when it is fitted. However, snapML and liblinear solvers do regularize it, leading to incomparable losses.

4 Second example: The Lasso

The Lasso, (Tibshirani, 1996; Chen et al., 1998), is an archetype of non-smooth ML problems, whose impact on ML, statistics and signal processing in the last three decades has been considerable (Bühlmann and van de Geer, 2011; Hastie et al., 2015). It consists of solving

$$\theta^* \in \arg \min_{\theta \in \mathbb{R}^p} \frac{1}{2} \|y - X\theta\|_2^2 + \lambda \|\theta\|_1,$$

where $X \in \mathbb{R}^{n \times p}$ is a design matrix containing $p$ features as columns, $y \in \mathbb{R}^n$ is the target vector, and $\lambda > 0$ is a regularization hyperparameter. The Lasso estimator was popularized for variable selection: when $\lambda$ is high enough, many entries in $\theta^*$ are exactly equal to 0. This leads to more interpretable models and reduces overfitting compared to the least-squares estimator.

Solvers for Problem (3) have evolved since its introduction by Tibshirani (1996). After generic quadratic program solvers, new dedicated solvers were proposed based on iterative reweighted least-squares (IRLS) (Grandvalet, 1998), followed by LARS (Efron et al., 2004), a homotopy method computing the full Lasso path. The LARS solver helped popularize the Lasso, yet the algorithm suffers from stability issues and can be very slow for worst case situations (Mairal and Yu, 2012). General purpose solvers became popular for Lasso-type problems with the introduction of the iterative soft-thresholding algorithm (ISTA, Daubechies et al. 2004), an instance of forward-backward splitting (Combettes and Wajs, 2005). These algorithms became standard in signal and image processing, especially when accelerated (FISTA, Beck and Teboulle 2009).

In parallel, proximal coordinate descent has proven particularly relevant for the Lasso in statistics. Early theoretical results were proved by Tseng (1993) and Sardy et al. (2000), before it became the standard solver of the widely distributed packages glmnet in R and scikit-learn in Python. For further improvements, some solvers exploit the sparsity of $\theta^*$, trying to identify its support to reduce the problem size. Best performing variants of this scheme are screening rules (e.g., El Ghaoui et al., 2012; Bonnefoy et al., 2015; Ndiaye et al., 2017) and working/active sets (e.g., Johnson and Guestrin 2015; Massias et al. 2018), including strong rules (Tibshirani et al., 2012).

While reviews of Lasso solvers have already been performed (Bach et al., 2012, Sec. 8.1), they are limited to certain implementation and design choices, but also naturally lack comparisons with more recent solvers and modern hardware, hence drawing biased conclusions.

The code for the benchmark is available at https://github.com/benchopt/benchmark_lasso/. Results obtained on 4 datasets, with 9 standard packages and some custom reimplementations, possibly leveraging GPU hardware, and 17 different solvers written in Python/numba/Cython, R, Julia or C++ (Table E.1) are presented in Figure 4. All solvers use efficient numerical implementations, possibly leveraging calls to BLAS, precompiled code in Cython or just-in-time compilation with numba.

The different parameters influencing the setup are

- the regularization strength $\lambda$, controlling the sparsity of the solution, parameterized as a fraction of $\lambda = \|X^\top y\|_\infty$ (the minimal hyperparameter such that $\theta^* = 0$).

---

3The Lasso path is the set of solutions of Problem (3) as $\lambda$ varies in $(0, \infty)$. 
• the dataset dimensions: MEG has small $n$ and medium $p$; rcv1.binary has medium $n$ and $p$; news20.binary has medium $n$ and very large $p$ while MillionSong has very large $n$ and small $p$ (Table E.2).

Figure 4: Benchmark for the Lasso, on 17 solvers, 4 datasets (rows), and 3 variants of the objective (columns) with decreasing regularization $\lambda$. The curves display the suboptimality of the objective function, $f(\theta^t) - f(\theta^*)$, as a function of time.

**Results** Figure 4 presents the result of the benchmark on the Lasso, in terms of objective suboptimality $f(\theta^t) - f(\theta^*)$ as a function of time.

Similarly to Section 3, the GPU solvers obtain good performances in most settings, but their advantage is less clear. A consistent finding across all settings is that coordinate descent-based methods outperform full gradient ones (ISTA and FISTA, even restarted), and are improved by the use of working set strategies (blitz, celer, skglm, glmnet). This observation is even more pronounced when the regularization parameter is large, as the solution is sparser.

When observing the influence of the dataset dimensions, we observe 3 regimes. When $n$ is small (MEG), the support of the solution is small and coordinate descent, LARS and noncvx-pro perform the best. When $n$ is much larger than $p$ (MillionSong), noncvx-pro clearly outperforms other solvers, and working set methods prove useless. Finally, when $n$ and $p$ are large (rcv1.binary, news20.binary), CD and working sets vastly outperforms the rest while noncvx-pro fails, as it requires solving a linear system of size $\min(n, p)$. We note that this setting was not tested in the original experiment of Poon and Peyré (2021), which highlights the need for extensive, standard experimental setups.
When the support of the solution is small (either small $\lambda$, either small $n$ since the Lasso solution has at most $n$ nonzero coefficients), LARS is a competitive algorithm. We expect this to degrade when $n$ increases, but as the LARS solver in scikit-learn does not support sparse design matrices we could not include it for news20.binary and rcv1.binary.

This benchmark is the first to evaluate solvers across languages, showing the competitive behavior of lasso.jl and glmnet compared to Python solvers. Both solvers have a large initialization time, and then converge very fast. To ensure that the benchmark is fair, even though the Benchopt library is implemented in Python, we made sure to ignore conversion overhead, as well as just-in-time compilation cost. We also checked the timing’s consistency with native calls to the libraries.

Since the Lasso is massively used for its feature selection properties, the speed at which the solvers identify the support of the solution is also an important performance measure. Monitoring this with Benchopt is straightforward, and a figure reporting this benchmark is in Appendix E.

5 Third example: How standard is a benchmark on ResNet18?

As early successes of deep learning have been focused on computer vision tasks (Krizhevsky et al., 2012), image classification has become a de facto standard to validate novel methods in the field. Among the different network architectures, ResNets (He et al., 2016) are extensively used in the community as they provide strong and versatile baselines (Xie et al., 2021; Tan and Le, 2019; Dosovitskiy et al., 2021; Brock et al., 2021; Liu et al., 2022). While many papers present results with such model on classical datasets, with sometimes extensive ablation studies (He et al., 2019; Wightman et al., 2021; Bello et al., 2021; Schmidt et al., 2021), the lack of standardized codebase and missing implementation details makes it hard to replicate their results.

The code for the benchmark is available at https://github.com/benchopt/benchmark_resnet_classif/. We provide a cross-dataset –SVHN, Netzer et al. (2011); MNIST, LeCun et al. (2010) and CIFAR-10, Krizhevsky (2009)– and cross-framework –TensorFlow/Keras, Abadi et al. (2015) and Chollet et al. (2015); PyTorch, Paszke et al. (2019)– evaluation of the training strategies for image classification with ResNet18 (see Appendix F for details on architecture and datasets). We train the network by minimizing the cross entropy loss relatively to the weights $\theta$ of the model. Contrary to logistic regression and the Lasso, this problem is non-convex due to the non-linearity of the model $f_\theta$.

Another notable difference is that we report the evolution of the test error rather than the training loss. Because we chose to monitor the test loss, the Solvers are defined as the combination of an optimization algorithm, its hyperparameters, the learning rate (LR) and weight decay schedules, and the data augmentation strategy. This is in contrast to a case where we would monitor the train loss, and therefore make the LR and weight decay schedules, as well as the data augmentation policy, part of the objective. We focus on 2 standard methods: stochastic gradient descent (SGD) with momentum and Adam (Kingma and Ba, 2015), as well as a more recently published one: Lookahead (Zhang et al., 2019). The LR schedules are chosen among fixed LR, step LR$^4$, and cosine annealing (Loshchilov and Hutter, 2017). We also consider decoupled weight decay for Adam (Loshchilov and Hutter, 2019), and coupled weight decay (i.e., $\ell_2$-regularization) for SGD. Regarding data augmentation, we use random cropping for all datasets and add horizontal flipping only for CIFAR-10, as the digits datasets do not exhibit a mirror symmetry. We detail the remaining hyperparameters in Table F.2, and discuss their selection as well as their sensitivity in Appendix F.

