Benchopt: Reproducible, efficient and collaborative optimization benchmarks

Anonymous Author(s)
Affiliation
Address
email

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: ℓ_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 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.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

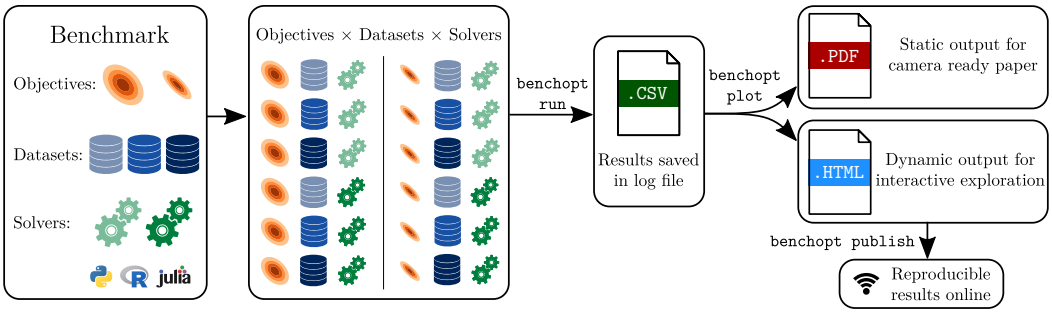

Figure 1: A visual summary of `Benchopt`. Each `Solver` is run (in parallel) on each `Dataset` and each variant of the `Objective`. Results are exported as a `CSV` file that is easily shared and can be automatically plotted as interactive `HTML` visualizations or `PDF` figures.

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:

- ℓ_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 coding standard to structure benchmarks for optimization in ML, where the problems depend on some input dataset $\mathcal{D}$. The considered problems are of the form

$$\theta^* \in \arg\min_{\theta \in \Theta} f(\theta; \mathcal{D}, \Lambda) \ , \tag{1}$$

where f is the objective function, Λ are its hyperparameters, and Θ is the feasible set for θ. The **scope** of the library is to evaluate optimization methods in their wide sense by considering the sequence $\{\theta^t\}_t$ produced to approximate θ^*. 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 Λ or the set Θ, 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 θ^t.

Datasets: The Dataset objects provide the data $\mathcal{D}$ 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 $\{\theta^t\}_t$. 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.

```
benchmark/
├── objective.py
├── datasets/
│   ├── dataset1.py
│   └── dataset2.py
└── solvers/
    ├── solver1.py
    └── solver2.py
```

Figure 2: Standard benchmark structure

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 and discusses the various design choices.

For each Dataset and Solver, and for each set of parameters, Benchopt retrieves a sequence $\{\theta^t\}_t$ and evaluates the metrics defined in the Objective for each θ^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. Also, benchmarks can easily be extended as one can add a Dataset or a Solver to the comparison by adding a single file. 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).

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 most meaningful evaluation. Besides a benchmark can easily be extended to track simultaneously an extra metric. Moreover, benchmarks can be run in parallel with checkpointing of the results, enabling large scale evaluations on many CPU or GPU nodes. Benchmark results are 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).

3 First example: ℓ_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, ℓ_2-regularized logistic regression provides a generalized linear model indexed by $\theta^* \in \mathbb{R}^p$ to discriminate the classes by solving

$$\theta^* = \underset{\theta \in \mathbb{R}^p}{\arg\min} \sum_{i=1}^{n} \log\left(1 + \exp(-y_i X_i^\top \theta)\right) + \frac{\lambda}{2}\|\theta\|_2^2 \ , \tag{2}$$

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).

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 not) with **sparse** features, which prevent explicit centering during preprocessing to keep memory usage limited. Details on packages, datasets and additional scenarios are available in Appendix C.

Results Figure C.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, θ^* is approximated by the best iterate across all runs (see Section B.5). 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]

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 –that

[1] The condition number of the dataset is divided by 5.9 after scaling.

[2] The condition number is multiplied by 407 after scaling.

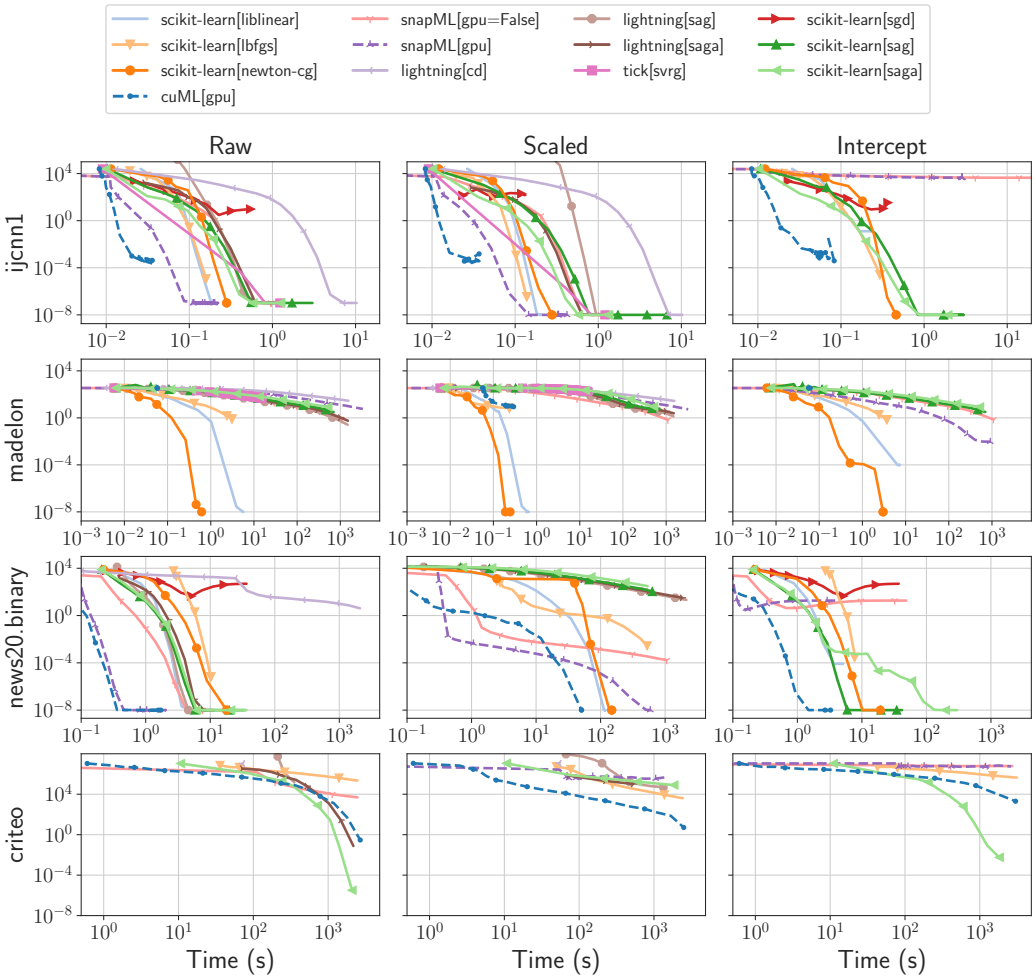

Figure 3: Benchmark for the ℓ_2-regularized logistic regression, on 13 solvers, 4 datasets (*rows*), and 3 variants of the `Objective` (*columns*) with $\lambda = 1$. The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time. The first column corresponds 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.

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 C.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 C.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

$$\theta^* \in \underset{\theta \in \mathbb{R}^p}{\arg\min} \, \tfrac{1}{2} \left\| y - X\theta \right\|^2 + \lambda \left\| \theta \right\|_1 \quad , \tag{3}$$

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 λ is high enough, many entries in θ^* 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[3]. 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 θ^*, 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.

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 D.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 λ, controlling the sparsity of the solution, parameterized as a fraction of $\lambda_{\max} = \left\| X^\top y \right\|_\infty$ (the minimal hyperparameter such that $\theta^* = 0$),
- 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 D.2).

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.

[3]The Lasso path is the set of solutions of Problem (3) as λ varies in $(0, \infty)$.

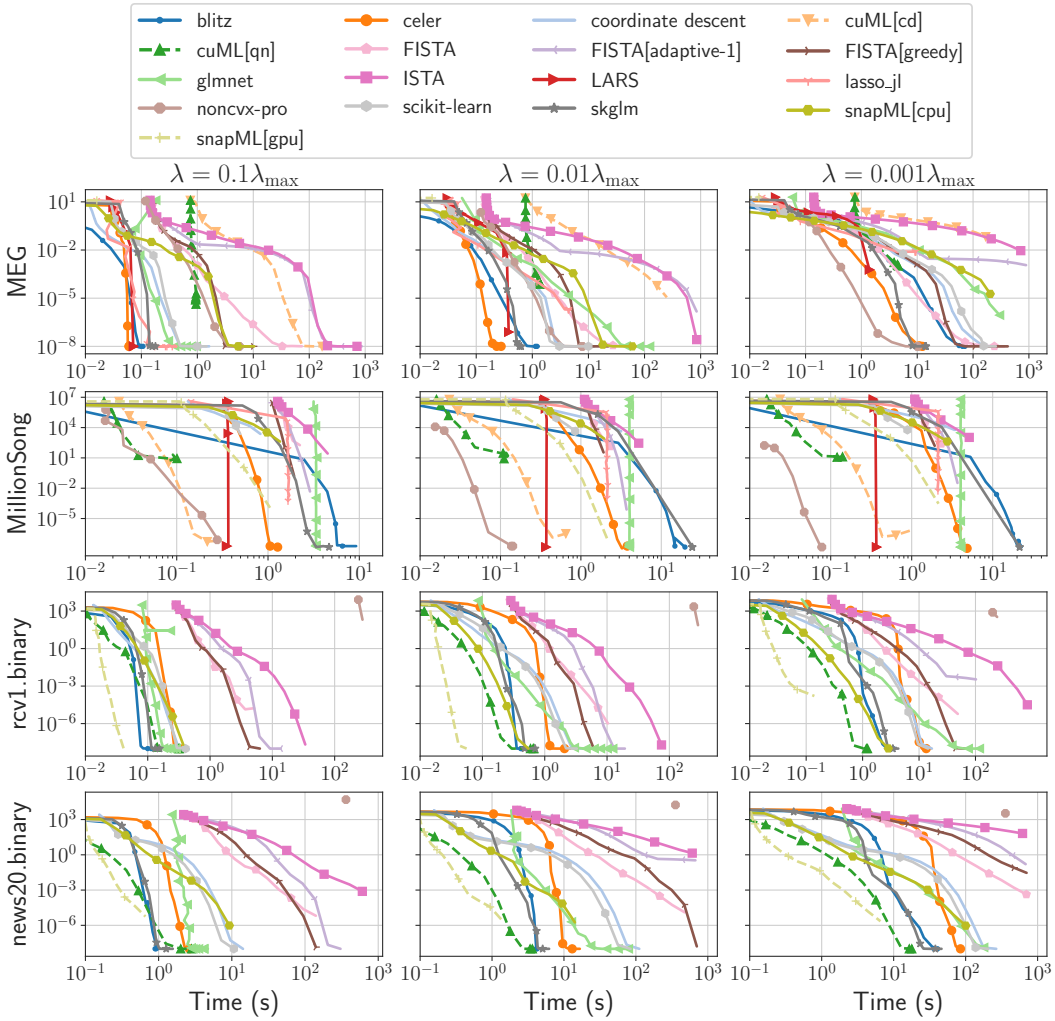

Figure 4: Benchmark for the Lasso, on 17 solvers, 4 datasets (*rows*), and 3 variants of the `Objective` (*columns*) with decreasing regularization λ. The curves display the suboptimality of the objective function, $f(\theta^t) - f(\theta^*)$, as a function of time.

