
From Learning to Optimize to Learning Optimization Algorithms

Camille Castera

University of Bordeaux
Bordeaux INP, CNRS, IMB, UMR 5251
Talence, France

Peter Ochs

Department of Mathematics and Computer Science
Saarland University, Germany

Abstract

Towards designing learned optimization algorithms that are usable beyond their training setting, we identify key principles that classical algorithms obey, but have up to now, not been used for Learning to Optimize (L2O). Following these principles, we provide a general design pipeline, taking into account data, architecture and learning strategy, and thereby enabling a synergy between classical optimization and L2O, resulting in a philosophy of *Learning Optimization Algorithms*. As a consequence our learned algorithms perform well far beyond problems from the training distribution. We demonstrate the success of these novel principles by designing a new learning-enhanced BFGS algorithm and provide numerical experiments evidencing its adaptation to many settings at test time.

1 INTRODUCTION

Learning to Optimize (**L2O**) is a modern and promising approach towards designing optimization algorithms that reach a new level of efficiency. L2O is even the state-of-the-art approach in some applications (Liu and Chen, 2019; Zhang et al., 2021b). However, it mostly excels when designed and trained specifically for each application and still fails to be widely applicable without retraining models. This need to adapt L2O specifically to each task is especially problematic given how difficult designing L2O algorithm is: the design is prone to many conceptional pitfalls and training models is not only computationally expensive (Chen et al., 2022) but also notoriously hard in L2O (Metz et al.,

2019). In contrast, standard optimization methods are widely applicable, sometimes way beyond the setting they were originally designed for, as attested for example by the success of momentum methods (Polyak, 1964) in deep learning (Jelassi and Li, 2022). This transfer of performance to different classes of problems is often achieved only at the cost of tuning a few scalar hyper-parameters. Analytically designed optimization algorithms usually come with theoretical guarantees, which most L2O algorithms lack completely of.

To bring L2O algorithms closer to actual Learned Optimization Algorithms (**LOA**), we identify key theoretical principles that hand-crafted optimization algorithms follow and provide strategies ensuring that L2O approaches inherit these properties. Thereby we systematically unify the advantages of both worlds: flexible applicability and theoretically controlled convergence guarantees from mathematical optimization, and complex operations from machine learning beyond what can be analytically designed. We illustrate our new approach by designing a learning-enhanced BFGS method. We present numerical and theoretical evidence that our approach benefits L2O.

2 RELATED WORK

Learning to Optimize L2O (Li and Malik, 2016) is an active topic of research. A lot of work focuses on unrolling (Gregor and LeCun, 2010; Ablin et al., 2019; Huang et al., 2022; Liu and Chen, 2019) and “Plug and Play” (Venkatakrisnan et al., 2013; Meinhart et al., 2017; Zhang et al., 2021b; Terris et al., 2024) approaches which improve over hand-crafted algorithms in several practical cases. L2O often lacks theoretical guarantees, with a few exceptions where convergence is enforced via “safeguards” that restrict the method (Moeller et al., 2019; Heaton et al., 2023; Martin and Furieri, 2024), or estimated statistically in distribution (Sucker and Ochs, 2023). One can also learn “good initializations” before using convergent algorithms (Sambharya et al., 2024). The L2O literature is broad, see Chen et al. (2022); Amos (2023) for detailed overviews of the topic.

Design principles One of our contributions (see Section 4) is to enforce robustness to geometric transformations in L2O. This is related to “geometric deep learning” (Bronstein et al., 2021): the more general topic of preserving equivariance properties (defined in Section 4) in the context of learning. It has many applications (Romero and Cordonnier, 2020; Chen et al., 2021; Hutchinson et al., 2021; Terris et al., 2024; Chen et al., 2023; Keriven and Vaiter, 2024; Levin and Díaz, 2024). It is also connected to learning on sets (Zaheer et al., 2017; Lee et al., 2019). To the best of our knowledge equivariance for L2O is only considered by Ollivier et al. (2017), from a probabilistic point of view and Tan et al. (2024). The latter argues that equivariance properties generally benefit L2O algorithms. Our work rather identifies specific key equivariance properties and studies how to handle them in every step of a general L2O pipeline (through Algorithm 1). Parallel to our approach, Liu et al. (2023) proposed to enforce convergence properties by design, hence stabilizing L2O methods, whereas we focus on generalization. Finally, improving the design of L2O algorithms from a practical perspective has been studied in (Wichrowska et al., 2017; Metz et al., 2019, 2022)

Learning quasi-Newton methods We use learning to enhance a BFGS-like algorithm. BFGS (Broyden, 1970; Fletcher, 1970; Goldfarb, 1970; Shanno, 1970) is the most popular quasi-Newton (QN) algorithm and has been extensively analyzed (Greenstadt, 1970; Dennis and Moré, 1977; Ren-Pu and Powell, 1983). Many extensions have been proposed, featuring limited memory (Liu and Nocedal, 1989), sparse (Toint, 1981) and non-smooth (Wang et al., 2022) versions, or modifications provably faster in specific settings (Rodomanov and Nesterov, 2021; Jin et al., 2022). Other approaches to make use of second-order derivatives only relying on gradients include symmetric-rank-one methods (Conn et al., 1991; Becker and Fadili, 2012) and the dynamical inertial Newton family of methods (Alvarez et al., 2002; Attouch et al., 2016, 2022) which is at the interface of first- and second-order optimization (Castera et al., 2024).

Several approaches, have previously been proposed to learn BFGS methods. A transformer model has been derived by (Gärtner et al., 2023), and (Liao et al., 2023) considered learning on the fly in the online setting. A recent work (Li et al., 2023) predicted a weighted average between DFP (Powell, 1983) and BFGS (a.k.a. a Broyden method). This is more akin to hyper-parameter tuning as their method remains in the span of Broyden’s methods. Our approach allows to build rather different learned QN algorithms by using a variational derivation of BFGS (see Section 5.1), originating from (Greenstadt, 1970; Goldfarb, 1970) and which has been

used in (Hennig and Kiefel, 2013) for Bayesian optimization.

3 SETTING AND PROBLEM STATEMENT

We propose a mathematical formalism for L2O, akin to the “semi-amortized” framework from (Amos, 2023). Compared to the latter we further decompose the algorithm into four pieces that will later allow us to mathematically discuss our main contribution of providing L2O algorithms with optimization properties in Section 4.

3.1 Mathematical Formalism

We denote by \mathbb{N} the set of non-negative integers and \mathbb{R} the set of real numbers. In what follows we consider unconstrained optimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x), \quad (1)$$

where $n \in \mathbb{N}$ is the dimension of the problem, and f belongs to \mathfrak{F}_n : the set of real-valued lower bounded twice-continuously differentiable functions on \mathbb{R}^n (with inner product $\langle \cdot, \cdot \rangle$ and norm $\|\cdot\|$). Gradient and Hessian matrix of f are denoted by ∇f and $\nabla^2 f$ respectively.

We consider L2O models that are applicable in any dimension (see Principle 1 below), like standard optimization algorithms. Therefore in the sequel, the dimension n is arbitrary and need not be the same for all the problems the algorithms are applied to. Nevertheless, for the sake of simplicity, the following discusses a fixed \mathbb{R}^n . We call *problem*, a triplet (f, x_0, S_0) , made of an objective function $f \in \mathfrak{F}_n$, an initialization $x_0 \in \mathbb{R}^n$ and a collection of vectors and matrices $S_0 \in \mathfrak{S}_n$, called *state* (\mathfrak{S}_n is the set of all possible states, clarified hereafter).

We formulate L2O algorithms in a generic form described in Algorithm 1 that takes as input a problem (f, x_0, S_0) and performs $K \in \mathbb{N}$ iterations before returning $x_K \in \mathbb{R}^n$. Algorithm 1 can also be mathematically represented by an operator $\mathcal{A}: \mathfrak{F}_n \times \mathbb{R}^n \times \mathfrak{S}_n \times \mathbb{N} \rightarrow \mathbb{R}^n$ such that the K -th iteration of the algorithm reads

$$x_K = \mathcal{A}(f, x_0, S_0, K).$$

Algorithm 1 is fully characterized by what we call an *oracle* \mathcal{C} , a *model* \mathcal{M}_θ , an *update* function \mathcal{U} and a *storage* function \mathcal{S} . At any iteration $k \in \mathbb{N}$, the oracle \mathcal{C} collects the information the algorithm has access to about f at the current point x_k and the state S_k and constructs an *input* $I_k \in \mathbb{R}^{n \times n_i}$, where $n_i \in \mathbb{N}$. The input I_k is then fed to a (machine learning) model represented by a parametric function $\mathcal{M}_\theta: \mathbb{R}^{n \times n_i} \rightarrow \mathbb{R}^{n \times m}$, where $\theta \in \mathbb{R}^p$ ($p \in \mathbb{N}$) is its parameter (in vector

form), and $m \in \mathbb{N}$. The model outputs a *prediction*, i.e., $y_k = \mathcal{M}_\theta(I_k)$, which is used by the update $\mathcal{U}: \mathbb{R}^{n \times n_i} \times \mathbb{R}^{n \times m} \times \mathbb{R}^{n_h}$ to improve the current point: $x_{k+1} = x_k + \mathcal{U}(I_k, y_k, \Gamma)$, where $\Gamma \in \mathbb{R}^{n_h}$ ($n_h \in \mathbb{N}$) are the hyper-parameters chosen by the user (a few scalars). The storage \mathcal{S} then collects, in S_{k+1} , the information from the k -th iterate that will be used at the next iteration. This abstract formalism is generic enough to encompass at the same time L2O and several classical algorithms. Moreover, this systematic structuring allows for the formulation and analysis of key principles for LOA in Section 4. We now illustrate this on an example.

Algorithm 1: Generic LOA

given: oracle \mathcal{C} , model \mathcal{M}_θ , update \mathcal{U} , storage \mathcal{S}

input: problem (f, x_0, S_0) , number of iterations K , hyper-parameter Γ

for $k = 0$ **to** $K - 1$:

```

     $I_k \leftarrow \mathcal{C}(f, x_k, S_k)$            // Construct input
     $y_k \leftarrow \mathcal{M}_\theta(I_k)$            // Model prediction
     $x_{k+1} \leftarrow x_k + \mathcal{U}(I_k, y_k, \Gamma)$  // Update step
     $S_{k+1} \leftarrow \mathcal{S}(S, x_k, I_k, y_k)$ 
    // Store relevant variables in state

```

return x_K

Example Throughout what follows we use the heavy-ball (HB) method (Polyak, 1964) as running example to illustrate the concepts we introduce. An iteration $k \in \mathbb{N}$ of HB reads:

$$x_{k+1} = x_k + \alpha d_k - \gamma \nabla f(x_k), \quad (2)$$

where $d_k = x_k - x_{k-1}$, $\alpha \in [0, 1)$ is called the momentum parameter and $\gamma > 0$ is the step-size. Notice that for $k = 0$, the algorithm requires not only x_0 but also $x_{-1} \in \mathbb{R}^n$. This is the reason for introducing a state in Algorithm 1, in this case we would have $S_0 = \{x_{-1}\}$. Any iteration k of HB reads as follows: the operator \mathcal{C} takes (f, x_k, S_k) , where $S_k = \{x_{k-1}\}$, and concatenates $\nabla f(x_k)$ and d_k as $I_k = (d_k, \nabla f(x_k)) \in \mathbb{R}^{n \times n_i}$, with $n_i = 2$. There is no learning phase hence no model \mathcal{M}_θ (by convention we say that $m = 0$ and $y_k = 0$). The update function \mathcal{U} has hyper-parameter $\Gamma = (\alpha, \gamma)$ —so $n_h = 2$ —and uses I_k to compute $\mathcal{U}(I_k, 0, \Gamma) = \alpha d_k - \gamma \nabla f(x_k)$, which yields the update (2). Finally, the storage \mathcal{S} stores x_k in S_{k+1} , as it will be required to compute d_{k+1} .

