Unit Tests for Stochastic Optimization

Tom Schaul **Ioannis Antonoglou** **David Silver**

DeepMind Technologies
130 Fenchurch Street, London, UK
{tom,ioannis,david}@deepmind.com

Abstract

Optimization by stochastic gradient descent is an important component of many large-scale machine learning algorithms. A wide variety of such optimization algorithms have been devised; however, it is unclear whether these algorithms are robust and widely applicable across many different optimization landscapes. In this paper we develop a collection of *unit tests* for stochastic optimization. Each unit test rapidly evaluates an optimization algorithm on a small-scale, isolated, and well-understood difficulty, rather than in real-world scenarios where many such issues are entangled. Passing these unit tests is not sufficient, but absolutely necessary for any algorithms with claims to generality or robustness. We give initial quantitative and qualitative results on numerous established algorithms. The testing framework is open-source, extensible, and easy to apply to new algorithms.

1 Introduction

Stochastic optimization [1] is among the most widely used components in large-scale machine learning, thanks to its linear complexity, efficient data usage, and often superior generalization [2, 3, 4]. In this context, numerous variants of stochastic gradient descent have been proposed, in order to improve performance, robustness, or reduce tuning effort [5, 6, 7, 8, 9]. These algorithms may derive from simplifying assumptions on the optimization landscape [10], but in practice, they tend to be used as general-purpose tools, often outside of the space of assumptions their designers intended. The troublesome conclusion is that practitioners find it difficult to discern where potential weaknesses of new (or old) algorithms may lie [11], and when they are applicable – an issue that is separate from raw performance. This results in essentially a trial-and-error procedure for finding the appropriate algorithm variant and hyper-parameter settings, every time that the dataset, loss function, regularization parameters, or model architecture change [12].

The objective of this paper is to establish a collection of benchmarks to evaluate stochastic optimization algorithms and guide algorithm design toward robust variants. Our approach is akin to unit testing, in that it evaluates algorithms on a very broad range of small-scale, isolated, and well-understood difficulties, rather than in real-world scenarios where many such issues are entangled. Passing these unit tests is not sufficient, but absolutely necessary for any algorithms with claims to generality or robustness. This is a similar approach to the very fruitful one taken by the black-box optimization community [13, 14].

The core assumption we make is that stochastic optimization algorithms are acting *locally*, that is, they aim for a short-term reduction in loss given the current noisy gradient information, and possibly some internal variables that capture local properties of the optimization landscape. These local actions include both approaching nearby optima, and navigating slopes, valleys or plateaus that are far from an optimum. The locality property stems from computational efficiency concerns, but it has the additional benefits of minimizing initialization bias and allowing for non-stationary optimization, because properties of the obervation surface observed earlier in the process (and their

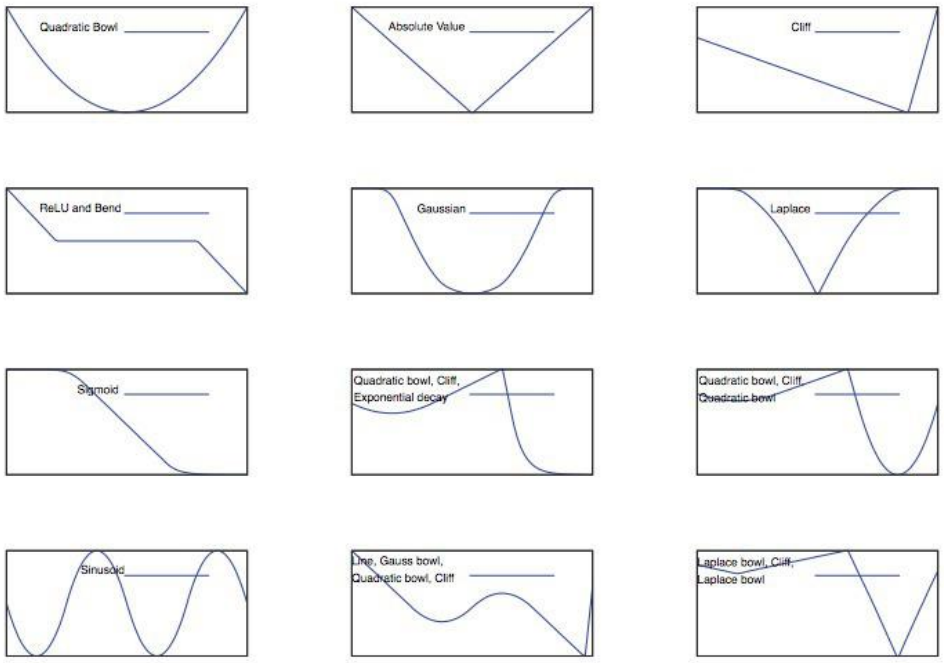

Figure 1: Some one-dimensional shape prototypes. The first six example shapes are atomic prototypes: a quadratic bowl, an absolute value, a cliff with a non differential point after which the derivative increases by a factor ten, a rectified linear shape followed by a bend, an inverse Gaussian, an inverse Laplacian. The next six example shapes are concatenations of atomic prototypes: a sigmoid as a concatenation of a non convex Gaussian a line and an exponential, a quadratic bowl followed by a cliff and then by an exponential function, a quadratic bowl followed by a cliff and another quadratic bowl, a sinusoid as a concatenation of quadratic bowls, a line followed by a Gaussian bowl, a quadratic bowl and a cliff and finally, a Laplace bowl followed by a cliff and another Laplace bowl.

conseuences for the algorithm state) are quickly forgotten. We therefore concentrate on building local unit tests, that investigate algorithm dynamics on a broad range of local scenarios, because we expect that detecting local failure modes will flag an algorithm as unlikely to be robust on more complex tasks – and as a first approximation, optimization on such a complex task can be seen as a sequence of many smaller optimization problems (many of which will not have local optima).