Aligning cross-framework implementations Due to some design choices, components with the same name in the different frameworks do not have the same behavior. For instance, when it comes to applying weight decay, PyTorch’s SGD uses coupled weight decay, while in TensorFlow/Keras weight decay always refers to decoupled weight decay. These two methods lead to significantly different performance and it is not straightforward to apply coupled weight decay in a post-hoc manner in TensorFlow/Keras (see further details in Section F.3). We conducted an extensive effort to align the networks implementation in different frameworks using unit testing to make the conclusions of our benchmarks independent of the chosen framework. We found additional significant differences (reported in Table F.3) in the initialization, the batch normalization, the convolutional layers and the weight decay scaling.

$^4$decreasing the learning rate by a factor 10 at mid-training, and again at 3/4 of the training
**Figure 5: ResNet18 image classification benchmark with PyTorch Solvers.** The best SGD configuration features data augmentation, momentum, cosine learning rate schedule and weight decay. In dashed black is the state of the art for the corresponding datasets with a ResNet18 measured by Zhang et al. (2019) for CIFAR-10, by Zheng et al. (2021) for SVHN with a PreAct ResNet18, by PapersWithCode for MNIST with all networks considered. Off-the-shelf ResNet implementations in TensorFlow/Keras do not support images smaller than $32 \times 32$ and is hence not shown for MNIST. Curves are exponentially smoothed.

**Results** The results of the benchmark are reported in Figure 5. Each graph reports the test error relative to time, with an ablation study on the solvers parameters. Note that we only report selected settings for clarity but that we run every possible combinations.$^5$

Firstly, reaching the state of the art for a vanilla ResNet18 is not straightforward. On the popular website Papers with code it has been so far underestimated. It can achieve 4.45% and 2.65% test error rates on CIFAR-10 and SVHN respectively (compared to 4.73% and 2.95% – for a PreAct ResNet18 – before that). Our ablation study shows that a variety of techniques is required to reach it. The most significant one is an appropriate data augmentation strategy, which lowers the error rate on CIFAR-10 from about 18% to about 8%. The second most important one is weight decay, but it has to be used in combination with a proper LR schedule, as well as momentum. While these techniques are not novel, they are regularly overlooked in baselines, resulting in underestimation of their performance level.

This reproducible benchmark not only allows a researcher to get a clear understanding of how to achieve the best performances for this model and datasets, but also provides a way to reproduce and extend these performances. In particular, we also include in this benchmark the original implementation of Lookahead (Zhang et al., 2019). We confirm that it slightly accelerates the convergence of the Best SGD, even with a cosine LR schedule – a setting that had not been studied in the original paper.

Our benchmark also evaluates the relative computational performances of the different frameworks. We observe that PyTorch-Lightning is significantly slower than the other frameworks we tested, in large part due to their callbacks API. We also notice that our TensorFlow/Keras implementation is significantly slower ($\approx 28\%$) than the PyTorch ones, despite following the best practices and our profiling efforts. Note that we do not imply that TensorFlow is intrinsically slower than PyTorch, but a community effort is needed to ensure that the benchmark performances are framework-agnostic.

A recurrent criticism of such benchmarks is that only the best test error is reported. In Figure 6, we measure the effect of using a train-validation-test split, by keeping a fraction of the training set as a validation set. The splits we use are detailed in Table F.1. Our finding is that the results of the ablation study do not change significantly when using such procedure, even though their validity is reinforced by the use of multiple trainings. Yet, a possible limitation of our findings is that some of the hyperparameters we used for our study, coming from the PyTorch-CIFAR GitHub repository, may have been tuned while looking at the test set.

$^5$The results are available online as a user-friendly interactive HTML file.
6 Conclusion and future work

We have introduced Benchopt, a library that makes it easy to collaboratively develop fair and extensive benchmarks of optimization algorithms, which can then be seamlessly published, reproduced, and extended. In the future, we plan on supporting the creation of new benchmarks, that could become the standards the community builds on. This work is part of a wider effort to improve reproducibility of machine learning results. It aims to contribute to raising the standard of numerical validation for optimization, which is pervasive in the statistics and ML community as well as for the experimental sciences that rely more and more on these tools for research.

7 Acknowledgements

It can not be stressed enough how much the Benchopt library relies on contributions from the community and in particular the Python open source ecosystem. In particular, it could not exist without the libraries mentioned in Appendix A.

This work was granted access to the HPC resources of IDRIS under the allocation 2022-AD011011172R2 and 2022-AD011013570 made by GENCI, which was used to run all the benchmarks. MM also gratefully acknowledges the support of the Centre Blaise Pascal’s IT test platform at ENS de Lyon (Lyon, France) for Machine Learning facilities. The platform operates the SIDUS solution (Quemener and Corvellec, 2013).

TL, CFD and JS contributions were supported by the Chaire IA CaMeLOt (ANR-20-CHIA-0001-01). AG, EL and TM contributions were supported by the Chaire IA ANR BrAIN (ANR-20-CHIA-0016). BMa contributions were supported by a grant from Digitéo France. MD contributions were supported by a public grant overseen by the French National Research Agency (ANR) through the program UDOPIA, project funded by the ANR-20-THIA-0013-01 and DATAIA convergence institute (ANR-17-CONV-0003). BN work was supported by the Télécom Paris’s Chaire DSAIDIS (Data Science & Artificial Intelligence for Digitalized Industry Services). BMo contributions were supported by a grant from the Labex MILYON.

References


Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]

3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] These are specified in Appendix, on a per-benchmark basis.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] Error bars are not reported for clarity, but Benchopt allows this in particular in html versions of the plots that can be found in https://benchopt.github.io/results/preprint_results.html.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes] In the introduction.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]
A Software ecosystem acknowledgement

The command line interface and API use the click, pyyaml and psutil (Rodola, 2016) libraries.

Numerical computations involve numpy (Harris et al., 2020) and scipy (Virtanen et al., 2020). For cross-language processing, we used rpy2 for calling R (R Core Team, 2017) libraries and PyJulia for interfacing with Julia (Bezanson et al., 2017). The benchmark runs extensively use joblib, loky (Moreau et al., 2017) and submitit for parallelization.

The results are stored and processed for visualizations using pandas (McKinney et al., 2010), matplotlib (Hunter, 2007) for static rendering, mako and plotly (Inc., 2015) for interactive web-pages. The participative results website relies partially on pygithub.

Our documentation is generated by multiple sphinx-based (Brandl, 2010) libraries (sphinx-bootstrap-theme, sphinx-click, sphinx-gallery (Nájera et al., 2020) and sphinx-prompt), and also the numpydoc and pillow (Clark, 2015) libraries.
B  A complete Benchmark example: Objective, Dataset and Solver classes for Ridge regression

Here, we provide code examples for a simple benchmark on Ridge regression. The Ridge regression – also called $\ell^2$-regularized least-squares or Tikhonov regression – is a popular method to solve least-square problems in the presence of noisy observations or correlated features. The problem reads:

$$
\min_{\theta} \frac{1}{2} \| y - X\theta \|_2^2 + \frac{\lambda}{2} \| \theta \|_2^2,
$$

where $X \in \mathbb{R}^{n \times p}$ is a design matrix, $y \in \mathbb{R}^n$ is the target vector and $\lambda$ is the regularization parameter. This problem is strongly convex and many methods can be used to solve it. Direct computation of the close form solution $\theta^* = (X^\top X + \lambda I_d)^{-1} X^\top y$ can be obtained using matrix factorization methods such as Cholesky decomposition or the SVD (Press et al., 2007) or iterative linear solver such as Conjugate-Gradient (Liu and Nocedal, 1989). One can also resort on first order methods such as gradient descent, coordinate descent (known as the Gauss-Seidel method in this context), or their stochastic variant.