When the support of the solution is small (either small λ, 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 it 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 D.

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

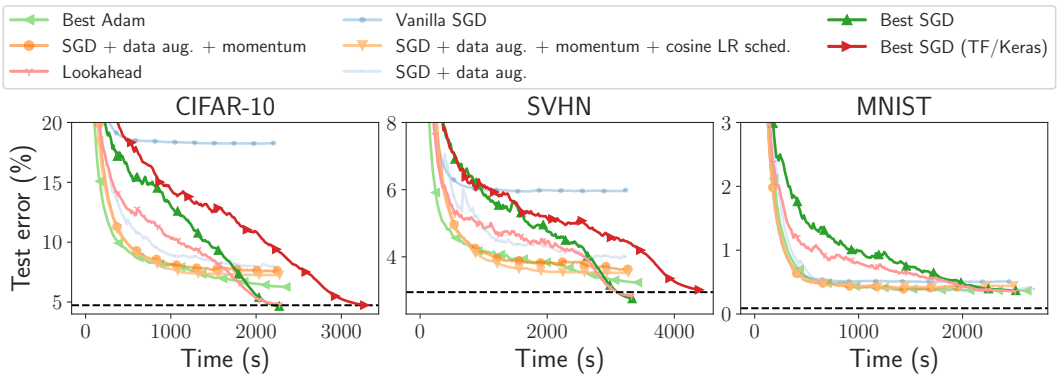

Figure 5: **ResNet18 image classification benchmark with `PyTorch Solvers`.** The best SGD configuration features data augmentation, momentum, cosine learning rate schedule and weight decay. In dashed black is the state of the art for the corresponding datasets with a ResNet18 measured by Zhang et al. (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 32×32 and is hence not shown for *MNIST*.

community as they provide strong and versatile baselines (Xie et al., 2017; 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.

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 E for details on architecture and datasets). We train the network by minimizing the cross entropy loss relatively to the weights θ of the model. Contrary to logistic regression and the Lasso, this problem is non-convex due to the non-linearity of the model f_θ. Another notable difference is that we report the evolution of the test error rather than the training 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. 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.,* ℓ_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 E.2.

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 E.2). We conducted an extensive effort to align them using unit testing to make the conclusions of our benchmarks independent of the chosen framework. We found additional significant differences (reported in Table E.3) in the initialization, the batch normalization, the convolutional layers and the weight decay scaling.

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. The results are available in the supplementary materials as a user-friendly interactive `HTML` file.

[4]decreasing the learning rate by a factor 10 at mid-training, and again at 3/4 of the training

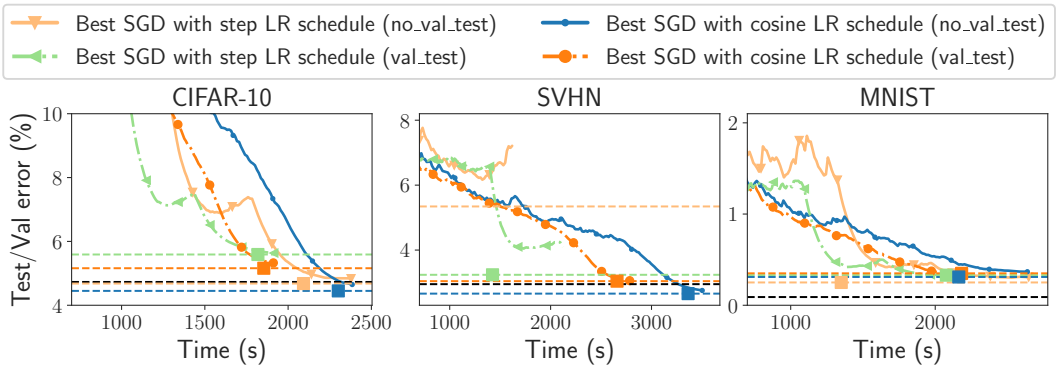

Figure 6: **ResNet18 image classification benchmark with a validation split.** In dashed black is the state of the art (see caption of Figure 5 for more details). In addition, we show in colored horizontal dashed lines, the test results for early stopping on the validation and on the test set for the different solvers, the square mark indicating the moment this stopping would happen. The curves for the train-val splits show the exponentially smoothed median results for five different random seeds.

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 E.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.

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.

References

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org.

Akiba, T., S. Sano, T. Yanase, T. Ohta, and M. Koyama (2019). *Optuna: A Next-generation Hyperparameter Optimization Framework*.

Bach, F., R. Jenatton, J. Mairal, and G. Obozinski (2012). "Optimization with sparsity-inducing penalties". In: *Foundations and Trends in Machine Learning* 4.1, pp. 1–106.

Bacry, E., M. Bompaire, S. Gaïffas, and S. Poulsen (2017). "Tick: A Python Library for Statistical Learning, with a Particular Emphasis on Time-Dependent Modeling". In: *ArXiv e-prints*.

Barbero, A. and S. Sra (2018). "Modular Proximal Optimization for Multidimensional Total-Variation Regularization". In: *Journal of Machine Learning Research* 19.56, pp. 1–82.

Barlow, R. E. and H. D. Brunk (1972). "The Isotonic Regression Problem and Its Dual". In: *Journal of the American Statistical Association* 67.337, pp. 140–147.

Bartz-Beielstein, T., C. Doerr, D. van den Berg, J. Bossek, S. Chandrasekaran, T. Eftimov, A. Fischbach, P. Kerschke, W. La Cava, M. Lopez-Ibanez, et al. (2020). "Benchmarking in optimization: Best practice and open issues". In: *arXiv preprint arXiv:2007.03488*.

Beck, A. and M. Teboulle (2009). "A fast iterative shrinkage-thresholding algorithm for linear inverse problems". In: *SIAM J. Imaging Sci.* 2.1, pp. 183–202.

Bello, I., W. Fedus, X. Du, E. D. Cubuk, A. Srinivas, T.-Y. Lin, J. Shlens, and B. Zoph (2021). "Revisiting ResNets: Improved Training and Scaling Strategies". In: *Advances in Neural Information Processing Systems*.

Bergstra, J. and Y. Bengio (2012). "Random search for hyper-parameter optimization". In: *J. Mach. Learn. Res.* 13.2.

Bergstra, J., D. Yamins, and D. Cox (17–19 Jun 2013). "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research 1. Atlanta, Georgia, USA: PMLR, pp. 115–123.

Bertin-Mahieux, T., D. P. Ellis, B. Whitman, and P. Lamere (2011). "The Million Song Dataset". In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*.

Bertrand, Q., Q. Klopfenstein, P.-A. Bannier, G. Gidel, and M. Massias (2022). "Beyond L1: Faster and Better Sparse Models with skglm". In: *arXiv preprint arXiv:2204.07826*.

Bertsekas, D. P. (1999). *Nonlinear programming*. Athena Scientific.

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1, pp. 65–98.

Bleakley, K. and J.-P. Vert (2011). "The group fused Lasso for multiple change-point detection". In.

Blondel, M. and F. Pedregosa (2016). *Lightning: Large-Scale Linear Classification, Regression and Ranking in Python*.

Boisbunon, A., R. Flamary, and A. Rakotomamonjy (2014). "Active set strategy for high-dimensional non-convex sparse optimization problems". In: *ICASSP*. IEEE, pp. 1517–1521.

Bolte, J., S. Sabach, and M. Teboulle (2014). "Proximal alternating linearized minimization for nonconvex and nonsmooth problems". In: *Mathematical Programming* 146.1, pp. 459–494.

Bonnefoy, A., V. Emiya, L. Ralaivola, and R. Gribonval (2015). "Dynamic screening: accelerating first-order algorithms for the Lasso and Group-Lasso". In: *IEEE Trans. Signal Process.* 63.19, p. 20.

Bottou, L. (2010). "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *COMPSTAT*. Physica-Verlag, pp. 177–186.

Boyd, S., N. Parikh, E. Chu, B. Peleato, and J. Eckstein (2011). "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3.1.

Boyd, S. P. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge, UK ; New York: Cambridge University Press.

Boykov, Y., O. Veksler, and R. Zabih (2001). "Fast approximate energy minimization via graph cuts". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.11, pp. 1222–1239.

Brandl, G. (2010). "Sphinx documentation". In: *URL http://sphinx-doc. org/sphinx. pdf*.

Bredies, K., K. Kunisch, and T. Pock (2010). "Total generalized variation". In: *SIAM J. Imaging Sci.* 3.3, pp. 492–526.

Breheny, P. and J. Huang (2011). "Coordinate descent algorithms for nonconvex penalized regression, with applications to biological feature selection". In: *Ann. Appl. Stat.* 5.1, p. 232.

Brock, A., S. De, S. L. Smith, and K. Simonyan (2021). "High-performance large-scale image recognition without normalization". In: *ICML*. PMLR, pp. 1059–1071.

Bühlmann, P. and S. van de Geer (2011). *Statistics for high-dimensional data*. Springer Series in Statistics. Methods, theory and applications. Heidelberg: Springer.

Candès, E. J., M. B. Wakin, and S. P. Boyd (2008). "Enhancing Sparsity by Reweighted l_1 Minimization". In: *J. Fourier Anal. Applicat.* 14.5-6, pp. 877–905.

Chambolle, A. and P.-L. Lions (1997). "Image recovery via total variation minimization and related problems". In: *Numerische Mathematik* 76.2, pp. 167–188.

Chambolle, A. and T. Pock (2011). "A first-order primal-dual algorithm for convex problems with applications to imaging". In: *Journal of Mathematical Imaging and Vision* 40.1.