Our divide-and-conquer approach consists of disentangling potential difficulties and testing them in isolation or in simple couplings. Given that our unit tests are small and quick to evaluate, we can have a much larger collection of them, testing hundreds of qualitatively different aspects in less time than it would take to optimize a single traditional benchmark to convergence, thus allowing us to spot and address potential weaknesses early.

Our main contribution is a testing framework, with unit tests designed to test aspects such as: discontinuous or non-differentiable surfaces, curvature scales, various noise conditions and outliers, saddle-points and plateaus, cliffs and asymmetry, and curl and bootstrapping. It also allows test cases to be concatenated by chaining them in a temporal series, or by combining them into multi-dimensional unit tests (with or without variable coupling). We give initial quantitative and qualitative results on a number of established algorithms.

We do not expect this to replace traditional benchmark domains that are closer to the real-world, but to complement it in terms of breadth and robustness. We have tried to keep the framework general and extendable, in the hope it will further grow in diversity, and help others in doing robust algorithm design.

2 Unit test Construction

Our testing framework is an open-source library containing a collection of unit tests and visualization tools. Each *unit test* is defined by a prototype function to be optimized, a prototypical scale, a noise prototype, and optionally a non-stationarity prototype. A *prototype function* is the concatenation of one or more local shape prototypes. A multi-dimensional unit test is a composition of one-dimensional unit tests, optionally with a rotation prototype or curl prototype.

2.1 Shape Prototypes

Shape prototypes are functions defined on an interval, and our collection includes linear slopes (zero curvature), quadratic curves (fixed curvature), convex or concave curves (varying curvature), and curves with exponentially increasing or decreasing slope. Further, there are a number of non-differentiable local shape prototypes (absolute value, rectified-linear, cliff). All of these occur in realistic learning scenarios, for example in logistic regression the loss surface is part concave and part convex, an MSE loss is the prototypical quadratic bowl, but then regularization such as L1 introduces non-differentiable bends (as do rectified-linear or maxout units in deep learning [15, 16]). Steep cliffs in the loss surface are a common occurrence when training recurrent neural networks, as discussed in [11]. See the top rows of Figure 1 for some examples of shape prototypes.

2.2 One-dimensional Concatenation

In our framework, we can chain together a number of shape prototypes, in such a way that the resulting function is continuous and differentiable at all junction points. We can thus produce many prototype functions that closely mimic existing functions, e.g., the Laplace function, sinusoids, saddle-points, step-functions, etc. See the bottom rows of Figure 1 for some examples.

A single *scale* parameter determines the scaling of a concatenated function across all its shapes using the junction constraints. Varying the scales is an important aspect of testing robustness because it is not possible to guarantee well-scaled gradients without substantial overhead. In many learning problems, effort is put into proper normalization [17], but that is insufficient to guarantee homogeneous scaling, for example throughout all the layers of a deep neural network.

2.3 Noise Prototypes

The distinguishing feature of stochastic gradient optimization (compared to batch methods) is that it relies on sample gradients (coming from a subset of even a single element of the dataset) which are inherently noisy. In out unit tests, we model this by four types of stochasticity:

- Scale-independent additive Gaussian noise on the gradients, which is equivalent to random translations of inputs in a linear model with MSE loss. Note that this type of noise flips the sign of the gradient near the optimum and makes it difficult to approach precisely.
- Multiplicative (scale-dependent) Gaussian noise on the gradients, which multiplies the gradients by a positive random number (signs are preserved). This corresponds to a learning scenario where the loss curvature is different for different samples near the current point.
- Additive zero-median Cauchy noise, mimicking the presence of outliers in the dataset.
- Mask-out noise, which zeros the gradient (independently for each dimension) with a certain probability. This mimics both training with drop-out [18], and scenarios with rectified linear units where a unit will be inactive for some input samples, but not for others.

For the first three, we can vary the noise scale, while for mask-out we pick a drop-out frequency. This noise is not necessarily unbiased (as in the Cauchy case), breaking common assumptions made in algorithm design (but the modifications in section 2.5 are even worse). See Figure 2 for an illustration of the first two noise prototypes. Noise prototypes and prototype functions can be combined independently into one-dimensional unit tests.