The code for the benchmark is available at https://github.com/benchopt/benchmark_ridge/. The following code snippets are provided in the documentation as a template for new benchmarks.

B.1  Objective class

The Objective class is the central part of the benchmark, defining the objective function. This class allows us to monitor the quantities of interest along the iterations of the solvers, amongst which the objective function value. An Objective class should define 3 methods:

- set_data(**data): it allows specifying the nature of the data used in the benchmark. The data is passed as a dictionary of Python variables, so no constraint is enforced to what can be passed here.

- compute(theta): it allows evaluating the objective function for a given value of the iterate, here called $\theta$. This method should take only one parameter, the output returned by the Solver. All other parameters should be stored in the class with the set_data method. The compute function should return a float (understood as the objective value) or a dictionary. If a dictionary is returned it should contain a key called value (the objective value) and all other keys should correspond to float values allowing tracking more than one quantity of interest (e.g. train and test errors).

- to_dict(): a method that returns a dictionary to be passed to the set_objective() method of a Solver.

An Objective class needs to inherit from a base class, benchopt.BaseObjective. Below is the implementation of the Ridge regression Objective class.

```python
from benchopt import BaseObjective

class Objective(BaseObjective):
    name = "Ridge regression"
    parameters = {"reg": [0.1, 1, 10]}

def __init__(self, reg=1):
    self.reg = reg

def set_data(self, X, y):
    self.X, self.y = X, y

def compute(self, theta):
    res = self.y - self.X @ theta
    return 0.5 * res @ res + 0.5 * self.reg * theta @ theta

def to_dict(self):
    return dict(X=self.X, y=self.y, reg=self.reg)
```
B.2 Dataset class

A Dataset class defines data to be passed to the Objective. More specifically, a Dataset class should implement one method:

- get_data(): A method outputting a dictionary that can be passed as keyword arguments **data to the set_data method of the Objective.

A Dataset class also needs to inherit from a base class, benchopt.BaseDataset.

If a Dataset requires some packages to function, Benchopt allows listing some requirements. The necessary packages should be available via conda or pip.

Below is an example of a Dataset definition using the libsvmdata library, which exposes datasets from libsvm, such as leukemia, bodyfat and gisette – described in Table B.1.

```python
from benchopt import BaseDataset
from benchopt import safe_import_context

with safe_import_context() as import_ctx:
    from libsvmdata import fetch_libsvm

class Dataset(BaseDataset):
    name = "libsvm"
    install_cmd = "conda"
    requirements = ["libsvmdata"]
    parameters = ["dataset": ["bodyfat", "leukemia", "gisette"]]

    def __init__(self, dataset="bodyfat"):
        self.dataset = dataset

    def get_data(self):
        X, y = fetch_libsvm(self.dataset)
        return dict(X=self.X, y=self.y)
```

B.3 Solver class

A Solver class must define three methods:

- set_objective(**objective_dict): This method will be called with the dictionary objective_dict returned by the method to_dict from the Objective. The goal of this method is to provide all necessary information to the Solver so it can optimize the objective function.

- run(stop_value): This method takes only one parameter that controls the stopping condition of the Solver. Typically this is either a number of iterations n_iter or a tolerance parameter tol. Alternatively, a callback function that will be called at each iteration can be passed. The callback should return False once the computation should stop. The parameter stop_value is controlled by the stopping_strategy, see below for details.

- get_result(): This method returns a variable that can be passed to the compute method from the Objective. This is the output of the Solver.

If a Python Solver requires some packages such as scikit-learn, Benchopt allows listing some requirements. The necessary packages must be available via conda or pip.

Below is a simple Solver example using scikit-learn implementation of Ridge regression with different optimization algorithms.
import BaseSolver
import safe_import_context

# This context allow to manipulate the Solver object even if
# scikit-learn is not installed. It is used in 'benchopt install'.
with safe_import_context() as import_ctx:
    from sklearn.linear_model import Ridge

class Solver(BaseSolver):
    name = "scikit-learn"
    install_cmd = "conda"
    requirements = ["scikit-learn"]
    parameters = {
        "alg": ["svd", "cholesky", "lsqr", "sparse_cg", "saga"],
    }
    def __init__(self, alg="svd"):
        self.alg = alg
    def set_objective(self, X, y, reg=1):
        self.X, self.y = X, y
        self.clf = Ridge(
            fit_intercept=False, alpha=reg, solver=self.alg, tol=1e-10
        )
    def run(self, n_iter):
        self.clf.max_iter = n_iter + 1
        self.clf.fit(self.X, self.y)
    def get_result(self):
        return self.clf.coef_

B.4 Results from the benchmark

Descriptions of datasets  Table B.1 describes the datasets used in this benchmarks.

Table B.1: List of the datasets used in Ridge regression in Appendix B

<table>
<thead>
<tr>
<th>Datasets</th>
<th>References</th>
<th>Samples (n)</th>
<th>Features (p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>leukemia</td>
<td>Golub et al. (1999)</td>
<td>38</td>
<td>7129</td>
</tr>
<tr>
<td>bodyfat</td>
<td>Guyon et al. (2004)</td>
<td>252</td>
<td>8</td>
</tr>
<tr>
<td>gisette</td>
<td>Guyon et al. (2004)</td>
<td>6000</td>
<td>5000</td>
</tr>
</tbody>
</table>

We also run the solvers on the simulated data described below.

Generation process for simulated dataset  We generate a linear regression scenario with decaying correlation for the design matrix, i.e., the ground-truth covariance matrix is a Toeplitz matrix, with each element $\Sigma_{ij} = \rho^{|i-j|}$. As a consequence, the generated features have 0 mean, a variance of 1, and the correlation structure as:

$$E[X_i] = 0, \quad E[X_i^2] = 1 \quad \text{and} \quad E[X_iX_j] = \rho^{|i-j|}. \quad (5)$$

Our simulation scheme also includes the parameter density $= 0.2$ that controls the proportion of non-zero elements in $\theta^*$. The target vector is generated according to linear relationship with Gaussian noise:

$$y = X\theta^* + \epsilon,$$

such that the signal-to-noise ratio is $\text{snr} = \frac{\|X\theta^*\|_2}{\|\epsilon\|_2}$. 
We use a signal-to-noise ratio $\text{SNR} = 3$, a correlation $\rho$ of 0 or 0.6 with $n = 500$ samples and $p = 1000$ features.

**Description of the solvers** Table B.2 describes the different solvers compared in this benchmark.

**Table B.2: List of solvers used in the Ridge benchmark in Appendix B**

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Description</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerated GD</td>
<td>Nesterov (1983)</td>
<td>Gradient Descent + acceleration</td>
<td>Python</td>
</tr>
<tr>
<td>scikit-learn[svd]</td>
<td>Pedregosa et al. (2011)</td>
<td>SVD (Singular Value Decomposition)</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[cholesky]</td>
<td>Pedregosa et al. (2011)</td>
<td>Cholesky decomposition</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[lsqr]</td>
<td>Pedregosa et al. (2011)</td>
<td>regularized least-squares</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[saga]</td>
<td>Pedregosa et al. (2011)</td>
<td>SAGA (Varianced reduced stochastic method)</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[cg]</td>
<td>Pedregosa et al. (2011)</td>
<td>Conjugate Gradient</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>CD</td>
<td>Bertsekas (1999)</td>
<td>Cyclic Coordinate Descent</td>
<td>Python (Numba)</td>
</tr>
<tr>
<td>lightning[cd]</td>
<td>Blondel and Pedregosa (2016)</td>
<td>Cyclic Coordinate Descent</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>snapML[cpu]</td>
<td>Dünner et al. (2018)</td>
<td>CD</td>
<td>Python, C++</td>
</tr>
<tr>
<td>snapML[gpu]</td>
<td>Dünner et al. (2018)</td>
<td>CD + GPU</td>
<td>Python, C++</td>
</tr>
</tbody>
</table>

**Results** Figure B.1 presents the performance of the different methods for different values of the regularization parameter in the benchmark. The algorithms based on the direct computation of the closed-form solution outperform iterative ones in a majority of presented datasets. Among closed-form algorithms, the Cholesky solver converges faster.
Figure B.1: Benchmark for the Ridge regression, on 10 solvers, 5 datasets (rows), and 3 variants of the Objective (columns) each with a different regularization value $\lambda \in \{0.01, 0.1, 1\}$. The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time.
C  Design choices

Benchopt has made some design choices, while trying as much as possible to leave users free of customizing the behavior on each benchmark. We detail the most important ones in this section.