Chen, S. S., D. L. Donoho, and M. A. Saunders (1998). "Atomic decomposition by basis pursuit". In: *SIAM J. Sci. Comput.* 20.1, pp. 33–61.

Cherkaoui, H., T. Moreau, A. Halimi, and P. Ciuciu (2019). "Sparsity-Based Semi-Blind Deconvolution of Neural Activation Signal in fMRI". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Brighton, UK.

Chollet, F. et al. (2015). *Keras*.

Clark, A. (2015). *Pillow (PIL Fork) Documentation*.

Combettes, P. L. and V. Wajs (2005). "Signal recovery by proximal forward-backward splitting". In: *Multiscale modeling & simulation* 4.4, pp. 1168–1200.

Combettes, P. L. and L. E. Glaudin (2021). "Solving Composite Fixed Point Problems with Block Updates". In: *Advances in Nonlinear Analysis*.

Condat, L. (2013a). "A Direct Algorithm for 1-D Total Variation Denoising". In: *IEEE SIGNAL PROCESSING LETTERS* 20.12.

Condat, L. (2013b). "A primal-dual splitting method for convex optimization involving Lipschitzian, proximable and linear composite terms". In: *Journal of Optimization Theory and Applications, Springer Verlag*.

Criteo-Labs (2015). "Criteo releases industry's largest-ever dataset for machine learning to academic community". In.

Daubechies, I., M. Defrise, and C. De Mol (2004). "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint". In: *Commun. Pure Appl. Math.* 57.11, pp. 1413–1457.

Defazio, A., F. Bach, and S. Lacoste-Julien (2014). "SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives". In: *Advances in Neural Information Processing Systems*. Vol. 28, pp. 1646–1654.

Dembo, R. S., S. C. Eisenstat, and T. Steihaug (1982). "Inexact Newton Methods". In: *SIAM J. Numer. Anal.* 19.2, pp. 400–408.

Dosovitskiy, A., L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby (2021). "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *ICLR*.

Dünner, C., T. Parnell, D. Sarigiannis, N. Ioannou, A. Anghel, G. Ravi, M. Kandasamy, and H. Pozidis (2018). "Snap ML: A hierarchical framework for machine learning". In: *Advances in Neural Information Processing Systems*. Vol. 31.

Efron, B., T. J. Hastie, I. M. Johnstone, and R. Tibshirani (2004). "Least angle regression". In: *Ann. Statist.* 32.2. With discussion, and a rejoinder by the authors, pp. 407–499.

El Ghaoui, L., V. Viallon, and T. Rabbani (2012). "Safe feature elimination in sparse supervised learning". In: *J. Pacific Optim.* 8.4, pp. 667–698.

Elad, M., P. Milanfar, and R. Rubinstein (2006). "Analysis versus synthesis in signal priors". In: *2006 14th European Signal Processing Conference*.

Falcon, W., J. Borovec, A. Wälchli, N. Eggert, J. Schock, J. Jordan, N. Skafte, Ir1dXD, V. Bereznyuk, E. Harris, T. Murrell, P. Yu, S. Præsius, T. Addair, J. Zhong, D. Lipin, S. Uchida, S. Bapat, H. Schröter, B. Dayma, A. Karnachev, A. Kulkarni, S. Komatsu, Martin.B, J.-B. SCHIRATTI, H. Mary, D. Byrne, C. Eyzaguirre, cinjon, and A. Bakhtin (May 2020). *PyTorchLightning/pytorch-lightning: 0.7.6 release*. Version 0.7.6.

Fan, J. and R. Li (2001). "Variable selection via nonconcave penalized likelihood and its oracle properties". In: *J. Amer. Statist. Assoc.* 96.456, pp. 1348–1360.

Fan, R.-E., K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin (2008). "LIBLINEAR: A Library for Large Linear Classification". In: *J. Mach. Learn. Res.* 9.

Farrens, S., A. Grigis, L. El Gueddari, Z. Ramzi, G. Chaithya, S. Starck, B. Sarthou, H. Cherkaoui, P. Ciuciu, and J.-L. Starck (2020). "PySAP: Python Sparse Data Analysis Package for multidisciplinary image processing". In: *Astronomy and Computing* 32, p. 100402.

Fletcher, R. and C. M. Reeves (Jan. 1964). "Function Minimization by Conjugate Gradients". In: *The Computer Journal* 7.2, pp. 149–154.

Forde, J., T. Head, C. Holdgraf, Y. Panda, G. Nalvarete, B. Ragan-Kelley, and E. Sundell (2018). *Reproducible research environments with repo2docker*. Tech. rep.

Friedman, J., T. Hastie, and R. Tibshirani (2010). "Regularization paths for generalized linear models via coordinate descent". In: *J. Stat. Softw.* 33.1, pp. 1–22.

Gao, F. and L. Han (2012). "Implementing the Nelder-Mead simplex algorithm with adaptive parameters". In: *Computational Optimization and Applications* 51.1, pp. 259–277.

Glorot, X. and Y. Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *AISTATS*. Vol. 9, pp. 249–256.

Golub, T. R., D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri, et al. (1999). "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring". In: *science* 286.5439, pp. 531–537.

Gong, P., C. Zhang, Z. Lu, J. Huang, and J. Ye (2013). "A general iterative shrinkage and thresholding algorithm for non-convex regularized optimization problems". In: *ICML*, pp. 37–45.

Gramfort, A., M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, L. Parkkonen, and M. S. Hämäläinen (2014). "MNE software for processing MEG and EEG data". In: *NeuroImage* 86, pp. 446–460.

Grandvalet, Y. (1998). "Least absolute shrinkage is equivalent to quadratic penalization". In: *International Conference on Artificial Neural Networks*. Springer, pp. 201–206.

Guyon, I., S. Gunn, A. Ben-Hur, and G. Dror (2004). "Result analysis of the nips 2003 feature selection challenge". In: *Advances in neural information processing systems* 17.

Hansen, N., A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff (2021). "COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting". In: *Optimization Methods and Software* 36.1. ArXiv e-prints, arXiv:1603.08785, pp. 114–144.

Hansen, N. and A. Ostermeier (2001). "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary computation* 9.2, pp. 159–195.

Harris, C. R., K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. (2020). "Array programming with NumPy". In: *Nature* 585.7825, pp. 357–362.

Hastie, T. J., R. Tibshirani, and M. Wainwright (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.

He, K., X. Zhang, S. Ren, and J. Sun (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *CVPR*, pp. 1026–1034.

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep residual learning for image recognition". In: *CVPR*.

He, T., Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li (2019). "Bag of tricks for image classification with convolutional neural networks". In: *CVPR*. Vol. 2019-June, pp. 558–567.

Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95.

Hutson, M. (2018). "Artificial intelligence faces reproducibility crisis". In: *Science* 359.6377, pp. 725–726.

Inc., P. T. (2015). *Collaborative data science*. URL: https://plot.ly.

Ioffe, S. and C. Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *ICML*, pp. 448–456.

Johnson, R. and T. Zhang (2013). "Accelerating Stochastic Gradient Descent Using Predictive Variance Reduction". In: *Advances in Neural Information Processing Systems*. Vol. 26.

Johnson, T. B. and C. Guestrin (2015). "Blitz: A Principled Meta-Algorithm for Scaling Sparse Optimization". In: *ICML*. Vol. 37, pp. 1171–1179.

Karahanoğlu, F. I., C. Caballero-Gaudes, F. Lazeyras, and D. Van De Ville (June 2013). "Total Activation: fMRI Deconvolution through Spatio-Temporal Regularization". In: *NeuroImage* 73, pp. 121–134.

Keerthi, S. S., D. DeCoste, and T. Joachims (2005). "A modified finite Newton method for fast solution of large scale linear SVMs." In: *Journal of Machine Learning Research* 6.3.

Kiefer, J. and J. Wolfowitz (1952). "Stochastic estimation of the maximum of a regression function". In: *Ann. Math. Stat.*, pp. 462–466.

Kingma, D. P. and J. Ba (2015). "Adam: A Method for Stochastic Optimization". In: *ICLR*, pp. 1–10.

Koh, K., J. Kim, and S. Boyd (2007). "An interior-point method for large-scale l1-regularized logistic regression." In: *J. Mach. Learn. Res.* 8.8, pp. 1519–1555.

Kolmogorov, V. and R. Zabin (2004). "What energy functions can be minimized via graph cuts?" In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.2, pp. 147–159.

Komodakis, N. and J.-C. Pesquet (2015). "Playing with Duality: An overview of recent primal-dual approaches for solving large-scale optimization problems". In: *IEEE Signal Processing Magazine, Institute of Electrical and Electronics Engineers*.

Kornblith, S. (Oct. 28, 2021). *Lasso.jl*. Version 0.6.3. JuliaStats.

Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images*. Tech. rep.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). "Imagenet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. South Lake Tahoe, NV, USA, pp. 1097–1105.

Lalanne, C., M. Rateaux, L. Oudre, M. P. Robert, and T. Moreau (July 2020). "Extraction of Nystagmus Patterns from Eye-Tracker Data with Convolutional Sparse Coding". In: *Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*. Montreal, QC, Canada: IEEE, pp. 928–931.

LeCun, Y., C. Cortes, and C. Burges (2010). "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2.

Lewis, D. D., Y. Yang, T. Russell-Rose, and F. Li (2004). "Rcv1: A new benchmark collection for text categorization research". In: *Journal of machine learning research* 5.Apr, pp. 361–397.

Liang, J., T. Luo, and C.-B. Schöenlieb (2022). "Improving "Fast Iterative Shrinkage-Thresholding Algorithm": Faster, Smarter, and Greedier". In: *SIAM J. Sci. Comput.* 44.3, A1069–A1091.

Liu, D. C. and J. Nocedal (Aug. 1989). "On the Limited Memory BFGS Method for Large Scale Optimization". In: *Math. Program.* 45.1-3, pp. 503–528.

Liu, Z., H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie (2022). "A ConvNet for the 2020s". In: *CVPR*.

Loshchilov, I. and F. Hutter (2017). "SGDR: Stochastic Gradient Descent with Warm Restarts". In: *ICLR*.

Loshchilov, I. and F. Hutter (2019). "Decoupled Weight Decay Regularization". In: *ICLR*.

Lueckmann, J.-M., J. Boelts, D. S. Greenberg, P. J. Gonçalves, and J. H. Macke (2021). "Benchmarking Simulation-Based Inference". In: *AISTATS*. Vol. 130. PMLR, pp. 343–351.

Mairal, J. and B. Yu (2012). "Complexity analysis of the Lasso regularization path". In: *ICML*, pp. 353–360.

Mairal, J. (2019). "Cyanure: An open-source toolbox for empirical risk minimization for python, c++, and soon more". In: *arXiv preprint arXiv:1912.08165*.