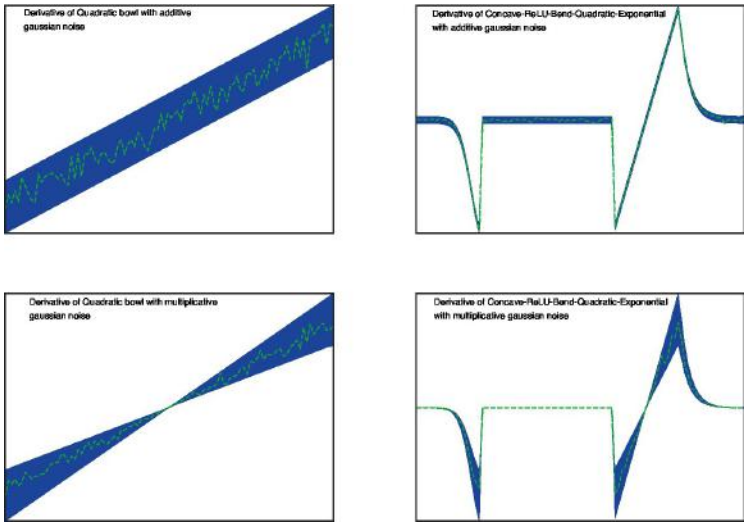

Figure 2: Examples of noise applied on prototype functions, green dashed are typical sample gradients, and the standard deviation range is the blue area. The upper two subplots depict Gaussian additive noise, while the lower two show Gaussian multiplicative noise. In the left column, the noise is applied to the gradients of a quadratic bowl prototype (note how the multiplicative noise goes to zero around the optimum in the middle), and on the right it is applied to a concatenation of prototypes.

2.4 Multi-dimensional Composition

A whole range of difficulties for optimization only exist in higher dimensional parameter spaces (e.g., saddle points, conditoning, correlation). Therefore, we build high-dimensional unit tests by composing together one-dimensional unit tests. For example for two one-dimensional prototype shapes $\mathcal{L}_a$ and $\mathcal{L}_b$ combined with a p-norm, the composition is $\mathcal{L}_{(a,b)}(\theta) = (\mathcal{L}_a(\theta_1)^p + \mathcal{L}_b(\theta_2)^p)^{\frac{1}{p}}$. Noise prototypes are composed independently of shape prototypes. While they may be composed of concatenated one-dimensional prototypes, higher-dimensional prototypes are not concatenated themselves. Various levels of *conditioning* can be achieved by having dramatically different scales in different component dimensions.

In addition to the choice of prototypes to be combined, and their scale, we permit a rotation in input space, which couples the dimensions together and avoids axis-alignment. These rotations are particularly important for testing diagonal/element-wise optimization algorithms.

2.5 Curl

In reinforcement learning a value function (the expected discounted reward for each state) can be learned using temporal-difference learning (TD), an update procedure that uses *bootstrapping*: i.e. it pulls the value of the current state towards the value of its successor state [19]. These stochastic update directions are not proper gradients of any scalar energy field [20], but they still form a (more general) vector field with non-zero curl, where the objective for the optimization algorithm is to converge to its fixed-point(s). See Figure 4 for a detailed example. We implemented this aspect by allowing different amounts of curl to be added on top of a multi-dimensional vector field in our unit tests, which is done by rotating the produced gradient vectors using a fixed rotation matrix. This is reasonably realistic; in fact, for the TD example in Figure 4, the resulting vector field is exactly the gradient field of a quadratic combined with a (small-angle) rotation.

2.6 Non-stationarity

In many settings it is necessary to optimize a non-stationary objective function. This may typically occur in a non-stationary task where the problem to be solved changes over time. However, non-

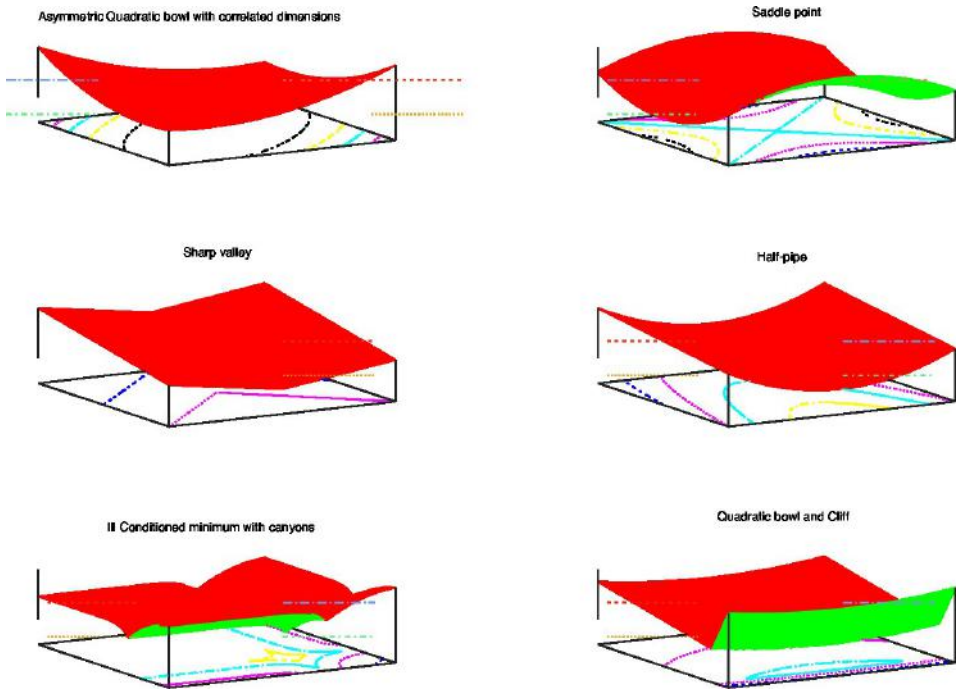

Figure 3: Examples of multivariate prototypes. The first subplot depicts an asymmetric quadratic bowl with correlated dimensions, the second a surface with a saddle point, the third a sharp valley surface, the fourth a half-pipe surface where the first dimension is a line and the second one a quadratic bowl. The fifth subplot depicts a surface with an ill conditioned minimum in the point where the two canyons overlap. The surface in the last subplot is the composition of a quadratic bowl in the first dimension and of a cliff in the second.