C.1  Estimating $\theta^*$ for convex problems

When the problem is convex, many solvers are guaranteed to converge to a global minimizer $\theta^*$ of the objective function $f$. To estimate $\theta^*$ and $f(\theta^*)$, Benchopt approximates $\theta^*$ by the iterate achieving the lowest objective value among all solvers for a given Dataset and Objective. This means that the sub-optimality plot proposed by Benchopt are only valid if at least one solver has converged to the optimal solution. Else, the curves are a lower bound estimate of the sub-optimality. In practice, for most considered convex problems, running the Solver for long enough ensures that $f(\theta^*)$ is correctly estimated.

C.2  Stopping solvers

Benchopt offers many ways to stop running a solver. The most common is to stop the solver when the objective value does not decrease significantly between iterations. For some convex problems, we also propose to track the duality gap (which upper bounds the suboptimality), as is done for the Lasso. For non convex problems, criteria such as gradient norm or violation of first order conditions can be used, as users do in practice. These criteria can easily be customized.

C.3  Wall-clock time versus number of iterations

Measuring time or iteration are two alternatives that make sense in their respective contexts. Practitioners mostly care about the time it takes to solve their problem, while researchers in mathematical optimization may want to abstract away the implementation and hardware details and only consider iteration. The benchmarks we have presented showcase efficient implementations and are also interested in hardware and implementation differences (e.g. CPU vs GPU solvers for ??, torch versus tensorflow for Section F.4), hence our focus on time. However, Benchopt does not impose a choice between the two measures: it is perfectly possible to create plots as a function of the number of iterations as evidenced for example in ??.
D \quad \ell_2\text{-regularized logistic regression}

D.1 List of solvers and datasets used in the benchmark in Section 3

Table D.1 and Table D.2 respectively present the Solvers and Datasets used in this benchmark.

Table D.1: List of solvers used in the \ell_2\text{-regularized logistic regression benchmark in Section 3}

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Description</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightning[sag]</td>
<td>Blondel and Pedregosa (2016)</td>
<td>SAG</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>lightning[saga]</td>
<td>Blondel and Pedregosa (2016)</td>
<td>SAG</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>lightning[cd]</td>
<td>Blondel and Pedregosa (2016)</td>
<td>Cyclic Coordinate Descent</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>Tick[svrg]</td>
<td>Bacry et al. (2017)</td>
<td>Stochastic Variance Reduced Gradient</td>
<td>Python, C++</td>
</tr>
<tr>
<td>scikit-learn[sgd]</td>
<td>Pedregosa et al. (2011)</td>
<td>Stochastic Gradient Descent</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[sag]</td>
<td>Pedregosa et al. (2011)</td>
<td>SAG</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[saga]</td>
<td>Pedregosa et al. (2011)</td>
<td>SAGA</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[newton-cg]</td>
<td>Pedregosa et al. (2011), Virtanen et al. (2020)</td>
<td>Truncated Newton Conjugate-Gradient</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>cuML[gpu]</td>
<td>Raschka et al. (2020)</td>
<td>L-BFGS + GPU</td>
<td>Python, C++</td>
</tr>
</tbody>
</table>

Table D.2: List of the datasets used in \ell_2\text{-regularized logistic regression in Section 3}

<table>
<thead>
<tr>
<th>Datasets</th>
<th>References</th>
<th>Samples (n)</th>
<th>Features (p)</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>ijcnn1</td>
<td>Prokhorov (2001)</td>
<td>49 990</td>
<td>22</td>
<td>4.5 \times 10^{-2}</td>
</tr>
<tr>
<td>madelon</td>
<td>Guyon et al. (2004)</td>
<td>2000</td>
<td>500</td>
<td>2.0 \times 10^{-3}</td>
</tr>
<tr>
<td>news20.binary</td>
<td>Keerthi et al. (2005)</td>
<td>19 996</td>
<td>1 355 191</td>
<td>3.4 \times 10^{-4}</td>
</tr>
<tr>
<td>criteo</td>
<td>Criteo-Labs (2015)</td>
<td>45 840 617</td>
<td>1 000 000</td>
<td>3.9 \times 10^{-5}</td>
</tr>
</tbody>
</table>

D.2 Results

Figure D.1 presents the performance results for the different solvers on the different datasets using various regularization parameter values, on unscaled raw data. We observe that when the regularization parameter \( \lambda \) increases, the problem tends to become easier and faster to solve for most methods. Also, the relative order of the method does not change significantly for the considered range of regularization.
Figure D.1: Additional benchmark for the $\ell_2$-regularized logistic regression on variants of the Objective (columns) with fit_intercept=False. The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time. The columns correspond to the objective detailed in Problem (2) with different value of $\lambda$: (first) $\lambda = 0.1$, (second) $\lambda = 1$ and (third) $\lambda = 10$. 
E Lasso

E.1 List of solvers and datasets used in the Lasso benchmark in Section 4

Table E.1 and Table E.2 respectively present the Solvers and Datasets used in this benchmark.

Table E.1: List of solvers used in the Lasso benchmark in Section 4

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Description</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>blitz</td>
<td>Johnson and Guestrin (2015)</td>
<td>CD + working set (Cyclic Minimization along coordinates)</td>
<td>Python, C++</td>
</tr>
<tr>
<td>coordinate descent</td>
<td>Friedman et al. (2010)</td>
<td>CD + working set (Cyclic Minimization along coordinates)</td>
<td>Python (Numba)</td>
</tr>
<tr>
<td>celer</td>
<td>Massias et al. (2018)</td>
<td>CD + working set + dual extrapolation</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>cuML[cd]</td>
<td>Raschka et al. (2020)</td>
<td>(Cyclic) Minimization along coordinates</td>
<td>Python, C++</td>
</tr>
<tr>
<td>FISTA</td>
<td>Beck and Teboulle (2009)</td>
<td>ISTA + acceleration</td>
<td>Python</td>
</tr>
<tr>
<td>glmnet</td>
<td>Friedman et al. (2010)</td>
<td>CD + working set + strong rules</td>
<td>R, C++</td>
</tr>
<tr>
<td>ISTA</td>
<td>Daubechies et al. (2004)</td>
<td>ISTA (Proximal GD)</td>
<td>Python</td>
</tr>
<tr>
<td>LARS</td>
<td>Efron et al. (2004)</td>
<td>Least-Angle Regression algorithm (LARS)</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>FISTA[adaptive-1]</td>
<td>Liang et al. (2022, Algo 4), Farrenς et al. (2020)</td>
<td>FISTA + adaptive restart</td>
<td>Python</td>
</tr>
<tr>
<td>FISTA[greedy]</td>
<td>Liang et al. (2022, Algo 5), Farrenς et al. (2020)</td>
<td>FISTA + greedy restart</td>
<td>Python</td>
</tr>
<tr>
<td>noncvx-pro</td>
<td>Poon and Peyré (2021)</td>
<td>Bilevel optim + L-BFGS</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>skglm</td>
<td>Bertrand et al. (2022)</td>
<td>CD + working set + primal extrapolation</td>
<td>Python (Numba)</td>
</tr>
<tr>
<td>scikit-learn</td>
<td>Pedregosa et al. (2011)</td>
<td>CD</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>snapML[gpu]</td>
<td>Dünner et al. (2018)</td>
<td>CD + GPU</td>
<td>Python, C++</td>
</tr>
<tr>
<td>snapML[cpu]</td>
<td>Dünner et al. (2018)</td>
<td>CD</td>
<td>Python, C++</td>
</tr>
<tr>
<td>lasso.jl</td>
<td>Kornblith (2021)</td>
<td>CD</td>
<td>Julia</td>
</tr>
</tbody>
</table>