Massias, M., A. Gramfort, and J. Salmon (2018). "Celer: a fast solver for the lasso with dual extrapolation". In: *ICML*, pp. 3315–3324.

Mattson, P., V. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, et al. (2020). "MLPerf: An industry standard benchmark suite for machine learning performance". In: *IEEE Micro* 40.2, pp. 8–16.

Mazumder, R., J. H. Friedman, and T. Hastie (2011). "Sparsenet: Coordinate descent with nonconvex penalties". In: *J. Amer. Statist. Assoc.* 106.495, pp. 1125–1138.

McKinney, W. et al. (2010). "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX, pp. 51–56.

Moreau, T., P. Glaser, R. Yurchak, and O. Grisel (June 2017). *Loky*. Version 3.0. URL: https://github.com/joblib/loky.

Nájera, Ó., E. Larson, L. Estève, G. Varoquaux, L. Liu, J. Grobler, E. S. de Andrade, C. Holdgraf, A. Gramfort, M. Jas, J. Nothman, O. Grisel, N. Varoquaux, E. Gouillart, M. Luessi, A. Lee, J. Vanderplas, T. Hoffmann, T. A. Caswell, B. Sullivan, A. Batula, jaeilepp, T. Robitaille, S. Appelhoff, P. Kunzmann, M. Geier, Lars, K. Sunden, D. Stańczak, and A. Y. Shih (May 2020). *sphinx-gallery/sphinx-gallery: Release v0.7.0*. Version v0.7.0.

Ndiaye, E., O. Fercoq, A. Gramfort, and J. Salmon (2017). "Gap Safe screening rules for sparsity enforcing penalties". In: *J. Mach. Learn. Res.* 18.128, pp. 1–33.

Nesterov, Y. E. (1983). "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". In: *Dokl. akad. nauk Sssr*. Vol. 269, pp. 543–547.

Netzer, Y., T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng (2011). "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *Advances in Neural Information Processing Systems*.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*, pp. 8024–8035.

Pedregosa, F., G. Negiar, and G. Dresdner (2020). "copt: composite optimization in Python". In.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. (2011). "Scikit-learn: Machine learning in Python". In: *J. Mach. Learn. Res.* 12, pp. 2825–2830.

Pineau, J., P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and H. Larochelle (2021). "Improving reproducibility in machine learning research: a report from the NeurIPS 2019 reproducibility program". In: *J. Mach. Learn. Res.* 22.

Pineau, J., K. Sinha, G. Fried, R. N. Ke, and H. Larochelle (2019). "ICLR reproducibility challenge 2019". In: *ReScience C* 5.2, p. 5.

Poon, C. and G. Peyré (2021). "Smooth Bilevel Programming for Sparse Regularization". In: *Advances in Neural Information Processing Systems*. Vol. 34, pp. 1543–1555.

Powell, M. J. (1964). "An efficient method for finding the minimum of a function of several variables without calculating derivatives". In: *The computer journal* 7.2, pp. 155–162.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2007). *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.

Prokhorov, D. (2001). "IJCNN 2001 neural network competition". In: *Slide presentation in IJCNN* 1.97, p. 38.

Quemener, E. and M. Corvellec (2013). "SIDUS—the solution for extreme deduplication of an operating system". In: *Linux Journal* 2013.235, p. 3.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria.

Raff, E. (2019). "A step toward quantifying independently reproducible machine learning research". In: *Advances in Neural Information Processing Systems*. Vol. 32, pp. 5486–5496.

Rapin, J. and O. Teytaud (2018). *Nevergrad - A gradient-free optimization platform*. https://GitHub.com/FacebookResearch/Nevergrad.

Raschka, S., J. Patterson, and C. Nolet (2020). "Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence". In: *Information-an International Interdisciplinary Journal* 11.4, p. 193.

Robbins, H. and S. Monro (1951). "A stochastic approximation method". In: *Ann. Math. Stat.*, pp. 400–407.

Rodola, G. (2016). "Psutil package: a cross-platform library for retrieving information on running processes and system utilization". In: *Google Scholar*.

Rougier, N. P. and K. Hinsen (2018). "ReScience C: a journal for reproducible replications in computational science". In: *International Workshop on Reproducible Research in Pattern Recognition*. Springer, pp. 150–156.

Rudin, L. I., S. Osher, and E. Fatemi (1992). "Nonlinear total variation based noise removal algorithms". In: *Physica D: Nonlinear Phenomena* 60.1-4, pp. 259–268.

Sardy, S., A. G. Bruce, and P. Tseng (2000). "Block coordinate relaxation methods for nonparametric wavelet denoising". In: *J. Comput. Graph. Stat.* 9.2, pp. 361–379.

Schmidt, M., N. Le Roux, and F. Bach (2017). "Minimizing Finite Sums with the Stochastic Average Gradient". In: *Math. Program.* 162.arXiv:1309.2388, pp. 83–112.

Schmidt, R. M., F. Schneider, and P. Hennig (2021). "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers". In: *ICML*. Vol. 139. PMLR, pp. 9367–9376.

Sculley, D., J. Snoek, A. Wiltschko, and A. Rahimi (2018). "Winner's curse? On pace, progress, and empirical rigor". In.

Silva, T. S. (2019). "How to Add Regularization to Keras Pre-trained Models the Right Way". In: *https://sthalles.github.io*.

Simonyan, K. and A. Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR*.

Sra, S., S. Nowozin, and S. J. Wright (2012). *Optimization for machine learning*. MIT Press.

Tan, M. and Q. Le (2019). "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *ICML*, pp. 6105–6114.

Tibshirani, R., J. Bien, J. Friedman, T. Hastie, N. Simon, J. Taylor, and R. J. Tibshirani (2012). "Strong rules for discarding predictors in lasso-type problems". In: *J. R. Stat. Soc. Ser. B Stat. Methodol.* 74.2, pp. 245–266.

Tibshirani, R. (1996). "Regression shrinkage and selection via the lasso". In: *J. R. Stat. Soc. Ser. B Stat. Methodol.* 58.1, pp. 267–288.

Tibshirani, R. J. (Feb. 2014). "Adaptive Piecewise Polynomial Estimation via Trend Filtering". In: *The Annals of Statistics* 42.1.

Tibshirani, R. J. and J. Taylor (2011). "The solution path of the generalized lasso". In: *Ann. Statist.* 39.3, pp. 1335–1371.

Tseng, P. (1993). "Dual coordinate ascent methods for non-strictly convex minimization". In: *Math. Program.* 59.1, pp. 231–247.

Vanschoren, J., J. van Rijn, B. Bischl, and L. Torgo (2013). "OpenML: networked science in machine learning". In: *SIGKDD Explorations* 15.2, pp. 49–60.

Virtanen, P., R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17, pp. 261–272.

Wales, D. J. and J. P. Doye (1997). "Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms". In: *The Journal of Physical Chemistry A* 101.28, pp. 5111–5116.

Wightman, R., H. Touvron, and H. Jégou (2021). *ResNet strikes back: An improved training procedure in timm*. Tech. rep., pp. 1–22.

Wright, S. and J. Nocedal (1999). *Numerical Optimization*. Science Springer.

Xie, S., R. Girshick, P. Dollár, Z. Tu, and K. He (2017). "Aggregated residual transformations for deep neural networks". In: *CVPR*, pp. 1492–1500.

Zhang, . (2010a). "Nearly unbiased variable selection under minimax concave penalty". In: *Ann. Statist.* 38.2, pp. 894–942.

Zhang, M., J. Lucas, J. Ba, and G. E. Hinton (2019). "Lookahead optimizer: k steps forward, 1 step back". In: *Advances in Neural Information Processing Systems*. Vol. 32.

Zhang, T. (2010b). "Analysis of multi-stage convex relaxation for sparse regularization". In: *J. Mach. Learn. Res.* 11.Mar, pp. 1081–1107.

Zheng, Y., R. Zhang, and Y. Mao (2021). "Regularizing neural networks via adversarial model perturbation". In: *CVPR*, pp. 8156–8165.

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.
   (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 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. The pipeline presented in Figure 1 could not exist without the libraries mentioned hereafter.

The command line interface and API use the `click`[5], `pyyaml`[6] 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`[7] for calling `R` (R Core Team, 2017) libraries and `PyJulia`[8] for interfacing with `Julia` (Bezanson et al., 2017). The benchmark runs extensively use `joblib` [9], `loky` (Moreau et al., 2017) and `submitit`[10] for parallelization.

The results are stored and processed for visualizations using `pandas` (McKinney et al., 2010), `matplotlib` (Hunter, 2007) for static rendering, `mako` [11] and `plotly` (Inc., 2015) for interactive webpages. The participative results website relies partially on `pygithub`[12].

Our documentation is generated by multiple `sphinx`-based (Brandl, 2010) libraries (`sphinx-bootstrap-theme`[13], `sphinx-click`[14], `sphinx-gallery` (Nájera et al., 2020) and `sphinx-prompt`[15]), and also the `numpydoc`[16] and `pillow` (Clark, 2015) libraries.

MM 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). This work was also granted access to the HPC resources of IDRIS under the allocation 2022-AD011011172R2 and 2022-AD011013570 made by GENCI.

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).

BM contributions were supported by a grant from Digiteo France.

MD contributions were supported by DATAIA convergence institute as part of the "Programme d'Investissement d'Avenir", (ANR-17-CONV-0003) operated by Inria.

BN work was supported by the Télécom Paris's Chaire DSAIDIS (Data Science & Artificial Intelligence for Digitalized Industry Services).

[5] https://github.com/pallets/click
[6] https://github.com/yaml/pyyaml
[7] https://rpy2.github.io
[8] https://pyjulia.readthedocs.io
[9] https://github.com/joblib/joblib
[10] https://github.com/facebookincubator/submitit
[11] https://github.com/sqlalchemy/mako/
[12] https://github.com/PyGithub
[13] https://github.com/ryan-roemer/sphinx-bootstrap-theme
[14] https://github.com/click-contrib/sphinx-click
[15] https://github.com/sbrunner/sphinx-prompt
[16] https://github.com/numpy/numpydoc

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 ℓ_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:

$$\min_{\theta} \frac{1}{2}\|y - X\theta\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2 \ , \tag{4}$$

where $X \in \mathbb{R}^{n \times p}$ is a design matrix, $y \in \mathbb{R}^n$ is the target vector and λ 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 $\theta^* = (X^\top X + \lambda Id)^{-1} X^\top y$ 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 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 θ. 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 .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.

```
from benchopt import BaseDataset
from benchopt import safe_import_context

# This context allow to manipulate the Dataset object even if
# libsvmdata is not installed. It is used in `benchopt install`.
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.