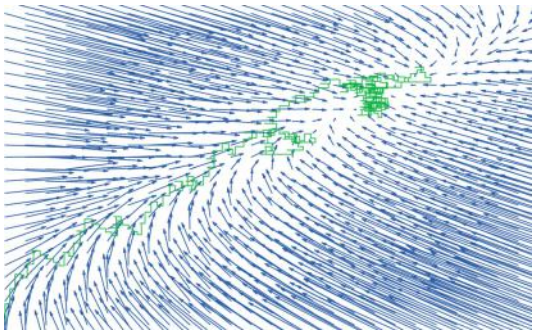

Figure 4: Here, we consider a very simple Markov process, with two states and stochastic transitions between them, and a reward of 0 in the first and of 1 in the second state. Consider the parameters of our optimization θ to be the two state values. Each TD update changes one of them, depending on the stochastic transition observed. In this figure, we plot the vector field of expected update directions (blue arrows) as a function of θ, as well as one sampled trajectory of the TD algorithm. Note how this vector field is not actually a gradient field, but instead has substantial curl, making it a challenging stochastic optimization task.

stationary optimization can even be important in large stationary tasks (with temporal structure in the samples), when the algorithm chooses to *track* a particular dynamic aspect of the problem, rather than attempting to converge to a global but static solution of the problem [21]. In addition, reinforcement learning (RL) tasks often involve non-stationary optimization. For example, many RL algorithms proceed by evaluating the value function using the TD algorithm described in the

previous section. This results in two sources of non-stationarity: the target value changes at every step (resulting in the previously described curl); and also the state distribution changes as the value function improves and better actions are selected. These scenarios can be therefore be viewed as non-stationary loss functions, but whose optimum moves as a function of the current parameter values.

We test non-stationarity in three different ways. We let the location of the optimum move smoothly, via random translations of the parameter space, or we let the the scale of the shape prototype vary randomly (on average by 10% in each direction), or, on noisy unit tests, we let the scale of the noise vary randomly. Currently, these changes happen once every 10 steps. A type of non-stationarity that involves more abrupt switching is discussed in section 4.1.

3 Experiments

3.1 Setup and Algorithms

For our experiments, we test the candidate algorithms on over 3000 unit tests, with up to 10 parameter dimensions. Each algorithm-unit test pairing is repeated 10 times, but with reusing the same 10 random seeds across all algorithms and setups. For each run k, we compute the true expected loss at the parameter value reached after 100 update steps $\mathcal{L}^{(k)} = \mathbb{E}\left[\mathcal{L}\left(\theta_{100}^{(k)}\right)\right]$.

The algorithms evaluated are SGD with fixed learning rate $\eta_0 \in [10^{-6}, 10]$, SGD with annealing with decay factor in $[10^{-2}, 1]$ and initial rates η_0, SGD with momentum (regular or Nesterov's variant [22]) $[0.1, 0.999]$ and initial rates η_0, SGD with parameter averaging [] with decay term in $[10^{-4}, 0.5]$ and exponent in $\{\frac{1}{2}, \frac{3}{4}, 1\}$, ADAGRAD [10] with initial rates η_0, ADADELTA [23] with decay parameter $(1-\gamma) \in [10^{-4}, 0.5]$ and regularizer in $[10^{-6}, 10^{-2}$, the incremental delta-bar-delta algorithm (IDBD [24]), RPROP [25] with initial stepsizes η_0, RMSprop [26] with minimal learning rates η_0, maximal learning rates in $[10, 10^3]$ and decay parameter γ, as well as conjugate gradients. For the hyper-parameters ranges, we always consider one value per order of magnitude, and exhaustively sweep all combinations.

3.2 Reference performance

Each unit test is associated with a reference performance $\mathcal{L}_{sgd}$, and a corresponding reference learning rate η_{best} that is determined by doing a parameter sweep over all fixed learning rates for SGD (34 values log-uniform between 10^{-10} and 10) and retaining the best-performing one.

In our aggregate plots, unit tests are sorted (per group) by their reference learning rate, i.e., those that require small steps on the left, and those where large steps are best on the right. Algorithm setups are sorted as well, on the vertical axis, by their median performance on a reference unit test (quadratic, additive noise).

3.3 Qualitative Evaluation

The algorithm performance $\mathcal{L}^{(k)}$ is converted to a normalized value $\mathcal{L}_{norm}^{(k)} = \frac{\mathcal{L}^{(k)} - \mathcal{L}_{init}}{\mathcal{L}_{sgd} - \mathcal{L}_{init}}$ where $\mathcal{L}_{init} = \mathbb{E}[\mathcal{L}(\theta_0)]$ is the expected loss value at the initial point, similar to the approach taken in [27], but even more condensed. In other words, a normalized value near zero corresponds to no progress, negative denotes divergence, and a value near one is equivalent to the best SGD. Based on these results, we assign a qualitative color value to the performance of each algorithm setup on each unit test, to able to represent it in a single pixel in the resulting figures:

- **Red:** Divergence or numerical instability in all run.
- **Violet:** Divergence or numerical instability in at least one run.
- **Orange:** Insufficient progress: median$(\mathcal{L}_{norm}) < 0.1$
- **Yellow:** Good progress: median$(\mathcal{L}_{norm}) > 0.1$ and high variability: $\mathcal{L}_{norm} < 0.1$ for at least $\frac{1}{4}$ of the runs.