3.2 LOA is L2O with Specific Generalization

We detail the stages of building L2O models and the specific goal of a subfield of L2O: our LOA approach.

Training: The parameter θ of an L2O model \mathcal{M}_θ is set by minimizing a *loss function* measuring how well (1) is solved on a *training set* of problems (f, x_0, S_0) (see (7)). Training is crucial to find a “good” θ but computationally expensive and hard in L2O (Metz et al., 2019). Our goal is to study cases where the model is trained on a fixed training set, and then used on other functions $f \in \mathfrak{F}_n$ *without retraining*.

Test phase: In machine learning, it standard to assume that the training set is sampled from an unknown underlying distribution of problems. Generalization, in the *statistical sense*, refers to asserting how the trained model \mathcal{M}_θ performs on the whole distribution. This is estimated by computing the performance on a *test set* sampled independently from the same distribution.

Generalization: One may wonder how the model \mathcal{M}_θ performs on problems not sampled from the aforementioned distribution. This is called out of distribution generalization and is unachievable in full generality (Wolpert, 1996). Instead, we note that hand-crafted optimization algorithms possess a different type of generalization property. We can sometimes use them for functions f (and initializations) they were not designed for, only by tuning the hyper-parameter Γ . For example, HB was originally designed for locally C^2 functions (Polyak, 1964), but works on larger classes of convex functions (Ghadimi et al., 2015) and even performs well on non-convex ones (Zavriev and Kostyuk, 1993). It also does not require a specific initialization and convergence rates are uniform in the dimension n (Polyak, 1987; Bertsekas, 1997). *LOA differs from the rest of L2O* by identifying specific generalization properties that most hand-crafted optimization algorithms have, and designing L2O algorithms that features them. We formulate these properties as a list of *principles*.

4 THE PRINCIPLES OF LOA

The cornerstone principle of LOA, is that optimization algorithms should be applicable in any dimension n . Since we *do not retrain the model* \mathcal{M}_θ , this implies the following.

Principle 1. *Algorithm 1 should be independent of the dimension n , i.e., the size p of θ and n_i should be independent of n and as small as possible.*

This principle makes LOA very different from the rest of L2O: the algorithm may be used on problems where p is much smaller than n . LOA is thus in the *under-parameterized* learning regime, so the training phase cannot be used to memorize many examples (unlike in the over-parameterized case (Zhang et al., 2021a)). We propose to cope with this via a careful algorithmic design revolving around three ideas.

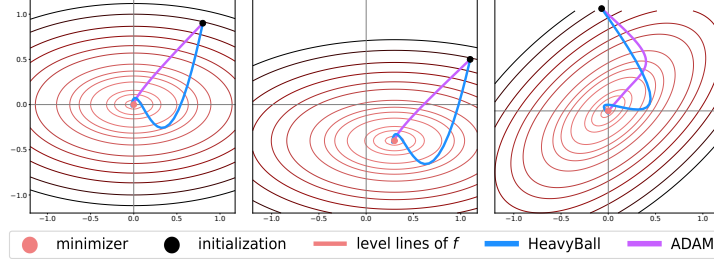


Figure 1: Illustration of equivariances on the landscape of a 2D function. Left: no transformation, middle: translation, right: rotation. Transforming f and x_0 does the same to the iterates of HB. ADAM (Kingma and Ba, 2015) is translation equivariant but not rotation equivariant.

Enhancement: We use learning to enhance existing hand-crafted algorithms, preserving their theoretically-grounded parts. We only replace parts based on heuristics with learning, eventually reducing the size of θ .

Adaption: LOA must adapt *on the fly* (along the iterates) to each problem by storing information in the state S_k . This can be achieved by recurrent neural networks, *e.g.*, LSTMs (Andrychowicz et al., 2016; Hochreiter and Schmidhuber, 1997) or by enhancing adaptive algorithms like ADAM (Kingma and Ba, 2015) or quasi-Newton methods, as we do in Section 5.

Hard-coded generalization: We show that most hand-crafted algorithms share generalization properties expressed through equivariance to key geometric transformations. This is one of the main contributions of our work, to which the rest of this section is dedicated to. Fix $f \in \mathfrak{F}_n$, $x_0 \in \mathbb{R}^n$ and S_0 and consider an invertible mapping $\mathcal{T}: \mathbb{R}^n \rightarrow \mathbb{R}^n$. Since \mathcal{T} is invertible, observe that for all $x \in \mathbb{R}^n$:

$$f(x) = f(\mathcal{T}^{-1}(\mathcal{T}(x))) = f \circ \mathcal{T}^{-1}(\mathcal{T}(x)) = \hat{f}(\hat{x}), \quad (3)$$

where we define \hat{f} as $f \circ \mathcal{T}^{-1}$ and $\hat{x} = \mathcal{T}(x)$, for all $x \in \mathbb{R}^n$. Therefore (3) expresses in particular that (f, x_0, S_0) and $(\hat{f}, \hat{x}_0, \hat{S}_0)$ are *two different representations of the same problem*, where \hat{S}_0 is one-to-one with S_0 such that for every vector¹ $\hat{v} \in \hat{S}_0$ there exists $v \in S_0$ such that $\hat{v} = \mathcal{T}(v)$. One would naturally want optimization algorithms (including Algorithm 1) to perform the same regardless the representation, *i.e.*,

$$f(\mathcal{A}(f, x_0, S_0, K)) = \hat{f}(\mathcal{A}(\hat{f}, \hat{x}_0, \hat{S}_0, K)), \quad \forall K \in \mathbb{N}. \quad (4)$$

According to (3), a sufficient condition is that $\mathcal{A}(\hat{f}, \hat{x}_0, \hat{S}_0, K) = \mathcal{T}(\mathcal{A}(f, x_0, S_0, K))$ holds for all (f, x_0, S_0) and K . When this is true, we say that the algorithm \mathcal{A} is *equivariant* to \mathcal{T} .

¹The case of matrices contained in S_0 is more complex and discussed in Appendix I.

Link with Generalization Machine learning exploits similarity in data; in L2O, this means similarity between landscapes of objective functions. This is in line with our approach since (3) expresses a specific form of similarity: that w.r.t. invertible transformations \mathcal{T} . Although equivariances can sometimes be learned via data augmentation (Nordenfors et al., 2025), we argue that in the under-parameterized setting (where LOA belongs), enforcing those by design avoids wasting parts of the small parameter θ relearning them.

Trade-off While one would naturally want equivariance with respect to any invertible \mathcal{T} , this imposes severe restrictions on the design of Algorithm 1. We therefore analyze equivariance only with respect to key transformations, summarized in Table 1. Actually, even most hand-crafted algorithms do not achieve all the equivariances considered in Table 1. A notable exception is Newton’s method, which is unsuitable for large-scale optimization. There is thus *always a tradeoff to find*. In fact, for hand-crafted algorithms, Fletcher (2000) hypothesizes that the success of BFGS comes from the tradeoff it achieves between its computational cost and the equivariances it possesses.

We now discuss how to enforce specific equivariances by design in Algorithm 1.

4.1 Translations

Let $v \in \mathbb{R}^n$, the translation \mathcal{T}_v is defined for all $x \in \mathbb{R}^n$ by $\mathcal{T}_v(x) = x + v$. Then $\hat{f} = f(\cdot - v)$ and $\hat{x}_0 = x_0 + v$. Most algorithms are translation equivariant (see Table 1 and Figure 1), which leads to the following principle.

Principle 2. *Algorithm 1 should be translation equivariant, *i.e.*, \mathcal{T}_v -equivariant for all $v \in \mathbb{R}^n$.*

Strategy Remark that since $\hat{x}_0 = x_0 + v$, then $\hat{x}_K = x_K + v \iff \sum_{k=0}^{K-1} \hat{d}_k = \sum_{k=0}^{K-1} d_k$. So we want to ensure that $\forall k \in \mathbb{N}$, $d_k = \hat{d}_k$. The quantities $\nabla f(x_k)$ and $d_k = x_k - x_{k-1}$ used in HB (Section 3.1) are translation *invariant* since for example

Table 1: Summary of invariance and equivariance properties of several algorithms (proved in Appendix H)

	Translation (Principle 2)	Permutation (Principle 3)	Orthogonal transform.	Geom. scaling (Principle 4)	Func. scaling (Principle 5)
Gradient desc.	✓	✓	✓	dep. Γ	dep. Γ
Heavy-Ball	✓	✓	✓	dep. Γ	dep. Γ
Newton Meth.	✓	✓	✓	✓	✓
BFGS	✓	✓	✓	✓	✓
ADAM	✓	✓	✗	dep. Γ	✓
Algorithm 2	✓	✓	✗	✓	✓

$\nabla \hat{f}(\hat{x}) = \nabla f(x)$, which makes HB translation equivariant (see the proof in Appendix H). This shows that it is often possible to make \mathcal{C} translation invariant (*i.e.*, $\mathcal{C}(f, x, S) = \mathcal{C}(\hat{f}, \hat{x}, \hat{S})$) which is then enough to make the algorithm translation equivariant, since by direct induction $\hat{y}_k = \mathcal{M}_\theta(\hat{I}_k) = \mathcal{M}_\theta(I_k) = y_k$ and thus $\mathcal{U}(\hat{I}_k, \hat{y}_k, \Gamma) = \mathcal{U}(I_k, y_k, \Gamma)$.

Practical consequences An easy way to ensure translation invariance of \mathcal{C} is to never output “absolute” quantities such as x_k but always differences like $d_k = x_k - x_{k-1}$, exactly like HB. We follow this strategy as it does not put any restriction on \mathcal{M}_θ nor \mathcal{U} .

4.2 Permutations

In optimization, the ordering of coordinates is often arbitrary. For example, the functions $(x, y) \mapsto x^2 + 2y^2$ and $(x, y) \mapsto 2x^2 + y^2$ represent the same problem with permuted coordinates. This transformation is represented by a *permutation matrix* P which contains only zeros except one element equal to 1 per line, and such that $P^T P = \mathbb{I}_n$ where \mathbb{I}_n denotes the identity matrix of size n . Fix such P and consider the corresponding transformation (with \hat{f} and \hat{x} redefined accordingly). As shown in Table 1, almost all popular algorithms are permutation equivariant.²

Principle 3. *Algorithm 1 should be equivariant to all permutation matrices P .*

Strategy Remark that this time to get $\hat{x}_k = Px_k$ for all $k \in \mathbb{N}$, we need $\hat{d}_k = Pd_k$ *i.e.* we need equivariance. Taking again the example of HB, we show in Appendix H that $\nabla \hat{f}(\hat{x}) = (P^{-1})^T \nabla f(x) = P \nabla f(x)$, and we should expect $\hat{d}_k = Pd_k$ (since this is what we want to obtain). Therefore we design \mathcal{C} to be equivariant to permutations, as this is what makes HB permutation equivariant. In L2O, we would get $\mathcal{M}_\theta(\hat{I}_k) = \mathcal{M}_\theta(P I_k)$, so we make \mathcal{M}_θ equivariant as

²A notable exception regards algorithms constructing block-diagonal matrices like K-FAC (Martens and Grosse, 2015), these blocks depend on the ordering of coordinates.

well ($\mathcal{M}_\theta(P I_k) = P \mathcal{M}_\theta(I_k)$) so that \mathcal{U} gets only permuted quantities, and finally design \mathcal{U} to preserve equivariance.