```python
from benchopt import BaseSolver
from benchopt 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

Datasets	References	Samples (n)	Features (p)
leukemia	Golub et al. (1999)	38	7129
bodyfat	Guyon et al. (2004)	252	8
gisette	Guyon et al. (2004)	6000	5000

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

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:

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

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

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

such that the signal-to-noise ratio is $\texttt{snr} = \frac{\|X\theta^*\|_2}{\|\varepsilon\|_2}$.

We use a signal-to-noise ratio `snr` $= 3$, a correlation ρ 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

Solver	References	Short Description	Language
`GD`	Boyd and Vandenberghe (2004)	Gradient Descent	`Python`
`Accelerated GD`	Nesterov (1983)	Gradient Descent + acceleration	`Python`
`scikit-learn[svd]`	Pedregosa et al. (2011)	SVD (Singular Value Decomposition)	`Python (Cython)`
`scikit-learn[cholesky]`	Pedregosa et al. (2011)	Cholesky decomposition	`Python (Cython)`
`scikit-learn[lsqr]`	Pedregosa et al. (2011)	regularized least-squares	`Python (Cython)`
`scikit-learn[saga]`	Pedregosa et al. (2011)	SAGA (Varianced reduced stochastic method)	`Python (Cython)`
`scikit-learn[cg]`	Pedregosa et al. (2011)	Conjugate Gradient	`Python (Cython)`
`CD`	Bertsekas (1999)	Cyclic Coordinate Descent	`Python (Numba)`
`lightning[cd]`	Blondel and Pedregosa (2016)	Cyclic Coordinate Descent	`Python (Cython)`
`snapML[cpu]`	Dünner et al. (2018)	CD	`Python, C++`
`snapML[gpu]`	Dünner et al. (2018)	CD + GPU	`Python, C++`

Results Figure C.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.

B.5 Estimating θ^* for convex problems

When the problem is convex, many solvers are guaranteed to converge to a global minimizer θ^* of the objective function f. To estimate θ^* and $f(\theta^*)$, `Benchopt` approximates θ^* 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.

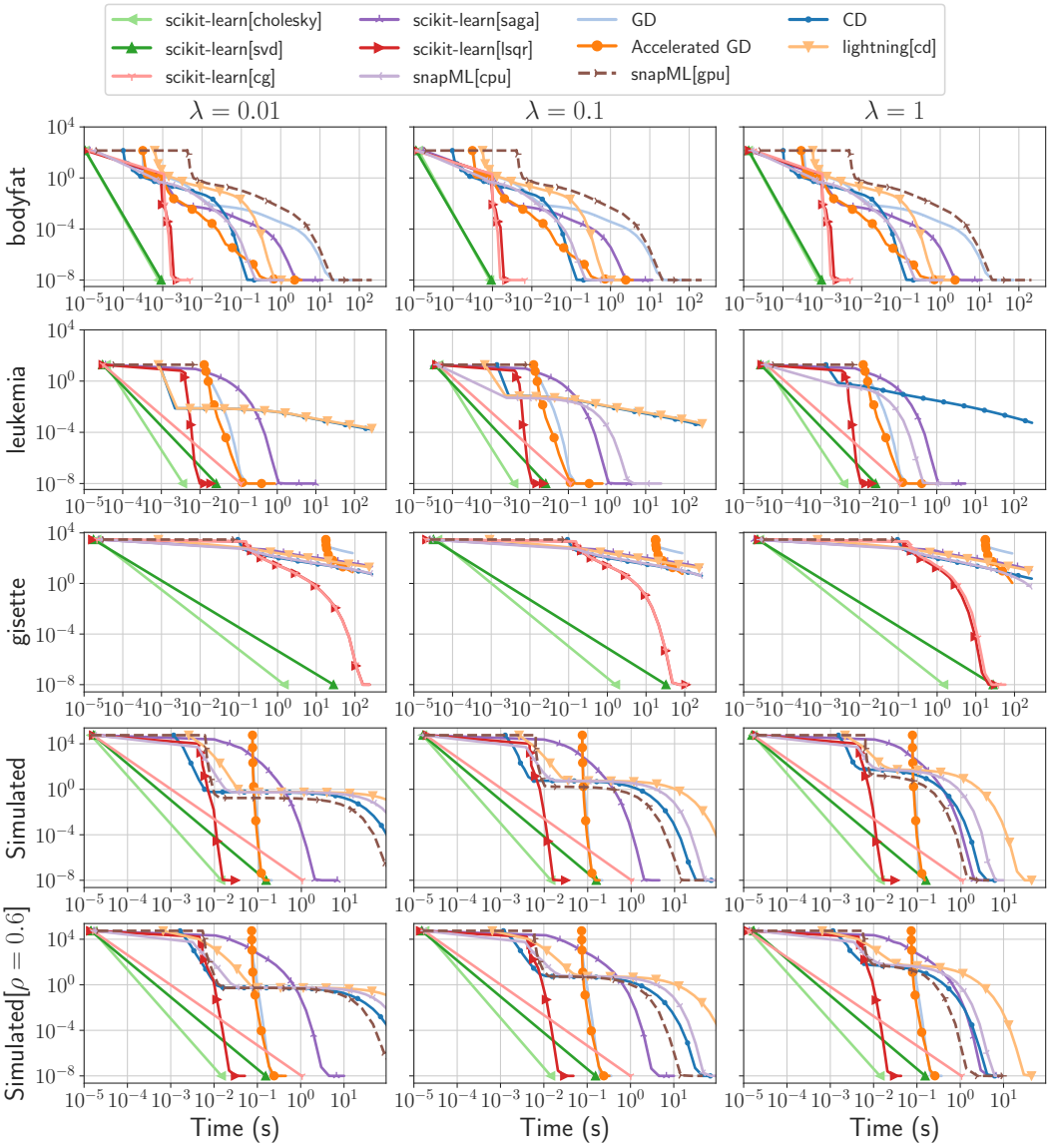

Figure B.1: Benchmark for the Ridge regression, on 10 solvers, 5 datasets (*rows*), and 3 variants of the `Objective` (*columns*) each with a different regularization value $\lambda \in \{0.01, 0.1, 1\}$. The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time.

C ℓ_2-regularized logistic regression

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

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

Table C.1: List of solvers used in the ℓ_2-regularized logistic regression benchmark in Section 3

Solver	References	Short Description	Language
`lightning[sag]`	Blondel and Pedregosa (2016)	SAG	`Python (Cython)`
`lightning[saga]`	Blondel and Pedregosa (2016)	SAGA	`Python (Cython)`
`lightning[cd]`	Blondel and Pedregosa (2016)	Cyclic Coordinate Descent	`Python (Cython)`
`Tick[svrg]`	Bacry et al. (2017)	Stochastic Variance Reduced Gradient	`Python, C++`
`scikit-learn[sgd]`	Pedregosa et al. (2011)	Stochastic Gradient Descent	`Python (Cython)`
`scikit-learn[sag]`	Pedregosa et al. (2011)	SAG	`Python (Cython)`
`scikit-learn[saga]`	Pedregosa et al. (2011)	SAGA	`Python (Cython)`
`scikit-learn[liblinear]`	Pedregosa et al. (2011), Fan et al. (2008)	Truncated Newton Conjugate-Gradient	`Python (Cython)`
`scikit-learn[lbfgs]`	Pedregosa et al. (2011), Virtanen et al. (2020)	L-BFGS (Quasi-Newton Method)	`Python (Cython)`
`scikit-learn[newton-cg]`	Pedregosa et al. (2011), Virtanen et al. (2020)	Truncated Newton Conjugate-Gradient	`Python (Cython)`
`snapml[cpu]`	Dünner et al. (2018)	CD	`Python, C++`
`snapml[gpu]`	Dünner et al. (2018)	CD + GPU	`Python, C++`
`cuML[gpu]`	Raschka et al. (2020)	L-BFGS + GPU	`Python, C++`

Table C.2: List of the datasets used in ℓ_2-regularized logistic regression in Section 3

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

C.2 Results

Figure C.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 λ 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.

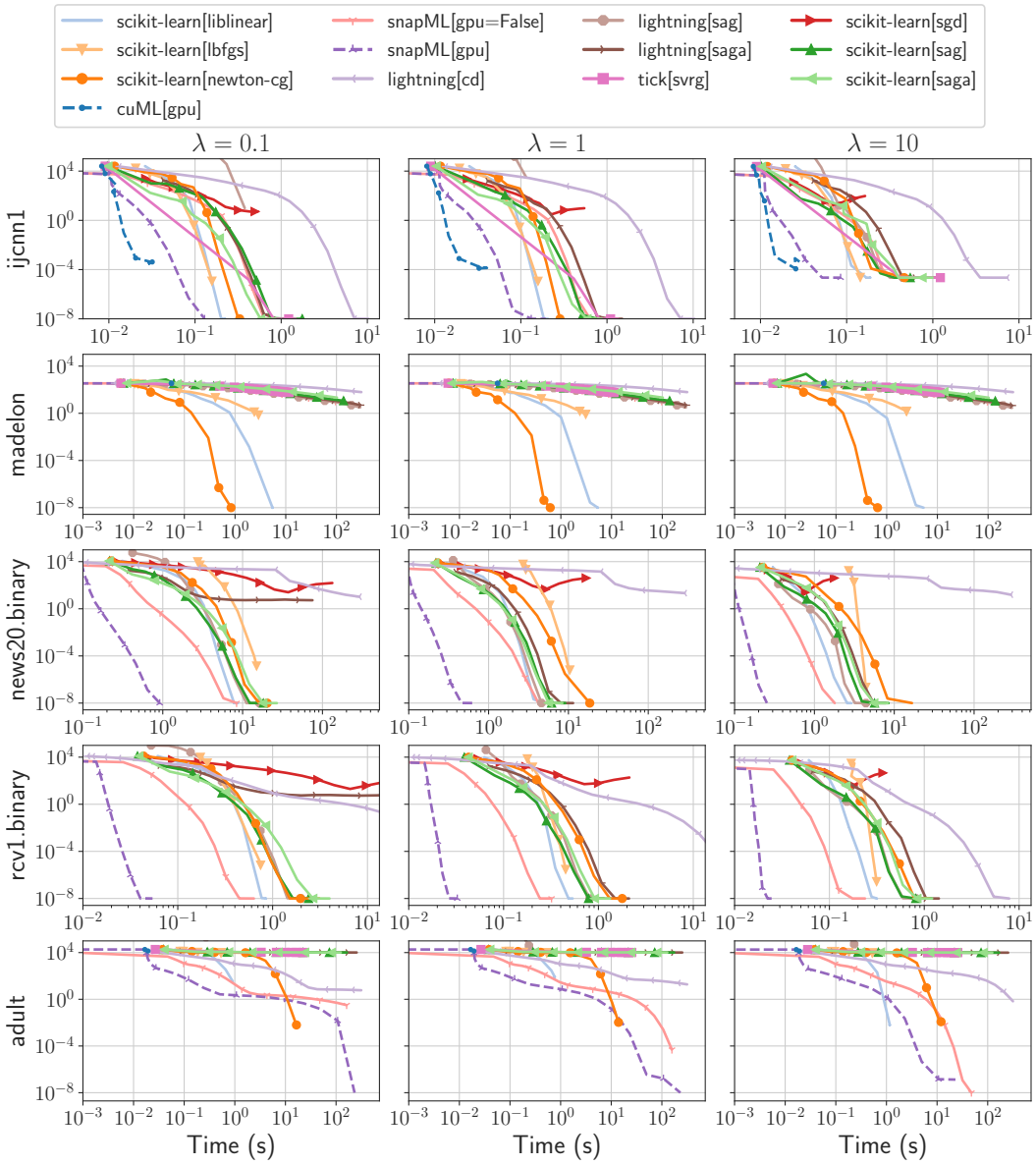

Figure C.1: Additional benchmark for the ℓ_2-regularized logistic regression on variants of the `Objective` (*columns*) with `fit_intercept=False`. The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time. The columns correspond to the objective detailed in Problem (2) with different value of λ: (*first*) $\lambda = 0.1$, (*second*) $\lambda = 1$ and (*third*) $\lambda = 10$.