- **Green:** Good progress: median($\mathcal{L}_{norm}$) > 0.1 and low variability: $\mathcal{L}_{norm} < 0.1$ for at most $\frac{1}{4}$ of the runs.
- **Blue:** Excellent progress: median($\mathcal{L}_{norm}$) > 2.

3.4 Results

Figures 5 and 6 shows the qualitative results of all algorithm variants on all the unit tests. There is a wealth of information in these visualizations. For example the relatively scarce amount of blue indicate that it is difficult to substantially beat well-tuned SGD in performance on most unit tests. Another unsurprising conclusion is that hyper-parameter tuning matters much less for the adaptive algorithms (ADAGRAD, ADADELTA, RPROP, RMSprop) than for the non-adaptive SGD variants. Also, while some unit tests are more tricky than others on average, there is quite some diversity in the sense that some algorithms may outdo SGD on a unit test where other algorithms fail (especially on the non-differentiable functions).

4 Realism and Future Work

We do not expect to replace real-world benchmark domains, but rather to complement them with our suite of unit tests. Still, it is important to have sufficient coverage of the types of potential difficulties encountered in realistic settings. To a much lesser degree, we may not want to clutter the test suite with unit tests that measure issues which never occur in realistic problems.

It is not straightforward to map very high-dimensional real-world loss functions down to low-dimensional prototype shapes, but it is not impossible. For example, in Figure 8 we show some random projections in parameter space of the loss function in an MNIST classification task with an MLP [28]. We defer a fuller investigation of this type, namely obtaining statistics on how commonly different prototypes are occurring, to future work.

However, the unit tests capture the properties of some examples that *can* be analyzed. One of them was discussed in section 2.5, another one is the simple loss function of a one-dimensional auto-encoder:

$$\mathcal{L}_\theta(x) = (x + \theta_2 \cdot \sigma(x \cdot \theta_1))^2$$

where σ is the sigmoid function. Even in the absence of noise, this minimal scenario has a saddle-point near $\theta = (0, 0)$, a plateau shape away from the axes, a cliff shape near the vertical axis, and a correlated valley near $\theta = (1, 1)$, as illustrated in Figure 7. All of these prototypical shapes are included in our set of unit tests.

An alternative approach is predictive: if the performance on the unit tests is highly predictive of an algorithm's performance on a some real-world task, then those unit tests must be capturing the essential aspects of the task. Again, building such a predictor is an objective for future work.

4.1 Algorithm Dynamics

Our long-term objective is to be able to do systematic testing and a full investigation of the optimization dynamics for a given algorithm. Of course, it is not possible to test it exhaustively on all possible loss functions (because there are infinitely many), but a divide-and-conquer approach may be the next best thing. For this, we introduce the notion of *algorithm state*, which is changing during optimization (e.g., the current stepsize or momentum). Now, a long optimization process can be seen as the *chaining* of a number of unit tests, while preserving the algorithm state in-between them. Our hypothesis is that the set of all possible chains of unit tests in our collection covers most of the qualitatively different (stationary or non-stationary) loss functions an optimization algorithm may encounter.

To evaluate an algorithm's robustness (rather than its expected performance), we can assume that an *adversary* picks the worst-case unit tests at each step in the sequence. An algorithm is only truly robust if it does not diverge under any sequence of unit tests. Besides the worst-case, we may also want to study typical expected behavior, namely whether the dynamics have an *attractor* in the algorithm's state space. If an attractor exists where the algorithm is stable, then it becomes useful to look at the secondary criterion for the algorithm, namely its expected (normalized) performance.