Table E.2: List of datasets used in the Lasso benchmark in Section 4

<table>
<thead>
<tr>
<th>Dataset</th>
<th>References</th>
<th>Samples (n)</th>
<th>Features (p)</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEG</td>
<td>Gramfort et al. (2014)</td>
<td>305</td>
<td>7498</td>
<td>1.0</td>
</tr>
<tr>
<td>news20</td>
<td>Keerthi et al. (2005)</td>
<td>19 996</td>
<td>1 355 191</td>
<td>3.4 × 10⁻⁴</td>
</tr>
<tr>
<td>rcv1</td>
<td>Lewis et al. (2004)</td>
<td>20 242</td>
<td>47 236</td>
<td>3.6 × 10⁻³</td>
</tr>
<tr>
<td>MillionSong</td>
<td>Bertin-Mahieux et al. (2011)</td>
<td>463 715</td>
<td>90</td>
<td>1</td>
</tr>
</tbody>
</table>

E.2 Support identification speed benchmark

Since the Lasso is massively used for its feature selection properties, the speed at which the solvers identify the support of the solution is also an important performance measure. To evaluate the behavior of solvers in this task, it is sufficient to add a single new variable in the Objective, namely the ℓ₀ pseudonorm of the iterate, allowing to produce Figure E.1 in addition to Figure 4.

E.3 Convergence in terms of iteration

??
Figure E.1: Additional benchmark for the Lasso on variants of the Objective (columns). The curves display the fraction of non-zero coefficients in iterates $\theta_t (\|\theta_t\|_0/p)$, as a function of time.

While practitioners are mainly concerned with the time it takes to solve their optimization problem, one may also be interested in the convergence as a function of the number of iterations. This is particularly relevant to compare theoretical convergence rates with experiments. Benchopt natively supports such functionality. Yet, this makes sense only if one iteration of each algorithm costs the same. Figure E.2 presents such a case on the leukemia dataset, using algorithms for which one iteration costs $n \times p$. One can observe that cyclic coordinate descent as implemented in Cython in scikit-learn or in Numba lead to identical results, while they outperform proximal gradient methods.
Figure E.2: Convergence speed with respect to the number of iterations for some solvers of the Lasso benchmark on the leukemia dataset.
F ResNet18

F.1 Description of the benchmark

Setting up the benchmark The three currently supported frameworks are TensorFlow/Keras (Abadi et al., 2015; Chollet et al., 2015), PyTorch (Paszke et al., 2019) and PyTorch Lightning (Falcon et al., 2020). We report here results for TensorFlow/Keras and PyTorch. To guarantee that the model behaves consistently across the different considered frameworks, we implemented several consistency unit tests. We followed the best practice of each framework to make sure to achieve the optimal computational efficiency. In particular, we tried as much as possible to use official code from the frameworks, and not third-party code. We also optimized and profiled the data pipelines to make sure that our training was not IO-bound. Our benchmarks were run using TensorFlow version 2.8 and PyTorch version 1.10.

Descriptions of the datasets

Table F.1: Description of the datasets used in the ResNet18 image classification benchmark

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Content</th>
<th>References</th>
<th>Classes</th>
<th>Train Size</th>
<th>Val. Size</th>
<th>Test Size</th>
<th>Image Size</th>
<th>RGB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>natural images</td>
<td>Krizhevsky (2009)</td>
<td>10</td>
<td>40k</td>
<td>10k</td>
<td>10k</td>
<td>32</td>
<td>✓</td>
</tr>
<tr>
<td>SVHN</td>
<td>digits in natural images</td>
<td>Netzer et al. (2011)</td>
<td>10</td>
<td>58.6k</td>
<td>14.6k</td>
<td>26k</td>
<td>32</td>
<td>✓</td>
</tr>
<tr>
<td>MNIST</td>
<td>handwritten digits</td>
<td>LeCun et al. (2010)</td>
<td>10</td>
<td>50k</td>
<td>10k</td>
<td>10k</td>
<td>28</td>
<td>✓</td>
</tr>
</tbody>
</table>

In Table F.1, we present some characteristics of the different datasets used for the ResNet18 benchmark. In particular, we specify the size of each splits when using the train-validation-test split strategy. The test split is always fixed, and is the official one.

While the datasets are downloaded and preprocessed using the official implementations of the frameworks, we made sure to test that they matched using a unit test.

ResNet The ResNet18 is the smallest variant of the architecture introduced by He et al. (2016). It consists in 3 stages:

1. A feature extension convolution that goes from 3 channels (RGB, or a repeated grayscale channel in the MNIST case) to 64, followed by a batch normalization and a ReLU.

2. A series of residual blocks. Residual blocks are grouped by scale, and each individual group starts with a strided convolution to reduce the image scale (except the first one). As the scale increases, so does the number of features (64, 128, 256, 512). In the ResNet18 case, each scale group has two individual residual blocks and there are four scales. A residual block is comprised of three convolution operations, all followed by a batch normalization layer, and the first two also followed by a ReLU. The input is then added to the output of the third batch normalization layer before being fed to a ReLU.

3. A classification head that performs global average pooling, before applying a fully connected (i.e. dense) layer to obtain logits.

Training’s hyperparameters

Table F.2: Hyperparameters used for each solver. If a hyperparameter’s value is not specified in the table, it was set as the default of the implementation (checked to be consistent across frameworks).

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>SGD</th>
<th>Adam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>0.1</td>
<td>0.001</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.9</td>
<td>N/A</td>
</tr>
<tr>
<td>Weight Decay</td>
<td>$5 \times 10^{-4}$</td>
<td>0.02</td>
</tr>
<tr>
<td>Batch Size</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>
In Table F.2, we specify the hyperparameters we used for the benchmark. For the SGD, the values were taken from the pytorch-cifar GitHub repository, while for Adam we took the most relevant ones from the work of Wightman et al. (2021).

F.2 Hyperparameter sensitivity

In the benchmark presented in Section 5, we consider fixed hyperparameters chosen from common practices to train ResNet18 models for an image classification task. However, in practice, these hyperparameters must be carefully set, either via a grid search, or via more adapted algorithms such as random search (bergstra2012random) or bayesian optimization (Paszke et al., 2019). It is therefore important to evaluate how sensitive an optimizer is to choosing the right parameters, as more sensitive methods will require more exhaustive hyperparameters search. In Figure F.1, we study this issue using Benchopt for image classification on CIFAR-10. Despite achieving the best results in terms of accuracy, SGD is way more sensitive to the choice of hyperparameters than Adam.\(^6\)

Another way to look at hyperparameter sensitivity is to evaluate how a given selection of hyperparameters performs for different tasks. Figure 5 shows that while SGD is sensitive to the choice of learning rate and weight decay, the selected values work very well across 3 different datasets.

F.3 Aligning TensorFlow and PyTorch ResNet18 training

We summarized in Table F.3 the different elements that have to be considered to align the training of a ResNet18 in PyTorch and TensorFlow. Let us detail here some lines of this table:

- **Bias in convolutions:** It can be seen in TensorFlow/Keras official implementation, that convolutions operations use a bias. This is in contrast to PyTorch’s official implementation in torchvision which does not. Since the convolutions are followed by batch normalization layers, with a mean removal, the convolutions’ bias is a spurious parameter, as was noted by Ioffe and Szegedy (2015). We therefore chose to use unbiased convolutions.

- **Decoupled weight decay scaling:** this led us to scale manually the weight decay used in TensorFlow by the learning rate when setting it. Moreover, because the weight decay is completely decoupled from the learning rate, it is important to update it accordingly when using a learning rate schedule, as noted in the TensorFlow documentation.