Practical consequences Permutation equivariance strongly restricts the choices for \mathcal{M}_θ as Zaheer et al. (2017) showed that the only fully-connected (FC) layer operating along the dimension n and preserving equivariance is very basic with only two learnable scalars and no bias. To be usable in any dimension n , many L2O models rely on per-coordinate predictions (Andrychowicz et al., 2016), making them permutation equivariant but completely neglecting interactions between coordinates. In Section 5 we propose a model that is permutation equivariant while allowing such interactions.

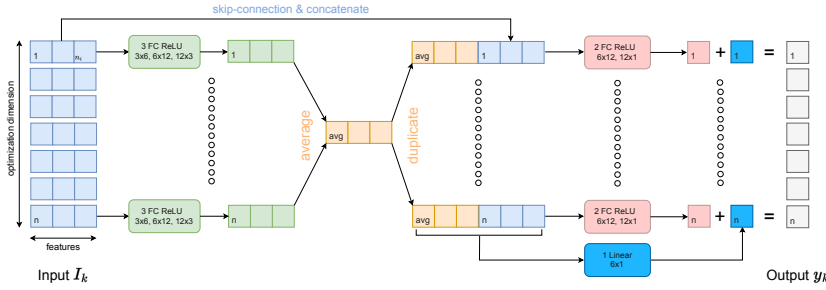
The orthogonal group Permutations matrices form a subset of the set of orthogonal matrices (square matrices with $P^T P = \mathbb{I}_n$). They correspond to so-called *rotations* and *reflections*. Several algorithms are equivariant to all orthogonal transformations, which makes this property appealing. Yet, we prove in Appendix H.4 that is hardly compatible with L2O since it does not hold for any FC layer with ReLU activation function. Fortunately, in many setting the canonical coordinate system has a clear meaning (e.g. each coordinate represents a weight of a neural network to train). So this requirement is *desirable but not crucial*, as the performance of ADAM in deep learning attests.

4.3 Rescaling

Let $\lambda > 0$ and consider the geometric rescaling $\mathcal{T}_\lambda(x) = \lambda x$, and redefine \hat{f} and \hat{x} accordingly.

Principle 4. *Algorithm 1 should be equivariant to geometric rescaling.*

This principle is also optional as it is usually not satisfied by first-order methods. Indeed, one can see that $\hat{f}(\hat{x}_k) = \frac{1}{\lambda} \nabla f(x_k)$ and we want $\hat{d}_k = \lambda d_k$. So for example one needs to tune (α, γ) in HB to recover equivariance (indicated by **dep. Γ** in Table 1). In contrast, Newton’s and QN methods are equivariant to



Layer	Size
FC	3 x 6
FC	6 x 12
FC	12 x 3
avg. & duplic.	—
FC	3 x 12
FC & linear	12 x 1 & 6 x 1

Figure 2: Architecture of our L2O model that preserves the principles of Section 4. The same layers are applied to each coordinate, and interactions between coordinates are allowed via an intermediate averaging and duplication.

geometric rescaling. In the context of LOA, we construct an algorithm (in Section 5) where we make \mathcal{M}_θ scale equivariant and show that our update \mathcal{U} can then make Principle 4 hold. This mildly restricts \mathcal{M}_θ , as *e.g.*, ReLU is scale equivariant but the sigmoid is not.

We consider a final, different, principle. For $\lambda > 0$, if the function is rescaled: $\hat{f} = \lambda f$, then $\nabla \hat{f} = \lambda \nabla f$. This does not transform the initialization: $\hat{x}_0 = x_0$ so we want *invariance* of the algorithm (and equivariance of function values).

Principle 5. *Algorithm 1 should be such that $\mathcal{A}(\lambda f, x_0, \hat{S}_0, K) = \mathcal{A}(f, x_0, S_0, K)$, $\forall \lambda > 0$.*

For similar reasons as geometric rescaling, this principle must be dealt with case-specifically, but is compatible with LOA as we show (see Theorem 1).

Remark 1. *ADAM is popular in deep learning (DL) because tuning its step-size is easier than for GD (Sivaprasad et al., 2020), which comes from its invariance to function rescaling (Principle 5). ADAM does not suffer from lacking Principle 4 in DL, thanks to scaled initialization strategies (He et al., 2015).*

4.4 Comparison to Common Heuristics

The principles stated in Section 4 provide a justification for heuristics that people have used in the L2O literature. For example a large part of the existing work followed Andrychowicz et al. (2016) and used coordinate-wise models, thus enabling Principle 1. Models also rely mostly on ∇f , which has the translation invariance property, crucial for Principle 2 (see Section 4.1). Notably, Wichrowska et al. (2017) discusses the scaling issue (Principle 5) and tries to mitigate it by decomposing inputs in magnitude and unit directions. Similarly, Lv et al. (2017) used what they call “random rescaling” (Section 4.1 therein), which is a data-augmentation technique that can exactly be interpreted as an attempt to learn Principle 4. Our work brings justification for these heuristics and provides design strategies

to replace them.

5 APPLICATION TO LEARNING QUASI-NEWTON ALGORITHMS

We illustrate our approach on the example of building a LOA, based on the BFGS method and prove it obeys the principles above. BFGS is a quasi-Newton (QN) method, *i.e.*, one that iteratively builds an approximation of the computationally-expensive inverse Hessian matrix used in Newton’s method. This is thus in line with the idea to adapt to each problem *on the fly* (see Section 4). While combining L2O and QN methods has been considered before (see Section 2), our approach differs in several aspects starting with the following.

5.1 A variational view on BFGS

Let $k \in \mathbb{N}$ be an iteration, we use the notation $\Delta g_k = \nabla f(x_k) - \nabla f(x_{k-1})$ and recall that $d_k = x_k - x_{k-1}$. QN methods are based on the fact that for quadratic functions the *secant equation* $d_k = \nabla^2 f(x_k)^{-1} \Delta g_k$ holds. QN methods aim to iteratively build approximations B_k to $\nabla^2 f(x_k)^{-1}$, with the constraint that B_k must preserve the secant equation: $d_k = B_k \Delta g_k$ and that B_k is symmetric. From a variational perspective, Greenstadt (1970); Goldfarb (1970) showed that BFGS aims to keep B_k close to the previous approximation B_{k-1} by taking B_k as the solution of the following problem:

$$B_k = \min_{\substack{B \in \mathbb{R}^{n \times n} \\ s.t. B_k \Delta g_k = d_k \text{ and } B_k - B_{k-1}^T = 0}} \|B - B_{k-1}\|_W. \quad (5)$$

Here $\|\cdot\|_W$ denotes the Frobenius norm reweighted by some symmetric positive definite matrix W . Denoting $y_k = W^{-1} \Delta g_k$ and $r_k = d_k - B_{k-1} \Delta g_k$, one can show that the solution of (5) is

$$B_k = B_{k-1} + \frac{1}{\langle \Delta g_k, y_k \rangle} \left[r_k y_k^T + y_k r_k^T - \frac{\langle \Delta g_k, r_k \rangle}{\langle \Delta g_k, y_k \rangle} y_k y_k^T \right]. \quad (6)$$

BFGS is then based on the heuristic trick (albeit elegant) that taking W^{-1} to be the *unknown* next approximation B_k , yields $y_k = d_k$ (due to the secant equation) and preserves positive-definiteness. Instead of using L2O to directly predict the matrix B_k as done in prior work, we use it precisely at this stage. We use a model \mathcal{M}_θ that *predicts directly* a different y_k than that of BFGS. Remark that there is no need to predict the matrix W since it only appears through the vector $y_k = W^{-1}\Delta g_k$. This allows enhancing BFGS with L2O while preserving the coherence of the algorithm.

5.2 Our Learned Algorithm

We now specify each part of our LOA-BFGS method.

The oracle \mathcal{C} For each iteration $k \in \mathbb{N}$ of our algorithm, we use the state $S_k = \{x_{k-1}, \nabla f(x_{k-1}), B_{k-1}\}$. Our oracle \mathcal{C} computes $\nabla f(x_k)$, d_k and Δg_k as in BFGS but also new features $B_{k-1}\Delta g_k$ and $-\gamma B_{k-1}\nabla f(x_k)$, gathered as $I_k = (B_{k-1}\Delta g_k, d_k, -\gamma B_{k-1}\nabla f(x_k))$. Note that all these features must be scale invariant since B_{k-1} approximates $\nabla^2 f(x_k)^{-1}$.

The learned model \mathcal{M}_θ Our model only takes three features as input ($n_i = 3$) but creates additional ones by applying a block of coordinate-wise FC layers, without bias, then averaging the result and concatenating it with I_k . This allows feature augmentation and makes each coordinate interacting with all the others. The result is then fed to another coordinate-wise block of FC layers (no bias) added to a linear layer yielding the output $y_k \in \mathbb{R}^n$. The architecture is detailed in Appendix K.1 and summarized in Figure 2. The linear layer acts as a skip-connection and will allow us to introduce a trick that stabilizes the training later in Section 6. Note that the cost of each operation inside \mathcal{M}_θ is proportional to n which is cheaper than the matrix-vector products of cost $O(n^2)$ that \mathcal{C} and \mathcal{U} (and vanilla BFGS) involve.

The update \mathcal{U} and storage \mathcal{S} Our update is that of BFGS:³ the approximation B_k is updated using (6) with a different y_k , and $x_{k+1} = x_k - \gamma B_k \nabla f(x_k)$, where $\gamma > 0$ is a step-size, usually close to 1 (or chosen by line-search). Like in vanilla BFGS, \mathcal{S} finally stores $\{x_k, \nabla f(x_k), B_k\}$ for the next iterate.

Initialization of B_{-1} QN methods require an initial approximated inverse Hessian matrix B_{-1} . While the simplest choice is \mathbb{I}_n , several works (Becker and Fadili,

2012; Becker et al., 2019) observed notable improvement by initializing with the Barzilai-Borwein (BB) step-size (Barzilai and Borwein, 1988): $\gamma_{\text{BB}}^{(0)} \stackrel{\text{def}}{=} \frac{\langle \Delta g_0, d_0 \rangle}{\|\Delta g_0\|^2}$. We follow this approach and take $B_{-1} = 0.8\gamma_{\text{BB}}^{(0)}\mathbb{I}_n$. With this choice, B_{-1} agrees with Principles 4 and 5 *without additional knowledge on f* (see Appendix I) and so do Algorithm 2 and BFGS.

The whole process is summarized in Algorithm 2. Note that replacing the model \mathcal{M}_θ with $y_k = d_k$ makes Algorithm 2 coincide exactly with BFGS. This will prove useful for training \mathcal{M}_θ (see Section 6).

Algorithm 2: Learning enhanced QN Algorithm

given: model \mathcal{M}_θ defined in Figure 2 and Sections 5.2 and K.1

input: function to minimize f , initialization x_0

input: initial state $S_0 = \{x_{-1}, \nabla f(x_{-1}), B_{-1}\}$,
(with $B_{-1} = 0.8\gamma_{\text{BB}}^{(0)}\mathbb{I}_n$)

input: number of iterations K , step-size γ
(default value $\gamma = 1$).

for $k = 0$ **to** $K - 1$:

Compute $\mathcal{C}(f, x_k, S_k)$:

$\Delta g_k = \nabla f(x_k) - \nabla f(x_{k-1})$

$d_k = x_k - x_{k-1}$

$I_k \leftarrow (B_{k-1}\Delta g_k, d_k, -\gamma B_{k-1}\nabla f(x_k))$

return I_k

Compute $\mathcal{M}_\theta(I_k)$:

return y_k

Compute Update step \mathcal{U} :

$r_k = d_k - B_{k-1}\Delta g_k$

$B_k = B_{k-1} +$

$\frac{1}{\langle \Delta g_k, y_k \rangle} \left[r_k y_k^T + y_k r_k^T - \frac{\langle \Delta g_k, r_k \rangle}{\langle \Delta g_k, y_k \rangle} y_k y_k^T \right]$

$x_{k+1} \leftarrow x_k - \gamma B_k \nabla f(x_k)$

Compute Storage \mathcal{S} :

$S_{k+1} = \{x_k, \nabla f(x_k), B_k\}$

return x_K

5.3 Theoretical Analysis

Based on our strategy from Section 4, our LOA follows the principles therein, as proved in Appendix I.