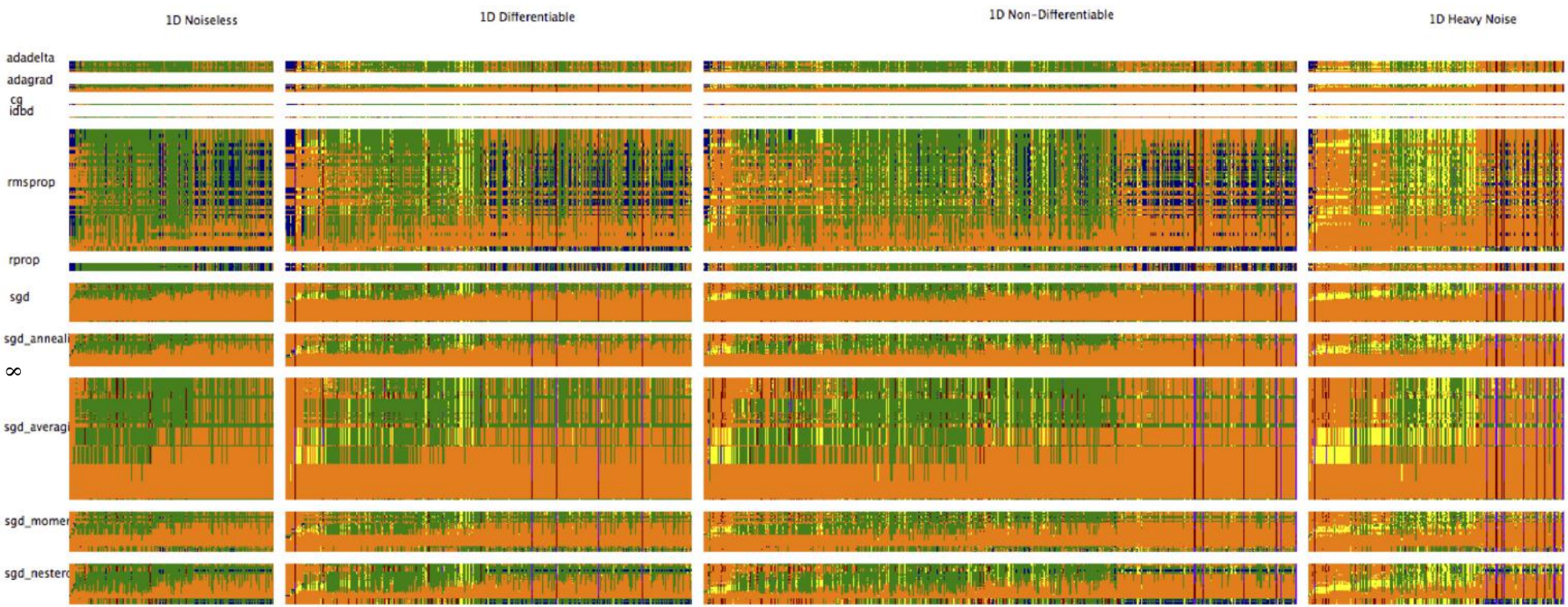

Figure 5: Qualitative results for all algorithm variants (350) on all stationnary, one-dimensional unit tests. Each column is one unit test, grouped by shared properties (see caption), for example the first group includes all noise-free 1D unit tests, where groups of unit tests can be partially overlapping. Each group of rows is one algorithm, with one set of hyper-parameters per row. The color code is: red/violet=divergence, orange=slow, yellow=variability, green=acceptable, blue=excellent (see main text for details).