- **Batch normalization momentum:** an important note here is that the convention used to implement the batch normalization momentum is not the same in the 2 frameworks. Indeed we have the relationship $\text{momentum}_{\text{TF}} = 1 - \text{momentum}_{\text{PT}}$.

\(^6\)We ran the same experiment on two other datasets obtaining similar figures.
Table F.3: Differences in off-the-shelf implementations of various components when training ResNet18 for image classification in PyTorch and TensorFlow. The selected versions are put in bold font for components that we were able to reconcile. This highlights the numerous details to consider when comparing experimental results.

<table>
<thead>
<tr>
<th>Component</th>
<th>PyTorch</th>
<th>TensorFlow/Keras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias in convolutions</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Decoupled weight decay scaling</td>
<td>Multiplied by learning rate</td>
<td>Completely decoupled</td>
</tr>
<tr>
<td>Batch normalization momentum</td>
<td>0.9</td>
<td>0.99</td>
</tr>
<tr>
<td>Conv2D weights init.</td>
<td>Fan out, normal</td>
<td>Fan average, uniform</td>
</tr>
<tr>
<td>Classification head init. (weights)</td>
<td>Fan in, uniform</td>
<td>Fan average, uniform</td>
</tr>
<tr>
<td>Classification head init. (bias)</td>
<td>Fan in, uniform</td>
<td>Zeros</td>
</tr>
<tr>
<td>Striding in convolutions</td>
<td>Starts one off</td>
<td>Ends one off</td>
</tr>
<tr>
<td>Variance estimation in batch norm</td>
<td>unbiased (eval)/biased (training)</td>
<td>biased</td>
</tr>
</tbody>
</table>

- **Conv2D weights initialization**: TensorFlow/Keras uses the default initialization which is a Glorot uniform initialization (Glorot and Bengio, 2010). PyTorch uses a He normal initialization (He et al., 2015). We used TensorFlow’s Variance Scaling framework to differentiate the 2.

- **Striding in convolutions**: when using a stride of 2 in convolutions on an even-size image, one needs to specify where to start the convolution in order to know which lines (one in every two) in the image will be removed. The decision is different between TensorFlow and PyTorch. This is not expected to have an effect on the final performance, but it makes it more difficult to compare the architectures when unit testing. We therefore decided to align the models on this aspect as well.

- **Variance estimation in batch normalization**: in order to estimate the batch variance during training for batch normalization layers, it is possible to chose between the unbiased and the biased variance estimator. The unbiased variance estimator applies a Bessel correction to the biased variance estimator, namely a multiplication by a factor \( \frac{m}{m-1} \), where \( m \) is the number of samples used to estimate. It is to be noted that PyTorch does uses the biased estimator in training, but stores the unbiased estimator for use during inference. TensorFlow does not allow for such a behaviour, and the 2 are therefore not reconcilable\(^7\). Arguably this inconsistency should not play a big role with large batch sizes, but can be significant for smaller batches, especially in deeper layers where the feature map size (and therefore the number of samples used to compute the estimates) is reduced.

**Adapting official ResNet implementations to small images** In addition to these elements, it is important to adapt the reference implementations of both frameworks to the small image case. Indeed, for the case of ImageNet, the ResNet applies two downsampling operations (a stride-2 convolution and a max pooling) at the very beginning of the network to make the feature maps size more manageable. In the case of smaller images, it is necessary to do without these downsampling operations (i.e. perform the convolution with stride 1 and get rid of the max pooling).

**Coupled weight decay in TensorFlow** In TensorFlow, the SGD implementation does not allow the setting of coupled weight decay. Rather, one has to rely on the equivalence (up to a scale factor of 2) between coupled weight decay and L2 regularization. However, in TensorFlow/Keras, adding L2 regularization on an already built model (which is the case for the official ResNet implementation), is not straightforward and we relied on the workaround of Silva (2019).

**F.4 VGG benchmark on CIFAR-10**

In order to show how flexible Benchopt is, we also ran a smaller version of the ResNet benchmark using a VGG16 (Simonyan and Zisserman, 2015) network instead of a ResNet18. In the Benchopt framework, this amounts to specifying a different model in the objective, while all the other pieces of code in the benchmark remain unchanged. Note that the VGG official implementations also need to

\(^7\)It is possible to use the unbiased estimator in TensorFlow for the batch normalization, even if not documented, but its application is consistent between training and inference unlike PyTorch.
Figure F.2: **VGG16 image classification benchmark with PyTorch solvers.** The best SGD configuration features data augmentation, momentum, step learning rate schedule and weight decay.

be adapted to the CIFAR-10 case by changing the classification head. This was not specified in the original paper, where no experiment was conducted on small-scale datasets, and we relied on available open source implementations (cifar10-vgg16 and cifar-vgg) to make this decision. Importantly, these implementations use batch normalization to make the training of VGG more robust to initialization, which is not the case in the official framework implementations.

In Figure F.2, we see that for the case of VGG, the application of weight decay is so important that without it, in cases with momentum, the model does not converge.
G  \( \ell_1 \)-regularized logistic regression

This additional benchmark is dedicated to \( \ell_1 \)-regularized logistic regression, in the same setting as Problem (2) but this time with an \( \ell_1 \)-regularization for the parameters of the model:

\[
\theta^* = \arg \min_{\theta \in \mathbb{R}^p} \sum_{i=1}^{n} \log \left( 1 + \exp(-y_i X_i^\top \theta) \right) + \lambda \|\theta\|_1 .
\] (6)

G.1 List of solvers and datasets used in the \( \ell_1 \)-regularized logistic regression benchmark

The code for the benchmark is available at https://github.com/benchopt/benchmark_logreg_l1/. Table G.1 and Table G.2 present the solvers and datasets used in this benchmark.

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Description</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>blitz</td>
<td>Johnson and Guestrin (2015)</td>
<td>CD + working set</td>
<td>Python, C++</td>
</tr>
<tr>
<td>coordinate descent</td>
<td>Friedman et al. (2010)</td>
<td>(Cyclic) Minimization along coordinates</td>
<td>Python (Numba)</td>
</tr>
<tr>
<td>coordinate descent (Newton)</td>
<td>Friedman et al. (2010)</td>
<td>CD + Newton</td>
<td>Python (Numba)</td>
</tr>
<tr>
<td>celer</td>
<td>Massias et al. (2018)</td>
<td>CD + working set + dual extrapolation</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>copt[FISTA line search]</td>
<td>Pedregosa et al. (2020), Beck and Teboulle (2009)</td>
<td>FISTA (ISTA + acceleration) + line search</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>copt[saga]</td>
<td>Pedregosa et al. (2020)</td>
<td>SAGA (Variance reduced stochastic method)</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>copt[svrg]</td>
<td>Pedregosa et al. (2020)</td>
<td>SVRG (Variance reduced stochastic method)</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>cuML[gpu]</td>
<td>Raschka et al. (2020)</td>
<td>L-BFGS + GPU</td>
<td>Python, C++</td>
</tr>
<tr>
<td>cyanure</td>
<td>Mairal (2019)</td>
<td>Proximal Minimization by Incremental Surrogate Optimization (MISO)</td>
<td>Python, C++</td>
</tr>
<tr>
<td>lightning</td>
<td>Blondel and Pedregosa (2016)</td>
<td>(Cyclic) Coordinate Descent</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>scikit-learn[newton-cg]</td>
<td>Pedregosa et al. (2011), Virtanen et al. (2020)</td>
<td>Truncated Newton Conjugate-Gradient</td>
<td>Python (Cython)</td>
</tr>
<tr>
<td>snapml[gpu=True]</td>
<td>Dünner et al. (2018)</td>
<td>CD + GPU</td>
<td>Python, C++</td>
</tr>
</tbody>
</table>
Table G.2: List of the datasets used in the $\ell_1$-regularized logistic regression benchmark