Theorem 1. *With the choice of B_{-1} above, Algorithm 2 follows Principles 1-2-3-4-5.*

Another benefit of preserving and enhancing existing algorithms is that their coherent structures allow deriving convergence results, proved later in Appendix J.

³To fit in our mathematical formalism, BFGS and our algorithm would need \mathcal{U} to also take S_k as input. Since this would not affect any of the discussion above, we ignored this for the sake of simplicity.

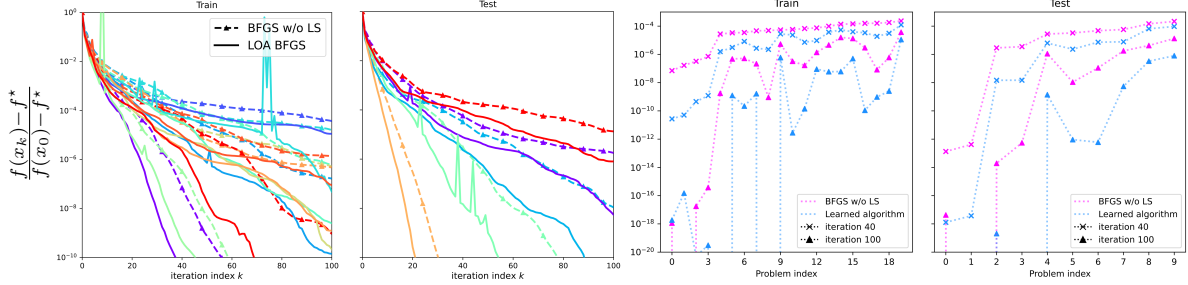


Figure 3: Performance of our learned BFGS method on quadratic functions in dimension $n = 100$ (training and test). Left plots show relative sub-optimality gap against iterations, each color represents a different problem. Right plots: relative sub-optimality gap for each problem at several stages (lower is better).

Theorem 2. Assume that f has L -Lipschitz continuous gradient and that for all $k \in \mathbb{N}$, B_k is positive definite with eigenvalues lower and upper-bounded by c , $C > 0$ respectively. Then for any step-size $\gamma \leq \frac{2}{CL}$, $(f(x_k))_{k \in \mathbb{N}}$ converges and $\lim_{k \rightarrow +\infty} \|\nabla f(x_k)\| = 0$.

It is important to note that Theorem 2 is more restrictive than usual convergence theorems. It is indeed based on strong assumptions regarding the eigenvalues of the matrices $(B_k)_{k \in \mathbb{N}}$. Yet, since B_k is constructed based on the output of the model \mathcal{M}_θ , the failure can only come from the learning part of the algorithm. One could thus optionally enforce the assumption by design of \mathcal{M}_θ in the fashion of (Moeller et al., 2019; Heaton et al., 2023). This would put additional restrictions on the model and does not seem necessary in the experiments below.

6 EXPERIMENTS AND PRACTICAL CONSIDERATIONS

In addition to the design choices that we already made to follow our principles (special model, use of ReLU, no bias, etc.), we discuss practical considerations that ease the training of our model \mathcal{M}_θ . In what follows we consider a training set of $D \in \mathbb{N}$ problems indexed by superscripts (f_j, x_0^j, S_0^j) , for $j = 1, \dots, D$. For each problem we run the algorithm for $K \in \mathbb{N}$ iterations and denote by $(x_k^j)_{k \in \{0, \dots, K\}}$ the resulting sequence.

Loss function Our loss function is based on the last values $(f_j(x_K^j))_{j \in \{1, \dots, D\}}$. However, we make it invariant to the optimal value by using the sub-optimality gap $f_j(x_K^j) - f_j^*$ where f_j^* is the minimum of f_j . A key element to take into account in optimization is that the magnitude of function values may heavily vary between problems and across iterations. This can be slightly mitigated by normalizing by the initial sub-optimality gap $f_j(x_0^j) - f_j^*$, however we instead propose to run vanilla BFGS for K iterations as well and normalize by its sub-optimality gap. After averaging over all

problems our loss function is:

$$\mathcal{L}(\theta) = \frac{1}{D} \sum_{j=1}^D \log \left(1 + \frac{f_j(x_K^j(\theta)) - f_j^*}{f_j(\tilde{x}_K^j) - f_j^*} \right), \quad (7)$$

where \tilde{x}_K^j is the K -th iteration of BFGS ran on the same problem. This is related to the idea of Chen et al. (2020) who trained by competing against a baseline, the novelty in (7) is the use of relative function values.

We make the loss even more robust to different magnitudes by applying a $\log(1 + \cdot)$ composition. We emphasize that the algorithm does not make use of f_j^* , which is only used at training time.

Initialization of \mathcal{M}_θ Training L2O models is notoriously difficult as the loss function may quickly explode (Wang et al., 2021). Our approach allows for a specific trick that dramatically stabilizes training. Indeed, according to Section 5.1, BFGS is a special case of our algorithm in which $y_k = d_k$. By initializing the weights of the last FC layer to zero and the linear layer to be $(0, 0, 0, 0, 1, 0)$ one can check that our algorithm is initialized to exactly coincide with BFGS before training. According to (7), the initial value of the loss function is always $\log(2)$ which dramatically stabilizes the training as shown in Figure 7 in Appendix L. To the best of our knowledge this is a new strategy.

Methodology and results We construct a training set of $D = 20$ problems made of ill-conditioned quadratics functions in dimension $n = 100$ with eigenvalues generated at random and random initializations. The details are provided in Appendix K and the execution time of each algorithm is reported in Appendix K.7. We train our model for $K = 40$ iterations. We then evaluate the performance of the algorithm on several settings that differ from the training one: more than 40 iterations, in-distribution test problems, quadratics in larger dimension, and finally on five other problems including logistic and ridge losses, and some real-world datasets

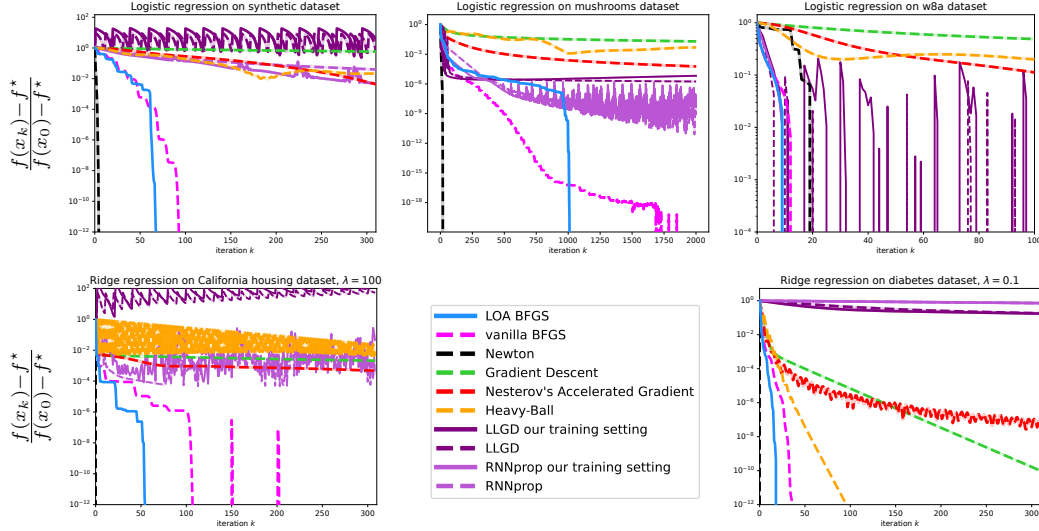


Figure 4: Comparison of our method against BFGS, other hand-crafted algorithms, and L2O methods (see Appendix K.4), on different types of problems (detailed in Appendix K) *without retraining* \mathcal{M}_θ . Our LOA always outperforms vanilla BFGS, evidencing its ability to work far beyond the training setting.

(Figure 4). We compare to several hand-crafted algorithms, as well as L2O baselines (Andrychowicz et al., 2016; Lv et al., 2017), see Appendix K.4. The code for reproducing the results, including the trained weights of our LOA are available in a public repository.⁴

Looking first at the training setting in Figure 3, observe that our L2O model improves upon BFGS for every problem after 40 iterations, sometimes by several orders of magnitudes. This significant improvement transfers to 100 (trained only for 40) iterations for almost all problems, and also to the test set, as well as on quadratic functions in dimension 500 (see Figure 5 in Appendix L).

Figure 4, shows that our LOA does not break on different objective functions and datasets, but even improves upon BFGS despite not having been trained on these. Algorithm 2 is also more robust than the other L2O algorithms considered: they exhibit slow decrease or heavy oscillations on most of the problems of Figure 4. The experiments evidence the benefits of the strategy proposed in Section 4 to achieve generalization.

7 CONCLUSION

We provided a new approach for designing more robust learned optimization algorithms. Our work blends all aspects of L2O: from optimization theory to machine learning models, including implementation and training considerations. We illustrate how promising the approach is in practice by applying our techniques to

build a L2O-enhanced BFGS algorithm. It results in an algorithm outperforming vanilla BFGS consistently beyond the training setting. Enhancing existing algorithms allowed us to provide preliminary theoretical guarantees, which most L2O algorithms lack of, as well as a new training strategy that significantly eases the training and mitigates the difficulty of training L2O models. Our approach is generic and can be applied to almost any algorithm (*e.g.*, HB discussed in Section 3.1). This work thus calls for exploring many directions, such as enhancing other algorithms, designing more advanced models, adding new principles. An important next step is to adapt the principles and the recipe to the case of stochastic algorithms.

Acknowledgments

This work is supported by the ANR-DFG joint project TRINOM-DS under number DFG-OC150/5-1. Part of this work was done while C. Castera was with the department of mathematics of the University of Tübingen, Germany. We thank the anonymous reviewers for their suggestions that helped improving the experiments. C. Castera thanks Michael Sucker for useful discussions and Armin Beck for pointing a mistake in a proof. We also thank the development teams of the following libraries: Python (Rossum, 1995), Matplotlib (Hunter, 2007) and Pytorch (Paszke et al., 2019).