D Lasso

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

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 Lasso benchmark in Section 4

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

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

Dataset	References	Samples (n)	Features (p)	Density
MEG	Gramfort et al. (2014)	305	7498	1.0
news20	Keerthi et al. (2005)	19 996	1 355 191	3.4×10^{-4}
rcv1	Lewis et al. (2004)	20 242	47 236	3.6×10^{-3}
MillionSong	Bertin-Mahieux et al. (2011)	463 715	90	1

D.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 ℓ_0 pseudonorm of the iterate, allowing to produce Figure D.1 in addition to Figure 4.

D.3 Convergence in terms or iteration

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 D.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`

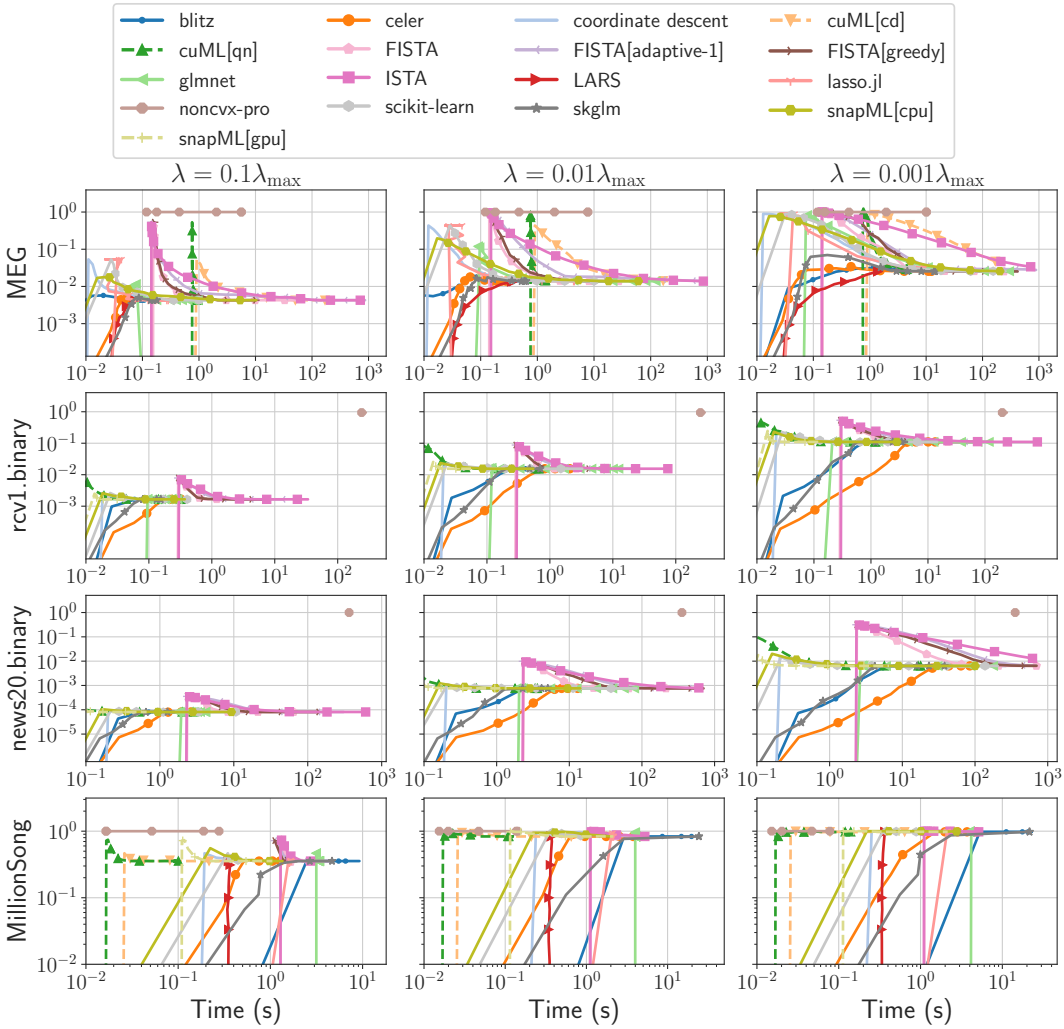

Figure D.1: Additional benchmark for the Lasso on variants of the `Objective` (*columns*). The curves display the fraction of non-zero coefficients in iterates θ_t ($\|\theta_t\|_0/p$), as a function of time.

in `scikit-learn` or in `Numba` lead to identical results, while they outperform proximal gradient methods.

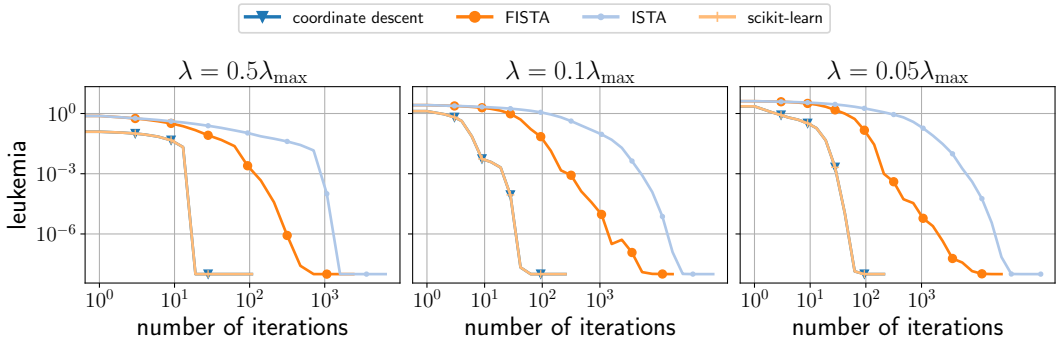

Figure D.2: Convergence speed with respect to the number of iterations for some solvers of the Lasso benchmark on the *leukemia* dataset.

E ResNet18

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

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

In Table E.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 hyperparameters

Table E.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).

Hyperparameter	SGD	Adam
Learning Rate	0.1	0.001
Momentum	0.9	N/A
Weight Decay	5×10^{-4}	0.02
Batch Size	128	128

In Table E.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).

E.2 Aligning TensorFlow and PyTorch ResNet18 training

Table E.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.

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

We summarized in Table E.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 $\texttt{momentum}_{\text{TF}} = 1 - \texttt{momentum}_{\text{PT}}$.
- **Conv2D weights intialization:** TensorFlow/Keras uses the default intialization which is a Glorot uniform intialization (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[17]. Arguably this inconsistency should not play a big role with large batch sizes, but can be significant for smaller batches, especially in deeper

[17] 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.

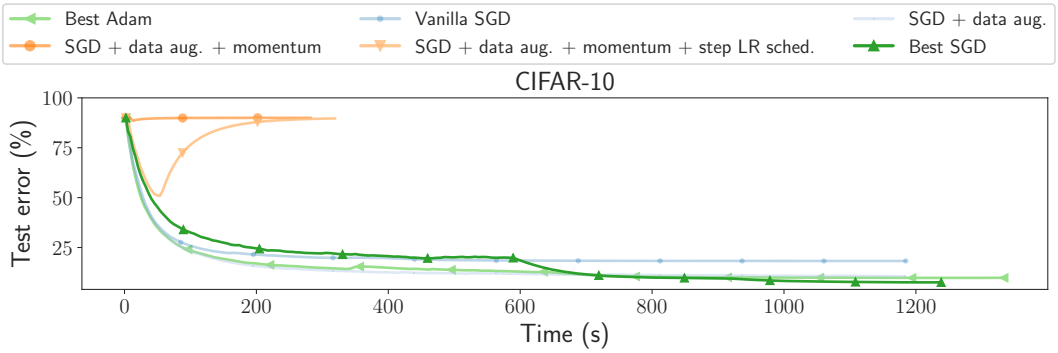

Figure E.1: **VGG16 image classification benchmark with `PyTorch` solvers.** The best SGD configuration features data augmentation, momentum, step learning rate schedule and weight decay.

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).

E.3 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 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 E.1, 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.

F ℓ_1-regularized logistic regression

This additional benchmark is dedicated to ℓ_1-regularized logistic regression, in the same setting as Problem (2) but this time with an ℓ_1-regularization for the parameters of the model:

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

F.1 List of solvers and datasets used in the ℓ_1-regularized logistic regression benchmark

Table F.1 and Table F.2 present the solvers and datasets used in this benchmark.