<table>
<thead>
<tr>
<th>Datasets</th>
<th>References</th>
<th>Samples (n)</th>
<th>Features (p)</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>gisette</td>
<td>Guyon et al. (2004)</td>
<td>6000</td>
<td>5000</td>
<td>$9.9 \times 10^{-1}$</td>
</tr>
<tr>
<td>colon-cancer</td>
<td>Guyon et al., 2004</td>
<td>62</td>
<td>2000</td>
<td>1.0</td>
</tr>
<tr>
<td>news20.binary</td>
<td>Keerthi et al. (2005)</td>
<td>19996</td>
<td>1355 191</td>
<td>$3.4 \times 10^{-4}$</td>
</tr>
<tr>
<td>rcv1.binary</td>
<td>Guyon et al., 2004</td>
<td>20242</td>
<td>19959</td>
<td>$3.6 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

G.2 Results

The results of the $\ell_1$-regularized logistic regression benchmark are in Figure G.1.
H Unidimensional total variation

The use of 1D Total Variation regularization takes its root in the taut-string algorithm (Barlow and Brunk, 1972) and can be seen as a special case of either the Rudin-Osher-Fatemi model (Rudin et al., 1992) or the Generalized Lasso (Tibshirani and Taylor, 2011) for a quadratic data fit term. It reads

$$\theta^* \in \arg \min_{\theta \in \mathbb{R}^p} F(y, X\theta) + \lambda \|D\theta\|_1,$$  \hspace{1cm} (7)

where $F$ is a data fidelity term, $X \in \mathbb{R}^{n \times p}$ is a design matrix with $n$ samples and $p$ features, $y \in \mathbb{R}^n$ is the target vector, $\lambda > 0$ is a regularization hyperparameter, and $D \in \mathbb{R}^{(p-1) \times p}$ is a finite difference operator defined by $(D\theta)_k = \theta_{k+1} - \theta_k$ for all $1 \leq k \leq p - 1$ (it is also possible to use cyclic differences).

Most often, the data fidelity term is the $\ell^2$-loss $F(y, z) = \frac{1}{2} \|y - z\|^2_2$, following an additive Gaussian noise hypothesis. But the data fit term can also account for other types of noises, such as noises with heavy tails using the Huber loss $F(y, z) = |y - z|_\mu$ where $| \cdot |_\mu$ is defined coordinate-wise by

$$|t|_\mu = \begin{cases} \frac{1}{2} t^2 & \text{if } |t| \leq \mu \\ \mu |t| - \frac{\mu^2}{2} & \text{otherwise.} \end{cases}$$

Problem (7) promotes piecewise-constant solutions – alternatively said, solutions such that their gradients is sparse – and was proved to be useful in several applications, in particular for change point detection (Bleakley and Vert, 2011; Tibshirani, 2014), for BOLD signal deconvolution in functional MRI (Karahanoğlu et al., 2013; Cherkaoui et al., 2019) or for detrending in oculomotor recordings (Lalanne et al., 2020).

The penalty $\theta \mapsto \|D\theta\|_1$ is convex but non-smooth, and its proximity operator has no closed form. Yet as demonstrated by Condat (2013a), the taut-string algorithm allows to compute this proximity operator in $O(p^2)$ operations in the worst case, but it enjoys a $O(p)$ complexity in most cases. Other methods do not rely on this proximity operator and directly solve Problem (7), using either primal-dual approaches (Chambolle and Pock, 2011; Condat, 2013b), or solving the dual problem (Komodakis and Pesquet, 2015). Finally, for 1-dimensional TV regularization, one can also use the synthesis formulation (Elad et al., 2006) to solve the problem. By setting $z = D\theta$ and $\theta = Lz + \rho$ where $L \in \mathbb{R}^{p \times p-1}$ is a lower triangular matrix representing an integral operator (cumulative sum), the problem is equivalent to a Lasso problem, and $\rho^*$ has a closed-form expression (see e.g., Bleakley and Vert 2011 for a proof). As a consequence, any lasso solver can be used to obtain the solution of the Lasso problem $z^*$ and the solution of the original Problem (7) $u^*$ is retrieved as $u^* = Lz^* + \rho^*$.

The code for the benchmark is available at https://github.com/benchopt/benchmark_tv_1d/ and Table H.1 details the different algorithms used in this benchmark.

Table H.1: List of solvers used in the 1D Total Variation benchmarks

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Formulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMM</td>
<td>Boyd et al. (2011)</td>
<td>Analysis</td>
<td>Primal-Dual Augmented La-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>grangian</td>
</tr>
<tr>
<td>ChambollePock</td>
<td>Chambolle and Pock (2011)</td>
<td>Analysis</td>
<td>Primal-Dual Hybrid Gradient</td>
</tr>
<tr>
<td>CondatVu</td>
<td>Condat (2013b)</td>
<td>Analysis</td>
<td>Primal-Dual Hybrid Gradient</td>
</tr>
<tr>
<td>DPGD</td>
<td>Komodakis and Pesquet (2015)</td>
<td>Analysis</td>
<td>Dual proximal GD (+ acceleration)</td>
</tr>
<tr>
<td>PGD</td>
<td>Condat (2013a)</td>
<td>Analysis</td>
<td>Proximal GD + taut–string</td>
</tr>
<tr>
<td></td>
<td>Barbero and Sra (2018)</td>
<td></td>
<td>ProxTV (+ acceleration)</td>
</tr>
<tr>
<td>celer</td>
<td>Massias et al. (2018)</td>
<td>Synthesis</td>
<td>CD + working set (lasso)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>only for $\ell_2$ data-fit</td>
</tr>
<tr>
<td>FP</td>
<td>Combettes and Glaudin (2021)</td>
<td>Synthesis</td>
<td>Fixed point with block updates</td>
</tr>
<tr>
<td>ISTA</td>
<td>Daubechies et al. (2004)</td>
<td>Synthesis</td>
<td>Proximal GD (+ acceleration)</td>
</tr>
<tr>
<td>skglm</td>
<td>Bertrand et al. (2022)</td>
<td>Synthesis</td>
<td>CD + working set</td>
</tr>
</tbody>
</table>
Simulated dataset  We use here simulated data, as applications based on fMRI and EOG signals require access to open and preprocessed data that we will make available on OpenML Vanschoren et al., 2013 in the future. The data are generated as follows: a block signal \( \theta \in \mathbb{R}^p \) is generated by first sampling a sparse random vector \( z \in \mathbb{R}^p \) with \( K \) non-zero coefficients positioned randomly, and taking random values following a \( \mathcal{N}(0, 1) \) distribution. Finally, \( \hat{\theta} \) is obtained by discrete integration as \( \hat{\theta}_i = \sum_{k=1}^i z_k \) for \( 1 \leq i \leq p \). The design matrix \( X \in \mathbb{R}^{n \times p} \) is a Gaussian random design with \( X_{ij} \sim \mathcal{N}(0, 1) \). The observations \( y \) are obtained as \( y = X\hat{\theta} + \epsilon \), with \( \epsilon \sim \mathcal{N}(0, 0.01) \) a Gaussian white noise. For all experiments, we used \( p = 500 \) and vary the number of non-zero coefficient \( K \), and the number of rows \( n \) of the matrix \( X \).

Results  Figure H.1 shows that the solvers using the synthesis formulation and coordinate descent-based solvers for the Lasso (\( \ell_2 \) data fit term) work best on this type of problem. For the Huber data fit term, the solver using the analysis formulation and the taut-string algorithm for the proximal operator are faster. An interesting observation from this benchmark is the behavior of the solvers based on primal-dual splitting or dual formulation. We observe that for all these solvers, the objective starts by increasing. This is probably due to a sub-optimal initialization of the dual variable compared to the primal one. While this initialization is seldom described in the literature, it seems to have a large
impact on the speed of these algorithms. This shows how Benchopt allows to reveal such behavior, and could lead to practical guidelines on how to select this dual initialization.