⁴<https://github.com/camcastera/L20toLOA>

References

- Pierre Ablin, Thomas Moreau, Mathurin Massias, and Alexandre Gramfort. Learning step sizes for unfolded sparse coding. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.
- Felipe Alvarez, Hedy Attouch, Jérôme Bolte, and Patrick Redont. A second-order gradient-like dissipative dynamical system with Hessian-driven damping.: Application to optimization and mechanics. *Journal de mathématiques pures et appliquées*, 81(8):747–779, 2002.
- Brandon Amos. Tutorial on amortized optimization. *Foundations and Trends in Machine Learning*, 16(5): 592–732, 2023.
- Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, H. Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 29, 2016.
- Hedy Attouch, Juan Peyrouquet, and Patrick Redont. Fast convex optimization via inertial dynamics with Hessian driven damping. *Journal of Differential Equations*, 261(10):5734–5783, 2016.
- Hedy Attouch, Zaki Chbani, Jalal Fadili, and Hassan Riahi. First-order optimization algorithms via inertial systems with Hessian driven damping. *Mathematical Programming*, 193:1–43, 2022.
- Jonathan Barzilai and Jonathan M Borwein. Two-point step size gradient methods. *IMA journal of numerical analysis*, 8(1):141–148, 1988.
- Stephen Becker and Jalal Fadili. A quasi-Newton proximal splitting method. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 25, 2012.
- Stephen Becker, Jalal Fadili, and Peter Ochs. On quasi-Newton forward-backward splitting: Proximal calculus and convergence. *SIAM Journal on Optimization*, 29(4):2445–2481, 2019.
- Dimitri P Bertsekas. Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334, 1997.
- Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.
- Charles George Broyden. The convergence of a class of double-rank minimization algorithms. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.
- Camille Castera, Hedy Attouch, Jalal Fadili, and Peter Ochs. Continuous Newton-like methods featuring inertia and variable mass. *SIAM Journal on Optimization*, 34(1):251–277, 2024.
- Dongdong Chen, Julián Tachella, and Mike E Davies. Equivariant imaging: Learning beyond the range space. In *International Conference on Computer Vision (ICCV)*, pages 4379–4388, 2021.
- Dongdong Chen, Mike Davies, Matthias J Ehrhardt, Carola-Bibiane Schönlieb, Ferdia Sherry, and Julián Tachella. Imaging with equivariant deep learning: From unrolled network design to fully unsupervised learning. *IEEE Signal Processing Magazine*, 40(1): 134–147, 2023.
- Tianlong Chen, Weiyi Zhang, Zhou Jingyang, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. Training stronger baselines for learning to optimize. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 7332–7343, 2020.
- Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *Journal of Machine Learning Research*, 23(189):1–59, 2022.
- Andrew R Conn, Nicholas IM Gould, and Philippe L Toint. Convergence of quasi-Newton matrices generated by the symmetric rank one update. *Mathematical Programming*, 50(1):177–195, 1991.
- John E Dennis, Jr and Jorge J Moré. Quasi-Newton methods, motivation and theory. *SIAM review*, 19(1):46–89, 1977.
- Roger Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, 1970.
- Roger Fletcher. Newton-like methods. In *Practical Methods of Optimization*, chapter 3, pages 44–79. John Wiley & Sons, Ltd, 2000.
- Guillaume Garrigos and Robert M Gower. Handbook of convergence theorems for (stochastic) gradient methods. *arXiv preprint arXiv:2301.11235*, 2023.
- Erik Gärtner, Luke Metz, Mykhaylo Andriluka, C Daniel Freeman, and Cristian Sminchisescu. Transformer-based learned optimization. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11970–11979, 2023.
- Euhanna Ghadimi, Hamid Reza Feyzmahdavian, and Mikael Johansson. Global convergence of the heavy-ball method for convex optimization. In *European Control Conference (ECC)*, pages 310–315. IEEE, 2015.
- Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24(109):23–26, 1970.

- John Greenstadt. Variations on variable-metric methods. *Mathematics of Computation*, 24(109):1–22, 1970.
- Karol Gregor and Yann LeCun. Learning fast approximations of sparse coding. In *International Conference on Machine Learning (ICML)*, pages 399–406, 2010.
- Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.
- Howard Heaton, Xiaohan Chen, Zhangyang Wang, and Wotao Yin. Safeguarded learned convex optimization. In *AAAI Conference on Artificial Intelligence*, volume 37, pages 7848–7855, 2023.
- Philipp Hennig and Martin Kiefel. Quasi-Newton methods: A new direction. *Journal of Machine Learning Research*, 14(1):843–865, 2013.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- Yunshi Huang, Emilie Chouzenoux, and Jean-Christophe Pesquet. Unrolled variational bayesian algorithm for image blind deconvolution. *IEEE Transactions on Image Processing*, 32:430–445, 2022.
- John D Hunter. Matplotlib: A 2D graphics environment. *Computing in science & engineering*, 9(3): 90–95, 2007.
- Michael J Hutchinson, Charline Le Lan, Sheheryar Zaidi, Emilien Dupont, Yee Whye Teh, and Hyunjik Kim. Lietransformer: Equivariant self-attention for lie groups. In *International Conference on Machine Learning (ICML)*, pages 4533–4543, 2021.
- Samy Jelassi and Yuanzhi Li. Towards understanding how momentum improves generalization in deep learning. In *International Conference on Machine Learning (ICML)*, pages 9965–10040, 2022.
- Qiujiang Jin, Alec Koppel, Ketan Rajawat, and Aryan Mokhtari. Sharpened quasi-Newton methods: Faster superlinear rate and larger local convergence neighborhood. In *International Conference on Machine Learning (ICML)*, pages 10228–10250, 2022.
- Nicolas Keriven and Samuel Vaiter. What functions can graph neural networks compute on random graphs? the role of positional encoding. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2024.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, volume 3, 2015.
- Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning (ICML)*, pages 3744–3753, 2019.
- Eitan Levin and Mateo Díaz. Any-dimensional equivariant neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 2773–2781, 2024.
- Ke Li and Jitendra Malik. Learning to optimize. In *International Conference on Learning Representations (ICLR)*, volume 4, 2016.
- Maojia Li, Jialin Liu, and Wotao Yin. Learning to combine quasi-Newton methods. *arXiv preprint arXiv:2210.06171*, 2023.
- Isaac Liao, Rumen Dangovski, Jakob Nicolaus Foerster, and Marin Soljacic. Learning to optimize quasi-Newton methods. *Transactions on Machine Learning Research*, 2023.
- Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.
- Jialin Liu and Xiaohan Chen. ALISTA: Analytic weights are as good as learned weights in LISTA. In *International Conference on Learning Representations (ICLR)*, volume 7, 2019.
- Jialin Liu, Xiaohan Chen, Zhangyang Wang, Wotao Yin, and HanQin Cai. Towards constituting mathematical structures for learning to optimize. In *International Conference on Machine Learning (ICML)*, pages 21426–21449, 2023.
- Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. In *International Conference on Machine Learning (ICML)*, pages 2247–2255, 2017.
- James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning (ICML)*, pages 2408–2417, 2015.
- Andrea Martin and Luca Furieri. Learning to optimize with convergence guarantees using nonlinear system theory. *IEEE Control Systems Letters*, 8:1355–1360, 2024.
- Tim Meinhardt, Michael Moller, Caner Hazirbas, and Daniel Cremers. Learning proximal operators: Using denoising networks for regularizing inverse imaging

- problems. In *International Conference on Computer Vision (ICCV)*, pages 1781–1790, 2017.
- Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning (ICML)*, pages 4556–4565, 2019.
- Luke Metz, C Daniel Freeman, James Harrison, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Practical tradeoffs between memory, compute, and performance in learned optimizers. In *Conference on Lifelong Learning Agents*, pages 142–164, 2022.
- Michael Moeller, Thomas Mollenhoff, and Daniel Cremers. Controlling neural networks via energy dissipation. In *International Conference on Computer Vision (ICCV)*, pages 3256–3265, 2019.
- Oskar Nordenfors, Fredrik Ohlsson, and Axel Flinthe. Optimization dynamics of equivariant and augmented neural networks. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856.
- Yann Ollivier, Ludovic Arnold, Anne Auger, and Nikolaus Hansen. Information-geometric optimization algorithms: A unifying picture via invariance principles. *Journal of Machine Learning Research*, 18(18):1–65, 2017.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.
- Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
- Boris T Polyak. *Introduction to optimization*. New York, Optimization Software, 1987.
- Michael JD Powell. Variable metric methods for constrained optimization. *Mathematical Programming The State of the Art*, pages 288–311, 1983.
- Ge Ren-Pu and Michael JD Powell. The convergence of variable metric matrices in unconstrained optimization. *Mathematical Programming*, 27:123–143, 1983.
- Anton Rodomanov and Yurii Nesterov. Greedy quasi-Newton methods with explicit superlinear convergence. *SIAM Journal on Optimization*, 31(1):785–811, 2021.
- David W Romero and Jean-Baptiste Cordonnier. Group equivariant stand-alone self-attention for vision. In *International Conference on Learning Representations (ICLR)*, volume 8, 2020.
- Guido Rossum. *Python reference manual*. CWI (Centre for Mathematics and Computer Science), 1995.
- Rajiv Sambharya, Georgina Hall, Brandon Amos, and Bartolomeo Stellato. Learning to warm-start fixed-point optimization algorithms. *Journal of Machine Learning Research*, 25(166):1–46, 2024.
- David F Shanno. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation*, 24(111):647–656, 1970.
- Prabhu Teja Sivaprasad, Florian Mai, Thijs Vogels, Martin Jaggi, and François Fleuret. Optimizer benchmarking needs to account for hyperparameter tuning. In *International Conference on Machine Learning (ICML)*, pages 9036–9045, 2020.
- Michael Sucker and Peter Ochs. PAC-Bayesian learning of optimization algorithms. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 8145–8164, 2023.
- Hong Ye Tan, Subhadip Mukherjee, Junqi Tang, and Carola-Bibiane Schönlieb. Boosting data-driven mirror descent with randomization, equivariance, and acceleration. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.
- Matthieu Terris, Thomas Moreau, Nelly Pustelnik, and Julian Tachella. Equivariant plug-and-play image reconstruction. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 25255–25264, 2024.
- Philippe L Toint. A sparse quasi-Newton update derived variationally with a nondiagonally weighted Frobenius norm. *mathematics of computation*, 37(156):425–433, 1981.
- Singanallur V Venkatakrishnan, Charles A Bouman, and Brendt Wohlberg. Plug-and-play priors for model based reconstruction. In *IEEE Global Conference on Signal and Information Processing*, pages 945–948, 2013.
- Stéfan van der Walt, Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.
- Shida Wang, Jalal Fadili, and Peter Ochs. Inertial quasi-Newton methods for monotone inclusion: efficient resolvent calculus and primal-dual methods. *arXiv preprint arXiv:2209.14019*, 2022.
- Xiang Wang, Shuai Yuan, Chenwei Wu, and Rong Ge. Guarantees for tuning the step size using a learning-to-learn approach. In *International Conference on Machine Learning (ICML)*, pages 10981–10990, 2021.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International Conference on Machine Learning (ICML)*, pages 3751–3760, 2017.

David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

SK Zavriev and FV Kostyuk. Heavy-ball method in nonconvex optimization problems. *Computational Mathematics and Modeling*, 4(4):336–341, 1993.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021a.

Kai Zhang, Yawei Li, Wangmeng Zuo, Lei Zhang, Luc Van Gool, and Radu Timofte. Plug-and-play image restoration with deep denoiser prior. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(10):6360–6376, 2021b.