Table F.1: List of solvers used in the ℓ_1-regularized logistic regression benchmark

Solver	**References**	**Short Description**	**Language**
`blitz`	Johnson and Guestrin (2015)	CD + working set	`Python`, `C++`
`coordinate descent`	Friedman et al. (2010)	(Cyclic) Minimization along coordinates	`Python (Numba)`
`coordinate descent (Newton)`	Friedman et al. (2010)	CD + Newton	`Python (Numba)`
`celer`	Massias et al. (2018)	CD + working set + dual extrapolation	`Python (Cython)`
`copt[FISTA line search]`	Pedregosa et al. (2020), Beck and Teboulle (2009)	FISTA (ISTA + acceleration) + line search	`Python (Cython)`
`copt[PGD]`	Pedregosa et al. (2020) and Combettes and Wajs (2005)	Proximal Gradient Descent	`Python (Cython)`
`copt[PGD linesearch]`	Pedregosa et al. (2020) and Combettes and Wajs (2005)	Proximal Gradient Descent + linesearch	`Python (Cython)`
`copt[saga]`	Pedregosa et al. (2020)	SAGA (Variance reduced stochastic method)	`Python (Cython)`
`copt[svrg]`	Pedregosa et al. (2020)	SVRG (Variance reduced stochastic method)	`Python (Cython)`
`cuML[gpu]`	Raschka et al. (2020)	L-BFGS + GPU	`Python`, `C++`
`cuML[qn]`	Raschka et al. (2020)	Orthant-Wise Limited Memory Quasi-Newton (OWL-QN)	`Python`, `C++`
`cyanure`	Mairal (2019)	Proximal Minimization by Incremental Surrogate Optimization (MISO)	`C++` + `Python` wrapper
`lightning`	Blondel and Pedregosa (2016)	(Cyclic) Coordinate Descent	`Python (Cython)`
`scikit-learn[liblinear]`	Pedregosa et al. (2011) and Fan et al. (2008)	Truncated Newton Conjugate-Gradient	`Python (Cython)`
`scikit-learn[lbfgs]`	Pedregosa et al. (2011) and Virtanen et al. (2020)	L-BFGS (Quasi-Newton Method)	`Python (Cython)`
`scikit-learn[newton-cg]`	Pedregosa et al. (2011) and Virtanen et al. (2020)	Truncated Newton Conjugate-Gradient	`Python (Cython)`
`snapml[gpu=True]`	Dünner et al. (2018)	CD + GPU	`Python`, `C++`
`snapml[gpu=False]`	Dünner et al. (2018)	CD	`Python`, `C++`

Table F.2: List of the datasets used in the ℓ_1-regularized logistic regression benchmark

Datasets	**References**	**Samples (n)**	**Features (p)**	**Density**
gisette	Guyon et al. (2004)	6000	5000	9.9×10^{-1}
colon-cancer	Guyon et al., 2004	62	2000	1.0
news20.binary	Keerthi et al. (2005)	19 996	1 355 191	3.4×10^{-4}
rcv1.binary	Guyon et al., 2004	20 242	19 959	3.6×10^{-3}

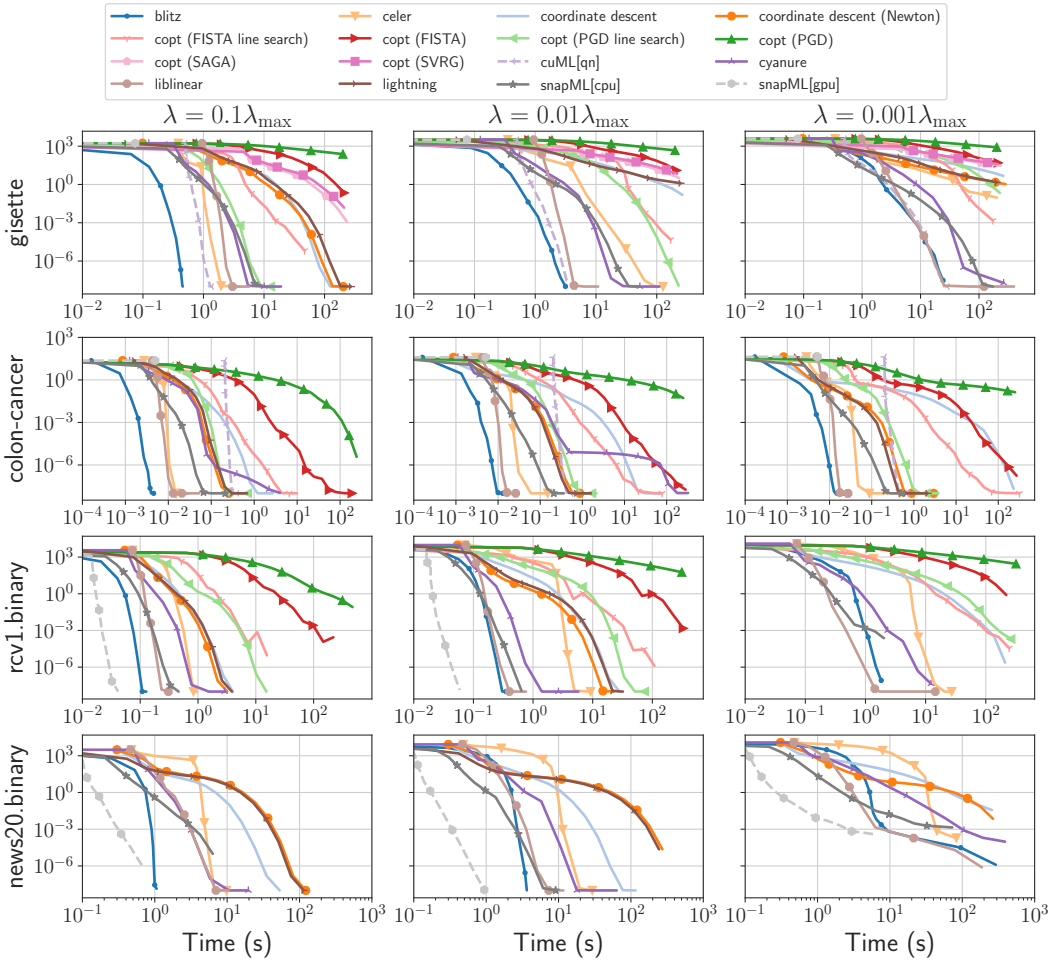

Figure F.1: Benchmark for the ℓ_1-regularized logistic regression on variants of the `Objective` (*columns*). The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time. The first column corresponds to the objective detailed in Problem (6) with $\lambda = 0.1\|X^\top y\|_\infty/2$, the second one with $\lambda = 0.01\|X^\top y\|_\infty/2$ and the third column with $\lambda = 0.001\|X^\top y\|_\infty/2$.

F.2 Results

The results of the ℓ_1-regularized logistic regression benchmark are in Figure F.1.

G 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

$$\theta^* \in \underset{\theta \in \mathbb{R}^p}{\arg\min}\, F(y, X\theta) + \lambda \left\| D\theta \right\|_1 \quad , \tag{7}$$

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 ℓ^2-loss $F(y, z) = \frac{1}{2} \left\| y - z \right\|_2^2$, 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 $\theta \mapsto \left\| D\theta \right\|_1$ 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 trianglar matrix representing an integral operator (cumulative sum), the problem is equivalent to a Lasso problem, and ρ^* 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 $u^* = Lz^* + \rho^*$.

Table G.1 details the different algorithms used in this benchmark.

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

Solver	**References**	**Formulation**	**Short Description**
`ADMM`	Boyd et al. (2011)	Analysis	Primal-Dual Augmented Lagrangian
`ChambollePock`	Chambolle and Pock (2011)	Analysis	Primal-Dual Hybrid Gradient
`CondatVu`	Condat (2013b)	Analysis	Primal-Dual Hybrid Gradient
`DPGD`	Komodakis and Pesquet (2015)	Analysis	Dual proximal GD (+ acceleration)
`PGD`	Condat (2013a)	Analysis	Proximal GD + taut–string
	Barbero and Sra (2018)		ProxTV (+ acceleration)
`celer`	Massias et al. (2018)	Synthesis	CD + working set (lasso) *only for ℓ_2 data-fit*
`FP`	Combettes and Glaudin (2021)	Synthesis	Fixed point with block updates
`ISTA`	Daubechies et al. (2004)	Synthesis	Proximal GD (+ acceleration)
`skglm`	Bertrand et al. (2022)	Synthesis	CD + working set

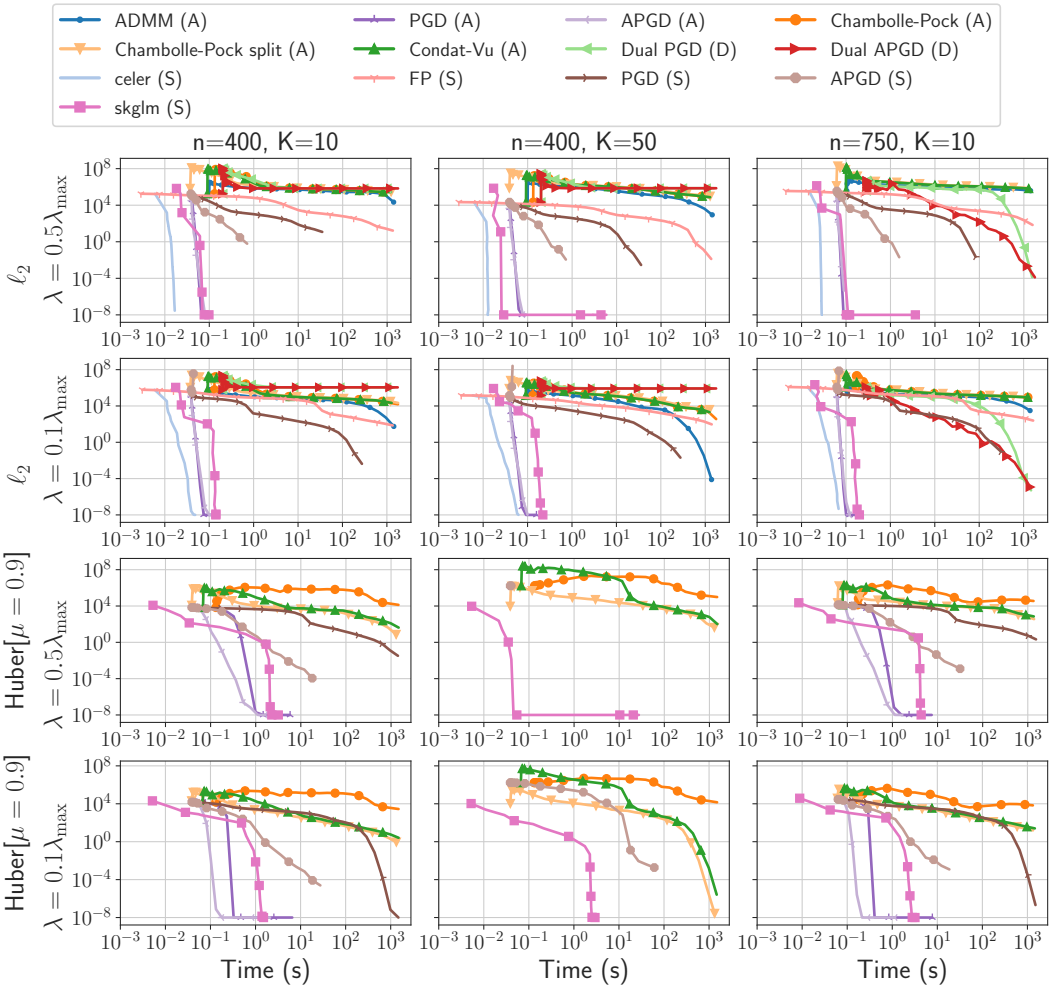

Figure G.1: Benchmark for the TV-regularized regression regression, on 13 solvers, 4 variants of the `Objective` (*rows*), and 3 configurations for a simulated dataset (*columns*). The curves display the suboptimality of the iterates, $f(\theta^t) - f(\theta^*)$, as a function of time. The solvers in this benchmark showcase the three resolution approaches with the Analysis (A), Dual (D) and Synthesis (S) formulations.