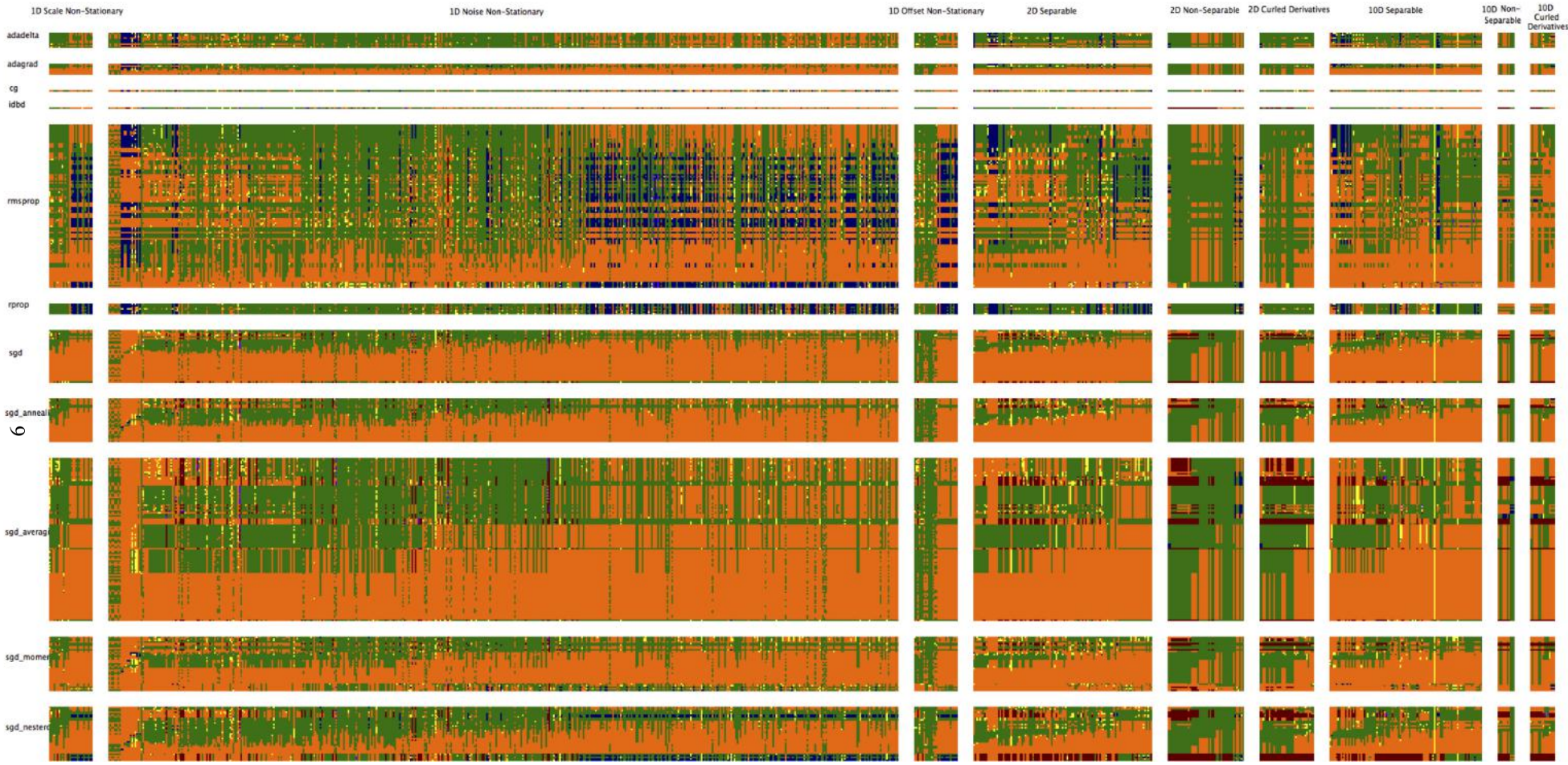

Figure 6: Qualitative results for all algorithm variants (350) on all non-stationary one-dimensional unit tests (firs three groups), on all two-dimensional ones (next three groups), and on all ten-dimensional ones (last three groups). The color code is: red/violet=divergence, orange=slow, yellow=variability, green=acceptable, blue=excellent. See Figure 5 and main text for details.

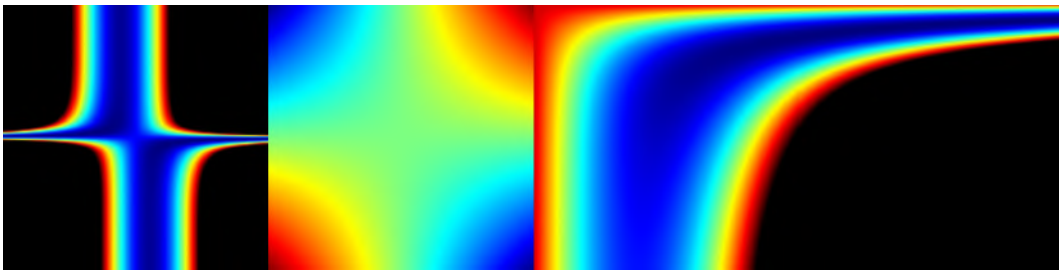

Figure 7: Illustration of the loss surface of a one-dimensional auto-encoder, as defined in the text, where the darkest blue corresponds to the lowest loss. **Left:** from the zoomed-out perspective if appears to be roughly a vertical valley, leading an optimizer toward the y-axis from almost anywhere in the space. **Center:** the zoomed-in perspective around the origin, which is looking like a prototypical saddle point. **Right:** the shape of the valley in the lower left quadrant, the walls of which become steeper the more the search progresses.

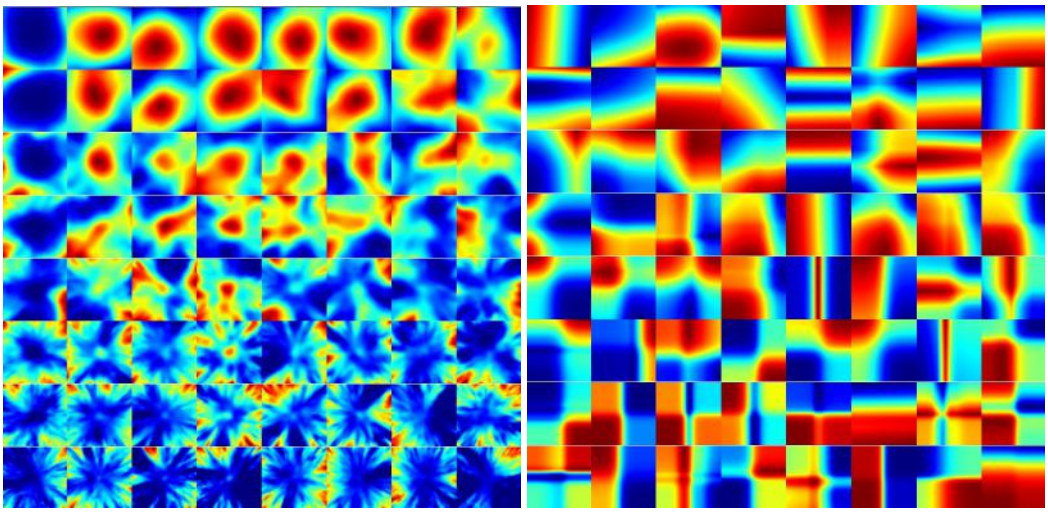

Figure 8: **Left:** collection of 64 random projections into two dimensions of the MNIST loss surface (based on one randomly sampled digit for each column). The projections are centered around the weights learned after one epoch of training, and different projections are plotted on scales between 0.05 (top row) and 0.5 (bottom row). **Right:** the same as on the left, but with axis-aligned projections.

We conjecture that this analysis may lead to novel insights into how to design robust and adaptive optimization algorithms.

5 Conclusion

This paper established a large collection of simple comparative benchmarks to evaluate stochastic optimization algorithms, on a broad range of small-scale, isolated, and well-understood difficulties. This approach helps disentangle issues that tend to be confounded in real-world scenarios, while retaining realistic properties. Our initial results on a dozen established algorithms (under a variety of different hyperparameter settings) show that robustness is non-trivial, and that different algorithms struggle on different unit tests. The testing framework is open-source, extensible to new function classes, and easy to use for evaluating the robustness of new algorithms.