CHECKLIST

1. For all models and algorithms presented, check if you include:
 - (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [Yes] Models and algorithms described in main text and Appendix.
 - (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes] Theorem stating the properties of the algorithm and principles it follows.
 - (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Yes] <https://github.com/camcastera/L20toL0A>.
2. For any theoretical claim, check if you include:
 - (a) Statements of the full set of assumptions of all theoretical results. [Yes] See Theorems 1 and Theorem 2.
 - (b) Complete proofs of all theoretical results. [Yes] Provided in appendix (Section I and J)
 - (c) Clear explanations of any assumptions. [Yes] Right after Theorem 2.
3. For all figures and tables that present empirical results, check if you include:
 - (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [Yes] Appendix and public repository.
 - (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Yes] Dataset explained or provided. Weight of trained model provided in the repository.
 - (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [Not Applicable]
 - (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Yes] Section K.6.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
 - (a) Citations of the creator If your work uses existing assets. [Not Applicable]
 - (b) The license information of the assets, if applicable. [Not Applicable]
 - (c) New assets either in the supplemental material or as a URL, if applicable. [Not Applicable]
 - (d) Information about consent from data providers/curators. [Not Applicable]
 - (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [Not Applicable]
5. If you used crowdsourcing or conducted research with human subjects, check if you include:
 - (a) The full text of instructions given to participants and screenshots. [Not Applicable]
 - (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [Not Applicable]
 - (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [Not Applicable]

Supplementary Material

H DETAILED ANALYSIS OF EQUIVARIANCE OF POPULAR ALGORITHMS

Below we detail how to obtain the properties listed in Table 1. We begin by studying the transformations.

H.1 The Chain-rule

In this section we detail how each transformation affects the derivatives of f . All these results are based on the chain-rule. For an invertible mapping $\mathcal{T}: \mathbb{R}^n \rightarrow \mathbb{R}^n$, we study the function $\hat{f} = f \circ \mathcal{T}^{-1}$. The chain rule states that $\forall y \in \mathbb{R}^n$

$$D_y(\hat{f}) = D_y(f \circ \mathcal{T}^{-1}) = D_{\mathcal{T}^{-1}(y)}(f) \cdot D_y(\mathcal{T}^{-1}),$$

where $D_y(\hat{f})$ is the Jacobian of \hat{f} at y . Rewriting this in terms of gradients (the transpose of the Jacobian):

$$\nabla \hat{f}(y) = (D_y(\mathcal{T}^{-1}))^T \nabla f(\mathcal{T}^{-1}(y)).$$

In most of what follows we will apply the chain rule above to the point $\hat{x} = \mathcal{T}(x)$, which yields

$$\nabla \hat{f}(\hat{x}) = (D_{\mathcal{T}(x)}(\mathcal{T}^{-1}))^T \nabla f(\mathcal{T}^{-1}(\mathcal{T}(x))) = (D_{\mathcal{T}(x)}(\mathcal{T}^{-1}))^T \nabla f(x), \quad (8)$$

so the Jacobian of \mathcal{T}^{-1} captures how the gradient is transformed. We now detail this for each transformation.

H.2 List of Transformations

We consider the transformations in Table 1. For each case we redefine \mathcal{T} and, without restating it, define $\hat{f} = f \circ \mathcal{T}^{-1}$ and $\hat{x} = \mathcal{T}(x)$, for all $x \in \mathbb{R}^n$.

Translation. Let $v \in \mathbb{R}^n$ and for all $x \in \mathbb{R}^n$, consider the translation $\mathcal{T}(x) = x + v$. Then one can see that $D(\mathcal{T}^{-1}) = \mathbb{I}_n$ which implies that $(D_{\mathcal{T}(x)}(\mathcal{T}^{-1}))^T = \mathbb{I}_n$. So, $\nabla \hat{f}(\hat{x}) = \nabla f(x)$ and similarly, one can show that $\nabla^2 \hat{f}(\hat{x}) = \nabla^2 f(x)$.

Orthogonal Linear Transformations. Let $P \in \mathbb{R}^{n \times n}$ an orthogonal matrix ($P^T P = \mathbb{I}_n$) and $\mathcal{T}: x \in \mathbb{R}^n \mapsto Px$. Then using the orthogonality of P , $\mathcal{T}^{-1}(x) = P^{-1}x = P^T x$. It is a linear mapping, so $D\mathcal{T}^{-1} = P^T$ and $(D_{\mathcal{T}(x)}(\mathcal{T}^{-1}))^T = (P^T)^T = P$. Therefore $\nabla \hat{f}(\hat{x}) = P \nabla f(x)$. Similarly, using the linearity of \mathcal{T}^{-1} , we can show that $\nabla^2 \hat{f}(\hat{x}) = P \nabla^2 f(x) P^T$.

Permutations. Permutation matrices are a specific type of orthogonal matrices, therefore the above directly applies.

Geometric Rescaling. Let $\lambda > 0$ and consider the transformation $\mathcal{T}: x \in \mathbb{R}^n \mapsto \lambda x$. Then $\mathcal{T}^{-1}(x) = \frac{1}{\lambda}x$ which is again a linear mapping so $D\mathcal{T}^{-1} = \frac{1}{\lambda}\mathbb{I}_n$. We deduce as before that $\nabla \hat{f}(\hat{x}) = \frac{1}{\lambda}\nabla f(x)$ and $\nabla^2 \hat{f}(\hat{x}) = \frac{1}{\lambda^2}\nabla^2 f(x)$.

Function Rescaling. Let $\lambda > 0$, when considering $\hat{f} = \lambda f$, the linearity of the differentiation directly gives $\nabla \hat{f} = \lambda \nabla f$ and $\nabla^2 \hat{f} = \lambda \nabla^2 f$.

H.3 Analysis of Popular Algorithms

We now show the properties of Table 1 for each algorithm therein, except for BFGS which is analyzed together with Algorithm 2 later in Section I. Each time, the proofs are done by induction. We can safely assume that x_0 and S_0 are properly adapted so that the induction holds for $k = 0$ (this was explained in Section 4).

To show each equivariance property (or invariance in the case of function rescaling), we fix $k \in \mathbb{N}$ and assume that the equivariance holds for all iterates up to k , and then prove that it still holds at iteration $k + 1$.

Gradient descent and HB We already extensively discussed the properties of HB which was our running example in Section 4. As for gradient descent, it is simply HB with $\alpha = 0$. Using the results of Section H.2 one can straightforwardly deduce translation, permutations and orthogonal equivariances. Thus we only discuss the

case of rescaling. For $\lambda > 0$ and $\hat{f} = f(\frac{1}{\lambda}\cdot)$, assuming that the induction hypothesis $\hat{x}_k = \lambda x_k$ holds, we previously showed that, $\nabla \hat{f}(\hat{x}_k) = \frac{1}{\lambda} \nabla f(x_k)$, so the iteration of HB reads,

$$\hat{x}_{k+1} = \hat{x}_k + \alpha \hat{d}_k + \gamma \nabla \hat{f}(\hat{x}_k) = \lambda x_k + \lambda \alpha d_k + \frac{\gamma}{\lambda} \nabla f(x_k).$$

So for $\lambda \neq 1$ we see that $\hat{x}_{k+1} \neq \lambda x_{k+1}$. This can be fixed however by tuning γ specifically for each problem (we would get $\hat{\gamma} = \lambda^2 \gamma$). The case of function rescaling $\hat{f} = \lambda f$ is almost identical.

Newton's method The update of Newton's method reads

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k).$$

As above, translation equivariance is straightforward. As for orthogonal matrices P , using the results from Section H.2 it holds that

$$\begin{aligned} \hat{x}_{k+1} &= \hat{x}_k - [\nabla^2 \hat{f}(\hat{x}_k)]^{-1} \nabla \hat{f}(\hat{x}_k) = Px_k - [P \nabla^2 f(x_k) P^T]^{-1} P \nabla f(x_k) \\ &= Px_k - (P^T)^{-1} [\nabla^2 f(x_k)]^{-1} P^{-1} P \nabla f(x_k) = Px_k - P [\nabla^2 f(x_k)]^{-1} \nabla f(x_k), \end{aligned}$$

which proves the equivariance.

For geometric rescaling by $\lambda > 0$, remark from Section H.2 that the inverse Hessian is rescaled by λ^2 and the gradient is rescaled by $1/\lambda$, so the result follows. The same is true for invariance with respect to function rescaling.

The ADAM Algorithm The iterations of the ADAM algorithm read

$$\begin{cases} m_k &= \beta_1 m_{k-1} + (1 - \beta_1) \nabla f(x_k) \\ v_k^2 &= \beta_2 v_{k-1}^2 + (1 - \beta_2) \nabla f(x_k) \odot \nabla f(x_k), \\ x_{k+1} &= x_k - \gamma \frac{m_k}{\sqrt{v_k^2}} \end{cases}$$

where $\beta_1, \beta_2 \in [0, 1)$, $\gamma > 0$ is the step-size, \odot denotes the element-wise product, the square root and quotient are applied element wise, and $m_{-1}, v_{-1}^2 \in \mathbb{R}^n$.

Again, translation equivariance is straightforward using the results from section H.2. The robustness with respect to the function rescaling is also easy to check in that case since both m_k and $\sqrt{v_k^2}$ are rescaled like ∇f , hence no need to adapt γ unlike HB and GD. However, for the same reason we see that equivariance w.r.t. geometric rescaling does not hold, except if we adapt γ . We now consider an orthogonal matrix P . According to section H.2 and assuming that up to iteration k ADAM is equivariant to orthogonal transformations, we have $\hat{f}(\hat{x}) = P \nabla f(x)$ and $\hat{m}_k = P m_k$. However, looking at v_k^2 , note that

$$\nabla \hat{f}(\hat{x}_k) \odot \nabla \hat{f}(\hat{x}_k) = (P \nabla f(x_k)) \odot (P \nabla f(x_k)) \quad (9)$$

which in general is not equal to $P(\nabla f(x_k) \odot \nabla f(x_k))$. Therefore, for most orthogonal matrices we do not have $\hat{v}_k = P v_k$ and equivariance does not hold.

Nevertheless, in the special case where P is a permutation matrix, each line of P contains exactly one coefficient equal to one and all others are zero. Since \odot is an element-wise operation, one can check that we then have $(P \nabla f(x_k)) \odot (P \nabla f(x_k)) = P(\nabla f(x_k) \odot \nabla f(x_k))$ such that $\hat{v}_k = P v_k$. Applying the same reasoning to all other element-wise operations in (9), we deduce that, despite not being equivariant to all orthogonal matrices, ADAM is permutation equivariant.

H.4 On the Difficulty of Preserving Orthogonal Equivariance for LOA

The discussion above regarding ADAM shows why preserving equivariance to all orthogonal matrices is very hard for LOA since even element-wise operations may not commute with orthogonal matrices.

To give an example, let P be an orthogonal matrix, let $y \in \mathbb{R}^n$ and $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ be a *non-linear* activation function (to be applied element wise in layers of a neural network). To preserve equivariance with respect to P , one would

want that $\sigma(Py) = P\sigma(y)$, which, for the i -th coordinate reads,

$$\sigma\left(\sum_{j=1}^n P_{i,j}y_j\right) = \sum_{j=1}^n P_{i,j}\sigma(y)_j. \quad (10)$$

Yet, since σ is assumed to be non-linear, it does not commute with the sum (which would still not even be sufficient for (10) to hold). Therefore, if y is the output of an FC layer (used coordinate-wise like in Algorithm 2) in a neural network, then applying a non-linear activation function (*e.g.*, ReLU or sigmoid), we see that equivariance with respect to P is broken. This shows that orthogonal equivariance is not even compatible with coordinate-wise FC layers and hence hardly possible to achieve for LOA.

Finally, note that when P is a permutation matrix, then $\forall i \in \{1, \dots, n\}$, there exists $l \in \{1, \dots, n\}$ such that $P_{i,l} = 1$ and for all $j \neq l$, $P_{i,j} = 0$. So (10) becomes

$$\sigma(P_{i,l}y_l) = P_{i,l}\sigma(y)_l \iff \sigma(y_l) = \sigma(y)_l, \quad (11)$$

and since σ is applied element wise, (11) holds true. So permutation equivariance is more compatible with LOA than orthogonal equivariance.