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 $\bar{\theta} \in \mathbb{R}^p$ is generated by sampling first 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, $\bar{\theta}$ is obtained by discrete integration as $\bar{\theta}_i = \sum_{k=1}^{i} z_k$ 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 $y = X\bar{\theta} + \epsilon$, 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 G.1 shows that the solvers using the synthesis formulation and coordinate descent-based solvers for the Lasso (ℓ_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).

H 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-ℓ_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:

$$\theta^* \in \underset{\theta \in \mathbb{R}^p}{\arg\min} \, \frac{1}{2n} \left\| y - X\theta \right\|^2 + \sum_{j=1}^{p} \rho_{\lambda,\gamma}(\theta_j) \ , \tag{8}$$

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:

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

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 H.1.

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

Solver	**References**	**Short Description**
`CD`	Breheny and Huang (2011), Mazumder et al. (2011)	Proximal coordinate descent
`PGD`	Bolte et al. (2014)	Proximal operator + Gradient descent
`GIST`	Gong et al. (2013)	Proximal gradient + Barzilai-Borwein rule
`WorkSet CD`	Boisbunon et al. (2014)	Coordinate descent + working set
`skglm`	Bertrand et al. (2022)	Accelerated coordinate descent + Working set

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 $\texttt{snr} = 3$, a correlation $\rho = 0.6$ with $n = 500$ observations and $p = 2000$ features.

H.1 Results

The result of the benchmark is presented in Figure H.1 The problem is non-convex and solvers are only guaranteed to converge to local minima; hence in Figure H.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.

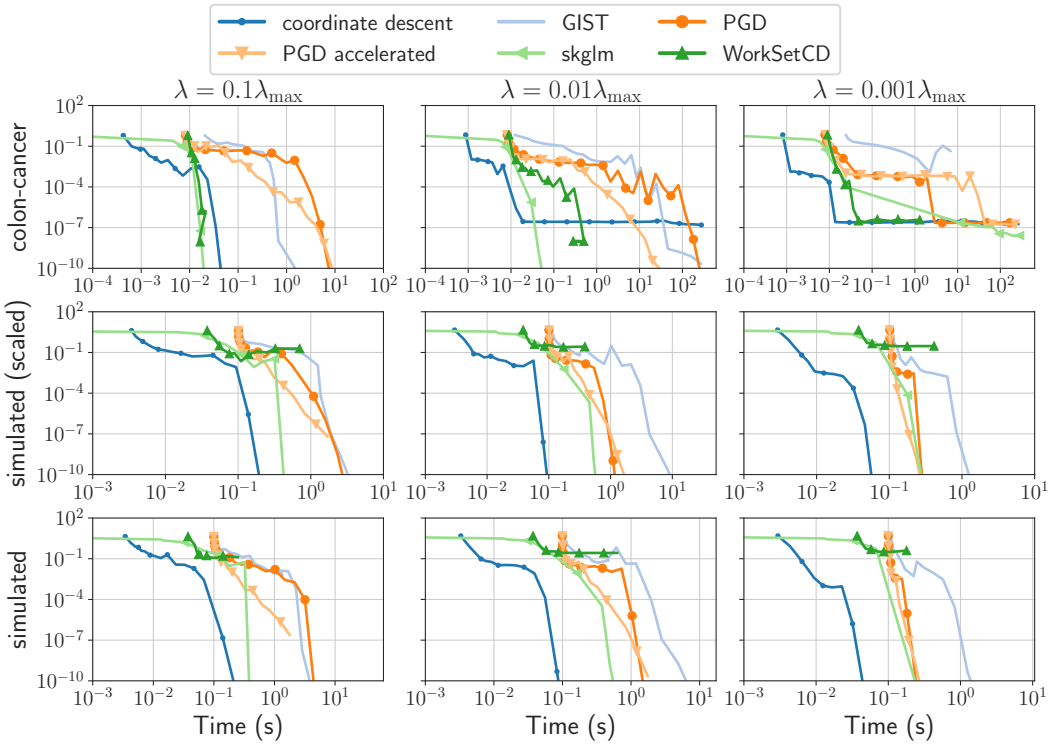

Figure H.1: Benchmark for the MCP regression on variants of the `Objective` (*columns*). The curves display the violation of optimality conditions, $\mathrm{dist}(-X^\top(X\theta_t - y)/n, \partial\rho_{\lambda,\gamma}(\theta_t))$, as a function of time. γ is set to 3, and λ is parameterized as a fraction of the Lasso's $\lambda_{\max}$, $\left\|X^\top y\right\|_\infty / n$.

I Zero-order optimization with on standard

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 `https://github.com/Y1fanHE/PyBenchFCN/`. In particular, among the 61 functions of interest we present here (see Figure I.1) a benchmark for three functions, defined for any $\mathbf{x} = (x_1, \dots, x_N) \in \mathbb{R}^N$:

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \qquad \text{(Rosenbrock)}$$

$$f(\mathbf{x}) = 10 \cdot N + \sum_{i=1}^{N} [(x_i^2 - 10 \cdot \cos(2\pi x_i)] \qquad \text{(Rastrigin)}$$

$$f(\mathbf{x}) = -20 \cdot \exp\left[-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{N} x_i^2}\right] - \exp\left[\frac{1}{d}\sum_{i=1}^{N}\cos(2\pi x_i)\right] + e + 20 \qquad \text{(Ackley)} \ .$$

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

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

Solver	**References**	**Short Description**
`Basin-hopping`	Wales and Doye (1997) and Virtanen et al. (2020)	Two-phase method: global step + local min.
`Nevergrad-RandomSearch`	Rapin and Teytaud (2018) and Bergstra and Bengio (2012)	Sampler by random search
`Nevergrad-CMA`	Rapin and Teytaud (2018) and Hansen and Ostermeier (2001)	CMA evolutionary strategy
`Nevergrad-TwoPointsDE`	Rapin and Teytaud (2018)	Evolutionary strategy
`Nevergrad-NGOpt`	Rapin and Teytaud (2018)	Adaptive evolutionary algorithm
`Nelder-Mead`	Gao and Han (2012) and Virtanen et al. (2020)	Direct search (downhill simplex)
`BFGS`	Virtanen et al. (2020)	BFGS with finite differences
`Powell`	Powell (1964) and Virtanen et al. (2020)	Conjugate direction method
`optuna-TPE`	Akiba et al. (2019) and Bergstra et al. (2013)	Sampler by Tree Parzen Estimation (TPE)
`optuna-CMA`	Akiba et al. (2019) and Hansen and Ostermeier (2001)	CMA evolutionary strategy

I.1 Results

The result of the benchmark is presented in Figure I.1. The functions are non-convex and solvers are only guaranteed to converge to local minima; hence in Figure I.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 implementation from `nevergrad` consistently reaches the global

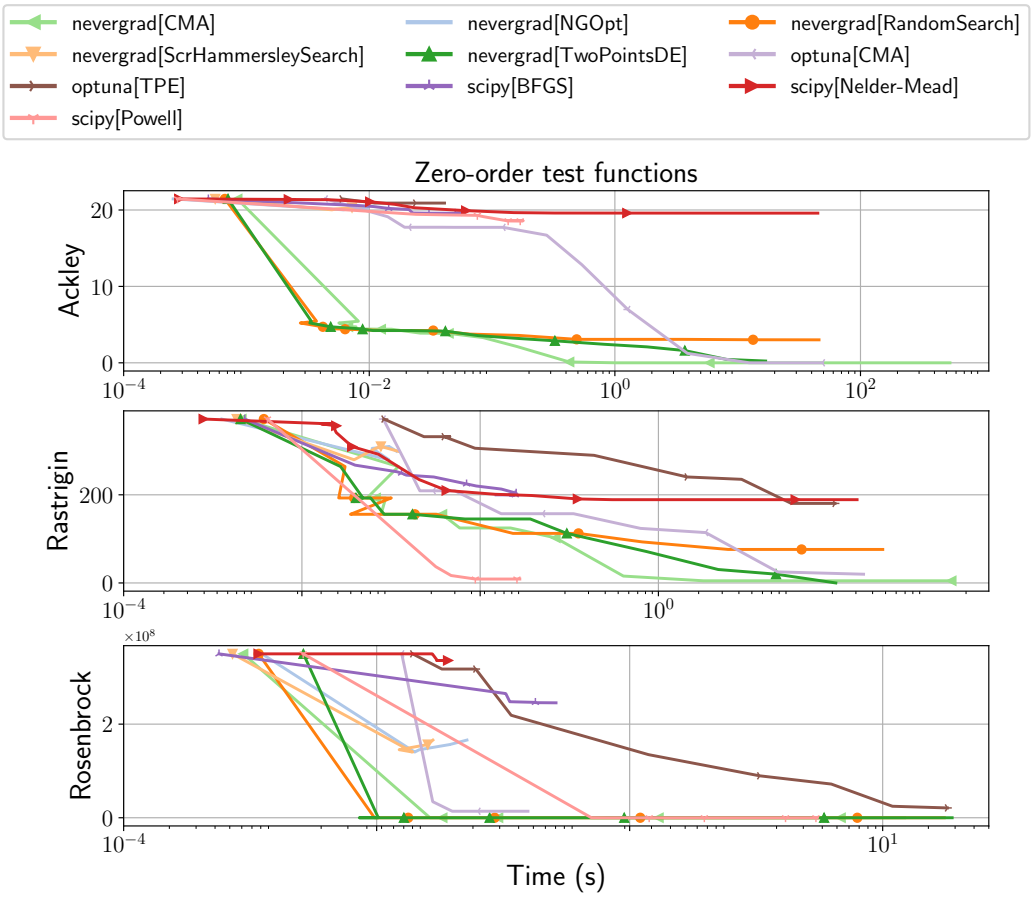

Figure I.1: Benchmark for the zero-order optimization on the Ackley, Rosenbrock and Rastrigin functions in dimension $N = 20$.

minimum. One can also observe that 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.