The full source code (see also Appendix A) is available under BSD license at:

`https://github.com/IoannisAntonoglou/optimBench`

Acknowledgements

We thank the anonymous ICLR reviewers for their many constructive comments.

References

[1] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[2] Léon Bottou. Online Algorithms and Stochastic Approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[3] Léon Bottou and Yann LeCun. Large Scale Online Learning. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[4] Léon Bottou and Olivier Bousquet. The Tradeoffs of Large Scale Learning. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. NIPS Foundation (http://books.nips.cc), 2008.

[5] A. Benveniste, M. Metivier, and P. Priouret. *Adaptive Algorithms and Stochastic Approximations*. Springer Verlag, Berlin, New York, 1990.

[6] N. Le Roux, P.A. Manzagol, and Y. Bengio. Topmoumoute online natural gradient algorithm, 2008.

[7] Antoine Bordes, Léon Bottou, and Patrick Gallinari. SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent. *Journal of Machine Learning Research*, 10:1737–1754, July 2009.

[8] Wei Xu. Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent. *ArXiv-CoRR*, abs/1107.2490, 2011.

[9] Tom Schaul, Sixin Zhang, and Yann LeCun. No More Pesky Learning Rates. In *International Conference on Machine Learning (ICML)*, 2013.

[10] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. 2010.

[11] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *arXiv preprint arXiv:1211.5063*, 2012.

[12] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In G. Orr and Muller K., editors, *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256. Society for Artificial Intelligence and Statistics, 2010.

[13] Nikolaus Hansen, Anne Auger, Steffen Finck, Raymond Ros, et al. Real-parameter black-box optimization benchmarking 2010: Experimental setup. 2010.

[14] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 1689–1696. ACM, 2010.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

[16] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[17] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient BackProp. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998.

[18] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[19] R.S. Sutton and A.G. Barto. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, Sep 1998.

[20] Etienne Barnard. Temporal-difference methods and Markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2):357–365, 1993.

[21] Richard S. Sutton, Anna Koop, and David Silver. On the role of tracking in stationary environments. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML 2007*, pages 871–878. ACM Press, 2007.

[22] Yurii Nesterov and Arkadii Semenovich Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13. SIAM, 1994.

[23] Matthew D Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701*, 2012.

[24] Richard S Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, pages 171–176, 1992.

[25] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[26] T Tieleman and G Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 2012.

[27] Tom Schaul and Yann LeCun. Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. In *International Conference on Learning Representations*, Scottsdale, AZ, 2013.

[28] Yann LeCun and Corinna Cortes. The MNIST dataset of handwritten digits. 1998. http://yann.lecun.com/exdb/mnist/.

A Appendix: Framework Software

As part of this work a software framework was developed for the computing and managing all the results obtained for all the different configurations of function prototypes and algorithms. The main component of the system is a database where all the results are stored and can be easily retrieved by querying the database accordingly. The building blocks of this database are the individual experiments, where each experiment is associated to a unit test and an algorithm with fixed parameters. An instance of an experiment database can either be loaded from the disk, or it can be created on the fly by running the associated experiments as needed. The code below creates a database and runs all the experiments for all the readily available algorithms and default unit tests, and then saves them to disk:

```
require 'experiment'
local db = experimentsDB()
db:runExperiments()
db:save('experimentsDB')
```

This database now can be loaded from the disk, and the user can query it in order to retrieve specific experiments, using *filters*. An example is shown below:

```
local db = experimentsDB()
db:load('experimentsDB')
local experiments = db:filter({fun={'quad', 'line'},
                               algo={'sgd'}, learningRate=1e-4})
```

The code above loads an experiment database from the disk and it retrieves all the experiments for all the quadratic and line prototype shapes, for all different types of noise and all scales, further selecting the subset of experiments to those optimized using SGD with learningRate equal to 1e-4. The user can rerun the extracted experiments or have access to the associated results, i.e., the expected value of the function in different optimization steps, along with the associated parameters values. In order to qualitatively assess the results the following code can be used:

```
db:ComputeReferenceValues()
db:cleanDB()
```

```
db:assessPerformance()
db:plotExperiments()
```

The code above computes the reference expected values for each prototype function, it removes the experiments for which no reference value is available, then it qualitatively assesses the performance of all the available experiments and finally it plots the results given the color configuration described in section 3.3. It is really easy to add a new algorithm in the database in order to evaluate its robustness. The code below illustrates a simple example:

```
db:addAlgorithm(algoname, algofun, opt)
db:testAlgorithm(algoname)
db:plotExperiments({}, {algoname})
```

Here a new algorithm with name `algoname`, function instance `algo` (which should satisfy the `optim` interface), and a table of different parameter configurations `opt` is added to the database and it is tested under all available functions prototypes. Finally, the last line plots a graph with all the results for this algorithm.

It is also possible to add a set of new unit tests to the database, and subsequently run a set of experiments associated with them. There are different parameters to be defined for the creation of a set of unit tests (that allow wildcard specification too):

1. the concatenated shape prototypes for each dimension,
2. the noise prototype to be applied to each dimension,
3. the scale of each dimension of the function,
4. in case of multivariate unit tests, a parameter specifies which p-norm is used for the combination,
5. a rotation parameter that induces correlation of the different parameter dimensions, and
6. a curl parameter that changes the vector field of a multivariate function.