Extensions We plan to extend this benchmark in the future to consider higher dimensional problems – e.g., 2D TV problems for images – or higher order TV regularization, such as Total Generalized Variation Bredies et al., 2010 or inf-convolution of TV functionals Chambolle and Lions, 1997 – used of instance for change point detection (Tibshirani, 2014). Yet, for 2D or higher dimensional problems, we can no longer use the synthesis formulation. It is however possible to apply the taut-string method of Condat (2013a) and graph-cut methods of Boykov et al. (2001) and Kolmogorov and Zabin (2004) for anisotropic TV, and dual or primal-dual methods for isotropic, such as Primal-Dual Hybrid Gradient algorithm (Chambolle and Pock, 2011).
I Linear regression with minimax concave penalty (MCP)

The Lasso problem (Tibshirani, 1996) is a least-squares regression problem with a convex non-smooth penalty that induces sparsity in its solution. However, despite its success and large adoption by the machine learning and signal processing communities, it is plagued with some statistical drawbacks, such as bias for large coefficients. To overcome these issues, the standard approach is to consider non-convex sparsity-inducing penalties. Several penalties have been proposed: Smoothly Clipped Absolute Deviation (SCAD, Fan and Li 2001), the Log Sum penalty (Candès et al., 2008), the capped-$\ell_1$ penalty (Zhang, 2010b) or the Minimax Concave Penalty (MCP, Zhang 2010a).

This benchmark is devoted to least-squares regression with the latter, namely the problem:

$$\theta^* \in \arg \min_{\theta \in \mathbb{R}^p} \frac{1}{2n} \| y - X\theta \|_2^2 + \sum_{j=1}^p \rho_{\lambda,\gamma}(\theta_j),$$  \hspace{1cm} (8)

where $X \in \mathbb{R}^{n \times p}$ is a design matrix containing $p$ features as columns, $y \in \mathbb{R}^n$ is the target vector, and $\rho_{\lambda,\gamma}$ the penalty function that reads as:

$$\rho_{\lambda,\gamma}(t) = \begin{cases} 
\lambda |t| - \frac{t^2}{2\gamma}, & \text{if } |t| \leq \gamma \lambda, \\
\frac{\gamma^2 \lambda^2}{2}, & \text{if } |t| > \gamma \lambda.
\end{cases}$$

Similarly to the Lasso, Problem (8) promotes sparse solutions but the optimization problem raises some difficulties due to the non-convexity and non-smoothness of the penalty. Nonetheless, several efficient algorithms have been derived for solving it. The ones we use in the benchmark are listed in Table I.1.

Table I.1: List of solvers used in the MCP benchmark

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>Breheny and Huang (2011), Mazumder et al. (2011)</td>
<td>Proximal coordinate descent</td>
</tr>
<tr>
<td>PGD</td>
<td>Bolte et al. (2014)</td>
<td>Proximal gradient descent</td>
</tr>
<tr>
<td>GIST</td>
<td>Gong et al. (2013)</td>
<td>Proximal gradient + Barzilai-Borwein rule</td>
</tr>
<tr>
<td>WorkSet CD</td>
<td>Boisbunon et al. (2014)</td>
<td>Coordinate descent + Barzilai-Borwein rule</td>
</tr>
<tr>
<td>skglm</td>
<td>Bertrand et al. (2022)</td>
<td>Accelerated coordinate descent + Working set</td>
</tr>
</tbody>
</table>

The code for the benchmark is available at https://github.com/benchopt/benchmark_mcp/. For this benchmark, we run the solvers on the colon-cancer dataset and on the simulated dataset described in Table B.4. We use a signal-to-noise ratio $\text{snr} = 3$, a correlation $\rho = 0.6$ with $n = 500$ observations and $p = 2000$ features.

I.1 Results

The result of the benchmark is presented in Figure I.1. The problem is non-convex and solvers are only guaranteed to converge to local minima; hence in Figure I.1 we monitor the distance of the negative gradient to the Fréchet subdifferential of the MCP, representing the violation of the first order optimality condition. Other metrics, such as objective of iterates sparsity, are monitored in the full benchmark, allowing to compare the different limit points obtained by the solvers.
Figure I.1: Benchmark for the MCP regression on variants of the Objective (columns). The curves display the violation of optimality conditions, $\text{dist}(−X^\top(X\theta_t − y)/n, ∂ρ_{λ,γ}(\theta_t))$, as a function of time. $γ$ is set to 3, and $λ$ is parameterized as a fraction of the Lasso’s $λ_{\text{max}}$, $\|X^\top y\|_{∞}/n$. 
J  Zero-order optimization on standard functions

Zero-order optimization refers to scenarios where only calls to the function to minimize are possible. This is in contrast with first-order optimization where gradient information is available. Grid search, random search, evolution strategies (ES) or Bayesian optimization (BO) are popular methods to tackle such a problem and are most commonly employed for hyperparameter optimization. This setting is also known as black-box optimization.

This benchmark demonstrates the usability of Benchopt for zero-order optimization considering functions classically used in the literature (Hansen et al., 2021). The functions are available in the PyBenchFCN package [link]. In particular, among the 61 functions of interest we present here (see Figure J.1) a benchmark for three functions, defined for any \( x = (x_1, \ldots, x_N) \in \mathbb{R}^N \):

\[
\begin{align*}
f(x) &= \sum_{i=1}^{N-1} [100(x_{i+1} - x_i)^2 + (1 - x_i)^2] \quad \text{(Rosenbrock)} \\
f(x) &= 10 \cdot N + \sum_{i=1}^{N} [(x_i^2 - 10 \cdot \cos(2\pi x_i))] \quad \text{(Rastrigin)} \\
f(x) &= -20 \cdot \exp \left[ -0.2 \sqrt{\frac{1}{d} \sum_{i=1}^{N} x_i^2} \right] - \exp \left[ \frac{1}{d} \sum_{i=1}^{N} \cos(2\pi x_i) \right] + e + 20 \quad \text{(Ackley)}
\end{align*}
\]

For each function, the domain is restricted to a box: \( \|x\|_\infty \leq 32 \) for Ackley, \( \|x\|_\infty \leq 30 \) for Rosenbrock and \( \|x\|_\infty \leq 5.12 \) for Rastrigin. The algorithms considered in the benchmark are listed in Table J.1. As BFGS requires first-order information, gradients are approximated with finite-differences.

Table J.1: List of solvers used in the zero-order benchmark

<table>
<thead>
<tr>
<th>Solver</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nevergrad-CMA</td>
<td>Rapin and Teytaud (2018) and Hansen and Ostermeier (2001)</td>
<td>CMA evolutionary strategy</td>
</tr>
<tr>
<td>Nevergrad-TwoPointsDE</td>
<td>Rapin and Teytaud (2018)</td>
<td>Evolutionary strategy</td>
</tr>
<tr>
<td>Nelder-Mead</td>
<td>Gao and Han (2012) and Virtanen et al. (2020)</td>
<td>Direct search (downhill simplex)</td>
</tr>
<tr>
<td>BFGS</td>
<td>Virtanen et al. (2020)</td>
<td>BFGS with finite differences</td>
</tr>
<tr>
<td>Powell</td>
<td>Powell (1964) and Virtanen et al. (2020)</td>
<td>Conjugate direction method</td>
</tr>
<tr>
<td>optuna-TPE</td>
<td>Akiba et al. (2019) and Bergstra et al. (2013)</td>
<td>Sampler by Tree Parzen Estimation (TPE)</td>
</tr>
<tr>
<td>optuna-CMA</td>
<td>Akiba et al. (2019) and Hansen and Ostermeier (2001)</td>
<td>CMA evolutionary strategy</td>
</tr>
</tbody>
</table>

The code for the benchmark is available at [link].

J.1 Results

The results of the benchmark are presented in Figure J.1. The functions are non-convex and solvers are only guaranteed to converge to local minima; hence in Figure J.1 we monitor the value of the function. The functions are designed such that the global minimum of the function is always 0. One
can observe that the CMA and TwoPointsDE implementations from nevergrad consistently reaches the global minimum. In addition, the CMA implementation from optuna is a bit slower than the one from nevergrad. Also one can notice that random search offers reasonable results. The TPE method seems to suffer from the curse of dimensionality, as most kernel methods in non-parametric estimation. Finally regarding the scipy solvers, Powell can be competitive, while Nelder–Mead and BFGS suffer a lot from local minima.