I PROOF OF THEOREM 1

Proof of Theorem 1. Principle 1 holds by construction of the algorithm where n is not used to choose p nor n_i . We now prove that Principles 2 to 5 hold one by one. As for other algorithms in Section H.3, in each case we explicitly state which transformation \mathcal{T} is considered and implicitly redefine $\hat{f} = f \circ \mathcal{T}^{-1}$ and define $(\hat{x}_k)_{k \in \mathbb{N}}$ as the iterates of the algorithm applied to $(\hat{f}, \hat{x}_0, \hat{S}_0)$ (all quantities with a “hat” symbol are defined accordingly). We again proceed by induction: we fix $k \in \mathbb{N}$ and assume that equivariance (or invariance for Principle 5) holds up to iteration k and show that it still holds at iteration $k+1$. We also show that the principles hold at $k=0$ by construction.

Unlike the algorithms discussed in Section H.3, Algorithm 2 and BFGS additionally use a matrix B_k , (stored in the state S_{k+1}). Since B_k aims to approximate the inverse Hessian $\nabla^2 f(x_k)^{-1}$, we expect B_k to be transformed by \mathcal{T} in the same way as $\nabla^2 f(x_k)^{-1}$ is (see Section H.2). We will prove that this is the case, again by induction.

Translation. Let $v \in \mathbb{R}^n$ and the translation $\mathcal{T}: x \in \mathbb{R}^n \mapsto x + v$. Assume that $\forall i \leq k$, $\hat{x}_i = \mathcal{T}(x_i) = x_i + v$ and that $\forall i \leq k-1$, $\hat{B}_i = B_i$. Then $\hat{d}_k = \hat{x}_k - \hat{x}_{k-1} = d_k$ and we showed in Section H.2 that $\nabla \hat{f}(\hat{x}_k) = \nabla f(x_k)$. Similarly, $\widehat{\Delta g}_k = \Delta g_k$. So,

$$\hat{I}_k = \left(\widehat{B}_{k-1} \widehat{\Delta g}_k, \hat{d}_k, -\gamma \widehat{B}_{k-1} \nabla \hat{f}(\hat{x}_k) \right) = (B_{k-1} \Delta g_k, d_k, -\gamma B_{k-1} \nabla f(x_k)) = I_k,$$

This is not surprising as we explained in Section 4 that we constructed \mathcal{C} so that the above is true. Then we directly deduce $\hat{y}_k = \mathcal{M}_\theta(\hat{I}_k) = \mathcal{M}_\theta(I_k) = y_k$ and the rest of the proof follows.

As for the case $k=0$, by construction (see Section 4), $\hat{x}_0 = x_0 + v$ and $\hat{x}_{-1} = x_{-1} + v$. One can then easily check that our choice of B_{-1} (defined in Section 5.2) is translation invariant. So by induction, Principle 2 holds.

Permutation.

Let P a permutation matrix of \mathbb{R}^n and let $\mathcal{T}: x \in \mathbb{R}^n \mapsto Px$. Assume that $\forall i \leq k$, $\hat{x}_i = \mathcal{T}(x_i) = Px_i$ and that $\forall i \leq k-1$, $\hat{B}_i = PB_i P^T$. Note that the hypothesis on B_i matches that of the inverse Hessian in Section H.2. Then $\hat{d}_k = Px_k - Px_{k-1} = Pd_k$. We showed in Section H.2 that $\nabla \hat{f}(\hat{x}_k) = P \nabla f(x_k)$, and similarly, $\widehat{\Delta g}_k = P \Delta g_k$. So,

$$\hat{I}_k = (PB_{k-1} P^T P \Delta g_k, Pd_k, -\gamma PB_{k-1} P^T P \nabla f(x_k)) = PI_k,$$

i.e., \mathcal{C} is permutation equivariant as intended. Then $\hat{y}_k = \mathcal{M}_\theta(\hat{I}_k) = \mathcal{M}_\theta(PI_k)$ and as justified in Appendix H.4, all the operations applied element-wise in \mathcal{M}_θ are permutation equivariant, and the averaging also is. So \mathcal{M}_θ is permutation equivariant, *i.e.*, $\hat{y}_k = Py_k$.

Regarding the step \mathcal{U} , we recall the notation $r_k = d_k - B_{k-1} \Delta g_k$ used in (6). Remark that $\hat{r}_k = Pd_k - PB_{k-1} \Delta g_k = Pr_k$, and substituting \hat{y}_k , $\widehat{\Delta g}_k$ and \hat{r}_k in (6) (and using again $P^T P = \mathbb{I}_n$), we obtain $\hat{B}_k = PB_k P^T$ and $\hat{x}_{k+1} = Px_{k+1}$.

Finally, at $k = 0$, by construction $\hat{x}_0 = Px_0$ and $\hat{x}_{-1} = Px_{-1}$ and one can easily check that $\hat{\gamma}_{\text{BB}}^{(0)} = \gamma_{\text{BB}}^{(0)}$ (again due to P being orthogonal), such that $\hat{B}_{-1} = B_{-1}$. So Principle 3 holds true.

Geometric rescaling. Let $\lambda > 0$ and let $\mathcal{T}: x \in \mathbb{R}^n \mapsto \lambda x$. Assume that $\forall i \leq k$, $\hat{x}_i = \mathcal{T}(x_i) = \lambda x_i$ and that $\forall i \leq k-1$, $\hat{B}_i = \lambda^2 B_i$ (as in Section H.2). Then $\hat{d}_k = \lambda x_k - \lambda x_{k-1} = \lambda d_k$. We also showed in Section H.2 that $\nabla \hat{f}(\hat{x}_k) = \frac{1}{\lambda} \nabla f(x_k)$, thus $\widehat{\Delta g}_k = \frac{1}{\lambda} \Delta g_k$. So,

$$\hat{I}_k = \left(\lambda^2 B_{k-1} \frac{1}{\lambda} \Delta g_k, \lambda d_k, -\gamma \lambda^2 B_{k-1} \frac{1}{\lambda} \nabla f(x_k) \right) = \lambda I_k,$$

which means that \mathcal{C} is equivariant as we prescribed. Then our model \mathcal{M}_θ is a composition of linear operations and ReLU activation functions which are all equivariant to rescaling, so the model is equivariant, *i.e.*, $\hat{y}_k = \lambda y_k$. Plugging this into the update step we obtain

$$\hat{B}_k = \lambda^2 B_{k-1} + \frac{1}{\langle \lambda^{-1} \Delta g_k, \lambda y_k \rangle} \left[\lambda^2 r_k y_k^T + \lambda^2 y_k r_k^T - \frac{\langle \lambda^{-1} \Delta g_k, \lambda r_k \rangle}{\langle \lambda^{-1} \Delta g_k, \lambda y_k \rangle} \lambda^2 y_k y_k^T \right] = \lambda^2 B_k.$$

Finally, the case $k = 0$ holds by construction and thanks to the BB step-size since

$$\hat{\gamma}_{\text{BB}}^{(0)} = \frac{\langle \lambda^{-1} \Delta g_0, \lambda d_0 \rangle}{\lambda^{-2} \|\Delta g_0\|^2} = \lambda^2 \gamma_{\text{BB}}^{(0)}.$$

This shows how the choice of B_{-1} is crucial to preserve equivariance to rescaling. Overall Principle 4 holds.

Function rescaling. Let $\lambda > 0$ and consider $\hat{f} = \lambda f$. For this last principle we want to prove invariance of the algorithm. Therefore assume that $\forall i \leq k$, $\hat{x}_i = x_i$ and that $\forall i \leq k-1$, $\hat{B}_i = \frac{1}{\lambda} B_i$ (it scales like the inverse Hessian). Then $\hat{d}_k = d_k$ and we also have $\nabla \hat{f}(\hat{x}_k) = \lambda \nabla f(x_k)$, thus $\widehat{\Delta g}_k = \lambda \Delta g_k$. So

$$\hat{I}_k = \left(\frac{1}{\lambda} B_{k-1} \lambda \Delta g_k, d_k, -\gamma \frac{1}{\lambda} B_{k-1} \lambda \nabla f(x_k) \right) = I_k,$$

so \mathcal{C} is invariant, which directly implies $\hat{y}_k = y_k$ and then

$$\hat{B}_k = \frac{1}{\lambda} B_{k-1} + \frac{1}{\langle \lambda \Delta g_k, y_k \rangle} \left[r_k y_k^T + y_k r_k^T - \frac{\langle \lambda \Delta g_k, r_k \rangle}{\langle \lambda \Delta g_k, y_k \rangle} y_k y_k^T \right] = \frac{1}{\lambda} B_k.$$

Finally, the case $k = 0$ holds again thanks to the use of the BB step-size to initialize B_{-1} , which proves that Principle 5 holds and concludes the proof. \square

Remark 2. The proof above can easily be applied to BFGS since it corresponds to the special case where \mathcal{M}_θ is replaced by $y_k = d_k$.

J PROOF OF THEOREM 2

Proof of Theorem 2. Assume that f has L -Lipschitz continuous gradient, that is, for all $x, y \in \mathbb{R}^n$,

$$\|\nabla f(x) - \nabla f(y)\| \leq L \|x - y\|.$$

Then the *descent lemma* (see *e.g.*, Garrigos and Gower (2023)) states that for all $x, y \in \mathbb{R}^n$,

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2} \|y - x\|^2. \quad (12)$$

Now let $(x_k)_{k \in \mathbb{N}}$ and $(B_k)_{k \in \mathbb{N}}$ be respectively the sequence of iterates and the matrices generated by Algorithm 2 applied to (f, x_0, S_0) . Using the descent lemma (12), we get,

$$f(x_{k+1}) \leq f(x_k) + \langle \nabla f(x_k), -\gamma B_k \nabla f(x_k) \rangle + \frac{L}{2} \gamma^2 \|B_k \nabla f(x_k)\|^2,$$

which we rewrite

$$f(x_{k+1}) \leq f(x_k) + \left\langle B_k \nabla f(x_k), -\gamma \nabla f(x_k) + \frac{L}{2} \gamma^2 B_k \nabla f(x_k) \right\rangle. \quad (13)$$

By construction, (see (6)), B_k is real symmetric, so there exists an orthogonal matrix $P_k \in \mathbb{R}^{n \times n}$ and a diagonal matrix $D_k \in \mathbb{R}^{n \times n}$ such that,

$$B_k = P_k D_k P_k^T.$$

Using this in (13), we obtain

$$f(x_{k+1}) \leq f(x_k) + \left\langle P_k D_k P_k^T \nabla f(x_k), -\gamma P_k P_k^T \nabla f(x_k) + \frac{L}{2} \gamma^2 P_k D_k P_k^T \nabla f(x_k) \right\rangle,$$

where we used the fact that $P_k P_k^T = \mathbb{I}_n$ to write $\nabla f(x_k) = P_k P_k^T \nabla f(x_k)$. We denote $g_k = P_k^T \nabla f(x_k)$ and get:

$$\begin{aligned} f(x_{k+1}) &\leq f(x_k) + \left\langle P_k D_k g_k, -\gamma P_k g_k + \frac{L}{2} \gamma^2 P_k D_k g_k \right\rangle \\ &\iff f(x_{k+1}) \leq f(x_k) + \left\langle D_k g_k, -\gamma g_k + \frac{L}{2} \gamma^2 D_k g_k \right\rangle, \end{aligned} \quad (14)$$

where we used the fact that P_k is orthogonal in the last line. Since D_k is orthogonal, denoting by $(g_{k,i})_{i \in \{1, \dots, n\}}$ and $(b_{k,i})_{i \in \{1, \dots, n\}}$ the coordinates of g_k and the eigenvalues of B_k , respectively, we deduce that

$$\begin{aligned} \left\langle D_k g_k, -\gamma g_k + \frac{L}{2} \gamma^2 D_k g_k \right\rangle &= \sum_{i=1}^n b_{k,i} g_{k,i}^2 \left(-\gamma + \frac{L}{2} \gamma^2 b_{k,i} \right) = \gamma \sum_{i=1}^n b_{k,i} g_{k,i}^2 \left(\frac{L}{2} \gamma b_{k,i} - 1 \right) \\ &\leq \gamma \sum_{i=1}^n b_{k,i} g_{k,i}^2 \underbrace{\left(\frac{L}{2} \gamma C - 1 \right)}_{\leq 0} \leq 0, \end{aligned}$$

where for the last line we used the assumption that for all $k \in \mathbb{N}$ and $\forall i \in \{1, \dots, n\}$, $0 < c \leq b_{k,i} \leq C$ and that $\gamma \leq \frac{2}{CL}$. We use this in (14):

$$f(x_{k+1}) \leq f(x_k) - \gamma \left(1 - \frac{L}{2} \gamma C \right) \sum_{i=1}^n b_{k,i} g_{k,i}^2 \leq f(x_k). \quad (15)$$

So the sequence $(f(x_k))_{k \in \mathbb{N}}$ is non-increasing, and since f is a lower-bounded function, then $(f(x_k))_{k \in \mathbb{N}}$ converges.

We now sum (15) from $k = 0$ to $K \in \mathbb{N}$,

$$\begin{aligned} \sum_{k=0}^K f(x_{k+1}) - f(x_k) &\leq -\gamma \left(1 - \frac{L}{2} \gamma C \right) \sum_{k=0}^K \sum_{i=1}^n b_{k,i} g_{k,i}^2 \\ &\iff \gamma \left(1 - \frac{L}{2} \gamma C \right) \sum_{k=0}^K \sum_{i=1}^n b_{k,i} g_{k,i}^2 \leq f(x_0) - f(x_{K+1}). \end{aligned} \quad (16)$$

Since f is lower bounded, the right-hand side of (16) is uniformly bounded, so

$$\gamma \left(1 - \frac{L}{2} \gamma C \right) \sum_{k=0}^{+\infty} \sum_{i=1}^n b_{k,i} g_{k,i}^2 < +\infty \iff \sum_{k=0}^{+\infty} \langle B_k \nabla f(x_k), \nabla f(x_k) \rangle < +\infty. \quad (17)$$

Finally, by assumption B_k is positive definite with eigenvalues uniformly lower-bounded by $c > 0$, therefore (17) implies that $\sum_{k=0}^{+\infty} c \|\nabla f(x_k)\|^2 < +\infty$, and thus in particular $\lim_{k \rightarrow +\infty} \|\nabla f(x_k)\| = 0$.

□

K ADDITIONAL DETAILS ON THE EXPERIMENTS

In this section we provide additional details on how to reproduce the experiments of Section 6.

K.1 The Model of Algorithm 2

The neural network used is exactly that described in Figure 2, we simply detail the FC and linear blocks. The first coordinate-wise FC block is made of 3 layers with output shapes (6, 12, 3), the second-one has 2 layers with output shapes (12, 1). We use ReLU activation functions for each layer except for the last layer of each block. The linear layer is of size 6×1 , again with no bias. The total number of parameter of the network is 216. In comparison, the training set is made of 20 problems in dimension $n = 100$, thus $p = 216$ is much smaller than $20 \times 100 = 2000$. We also apply the algorithm to problems in dimension 500 where even for a single problem $p < 500$.

K.2 Problems and Datasets

Quadratic functions To generate a quadratic function in dimension n , we proceed as follows. We create a matrix A by first sampling its largest and smallest eigenvalues λ_{\min} , λ_{\max} uniformly at random in $[0.1, 1]$ and $[1, 50]$ respectively. We then generate the $n - 2$ other eigenvalues of A uniformly at random in $[\lambda_{\min}, \lambda_{\max}]$. This gives us a diagonal matrix D containing the eigenvalues of A . We then generate another matrix B with Gaussian $\mathcal{N}(0, 1)$ entries and make it symmetric via $B \leftarrow B + B^T$. We then compute the orthogonal matrix P that diagonalizes B and use P to build $A = PDP^T$. We then sample a vector $b \in \mathbb{R}^n$ whose entry are sampled uniformly at random in $[0, 15]$. Our function f finally reads: $f: x \in \mathbb{R}^n \mapsto \frac{1}{2} \|Ax - b\|^2$. The quadratic functions in dimension 500 (Figure 5) are generated with the same process.

With this process, the largest eigenvalue of $\nabla^2 f$ is $\frac{\lambda_{\max}^2}{2}$. In our experiments the largest eigenvalue in any problem is approximately 1159 and the largest condition number (the ratio between the largest and smallest eigenvalues) is approximately 15156, hence our dataset includes ill-conditioned problems.

Regularized Logistic Regression We consider a binary logistic regression problem, as presented in Hastie et al. (2009). For the left plot of Figure 4, we generate two clouds of M points sampled from Gaussian distributions $\mathcal{N}(\mu_1, 1)$ and $\mathcal{N}(\mu_2, 1)$ where μ_1, μ_2 are themselves sampled from $\mathcal{N}(-1, 1)$ and $\mathcal{N}(1, 1)$ respectively. We store the coordinates of the $2M$ points in a matrix $A \in \mathbb{R}^{2M \times (n+1)}$ (a row of ones is concatenated with A , see (Hastie et al., 2009)). We also create a vector $b \in \mathbb{R}^{2M}$ where each b_i takes either the value 0 or 1 depending on which class the i -th data point belong to. Given these A and b , for all $x \in \mathbb{R}^{n+1}$, the function f is defined as:

$$f(x) = \frac{1}{2M} \sum_{i=1}^{2M} \log \left(1 + e^{x^T A_i} \right) - b_i x^T A_i + \frac{\eta}{2} \|x\|^2.$$

The last term is a regularization that makes the problem strongly convex. We use a very small $\eta = 10^{-3}$. Our experiments are done for $M = 100$ and $n = 50$.

The experiments on the `w8a` and `mushrooms` datasets (also on Figure 4) are also logistic regression experiments but with real-world data. The two datasets are publicly available but also provided in our public repository.

Ridge regression Ridge regression consists in minimizing

$$f(x) = \|Ax - b\|^2 + \frac{\lambda}{2} \|x\|^2,$$

where $\lambda > 0$ is a regularization parameter, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. This is also a type of quadratic function, however the two experiments on the right of Figure 4 are based on the `California housing` and `diabetes` datasets, which are real-world applications, hence different from the synthetic data we trained on. Moreover, in these datasets, the matrix A is usually not square and $m > n$, which makes the solution to $Ax = b$ not unique. This also differs from the training setting.

K.3 The BFGS Baseline

For a fair comparison, the BFGS algorithm is implemented exactly like Algorithm 2 but with $y_k = d_k$ instead of using learned model. We use the same strategy for initializing B_{-1} . For both algorithms we generate a random starting point $x_{-1} \in \mathbb{R}^n$, and perform a gradient descent step along $\nabla f(x_{-1})$ to obtain the true initialization

$x_0 \in \mathbb{R}^n$. Both algorithms thus always start at the same x_0 with the same state $S_0 = \{x_{-1}, \nabla f(x_{-1}), B_{-1}\}$. When using our algorithm with fixed step-size (during training and in most experiments), we compare it to BFGS with fixed step-size. When BFGS is used with line-search, then so is our algorithm.

K.4 The L2O Baselines

We consider two popular L2O baselines: the method from Andrychowicz et al. (2016), referred to as LLGD and the RNNprop method from Lv et al. (2017). The implementation provided in our public repository is adapted from Liu et al. (2023) <https://github.com/xhchrn/MS4L2O>. We consider two version of each L2O method: one trained in the training setting from the aforementioned papers, and one retrained on a setting closer to ours (deterministic, quadratic functions in dimension $n = 100$). Our repository contains the weights used for each model.

K.5 Training Strategy

Training set. Our training dataset is made of 10 quadratic functions in dimension $n = 100$ created following the strategy described in Section K.2. We generate two different initializations at random for each function, yielding a training dataset of 20 problems.

Initialization of the network. We initialize the parameter θ (the weights of our layers) by following the new initialization strategy introduced in Section 6. Recall that with this strategy, before training θ our model coincides with BFGS, stabilizing the training process as shown on Figure 7.

Training by unrolling. We run the algorithm for $K = 40$ iterations and use the loss function $\mathcal{L}(\theta)$ described in (7). However, we observe in practice that unrolling the last iterate (*i.e.*, computing the gradient of \mathcal{L} with respect to the last iterate K) is numerically unstable (known as the vanishing/exploding gradient problem). We mitigate this issue by computing $\mathcal{L}(\theta)$ every 5 iterations (*i.e.*, at iterations $\{5, 10, \dots, 40\}$) and by “detaching” the matrix B_k every 5 iterations (*i.e.*, neglecting the effect that old predictions of the model have on B_k). We then average the 8 values of the loss computed along the trajectory and “back-propagate” to compute the gradient of this loss function. Since we neglect the influence that old iterates have on B_k , we do not compute “true gradients” of \mathcal{L} . Yet, it is important to note that this is acceptable since training is only a mean to obtain a good parameter θ . This does not break any of our principles.

Training parameters. We train the model with the ADAM (Kingma and Ba, 2015) optimizer with gradient clipping. We do not compute the full gradient $\nabla \mathcal{L}(\theta)$ but a mini-batch approximation of it by selecting only two problems at random at every iteration. We use the learning rate 10^{-4} for the FC layers and 10^{-3} for the linear layer. We save the model that achieved the best training loss on average over one epoch (a full pass on the training dataset).

K.6 Computational Architecture

We ran all the experiments on a HP EliteBook 840 with 32 GiB of RAM, and an Intel Core Ultra 5 125U CPU with 12 cores at 4 GHz. No GPUs were used for the experiments. We used Python (Rossum, 1995) 3.12.3, Numpy (Walt et al., 2011) 2.1.2 and Pytorch (Paszke et al., 2019) 2.5.0 running on Ubuntu 24.04.

K.7 Wall-clock Time Estimation

We estimated the average wall-clock time per iteration for each algorithm on each problem. We note that the compute times reported are only estimations that are architecture and implementation dependant and may vary. The results reported in Table 2 were obtained with the architecture described in Section K.6. In the table we report the average cost of one iteration for each algorithm relative to that of gradient descent, *i.e.*, $\text{exetime(algo)}/\text{exetime(GD)}$.

L ADDITIONAL EXPERIMENTS

Compatibility with larger-dimensional problems Figure 5 shows that our LOA still performs well and consistently outperforms vanilla BFGS in dimension $n = 500$ despite having been trained on problems in dimension

Table 2: Average wall-clock time relative to gradient descent

	ridge california	ridge diabetes	logistic w8a	logistic mushrooms	logistic synthetic	average speed-down
Gradient Descent	1.00	1.00	1.00	1.00	1.00	1.00
Nesterov	1.03	1.05	1.00	1.50	0.87	1.09
Heavy-ball	1.10	1.16	1.00	2.38	0.96	1.32
BFGS	2.40	2.99	0.97	1.41	1.70	1.89
L2LbyGD	6.23	2.13	1.22	1.67	1.70	2.59
LOA BFGS	3.51	3.27	1.46	2.35	2.95	2.71
RNNprop	6.96	2.43	1.08	1.28	1.87	2.72
Newton	7.91	10.56	190.23	40.34	8.32	51.47

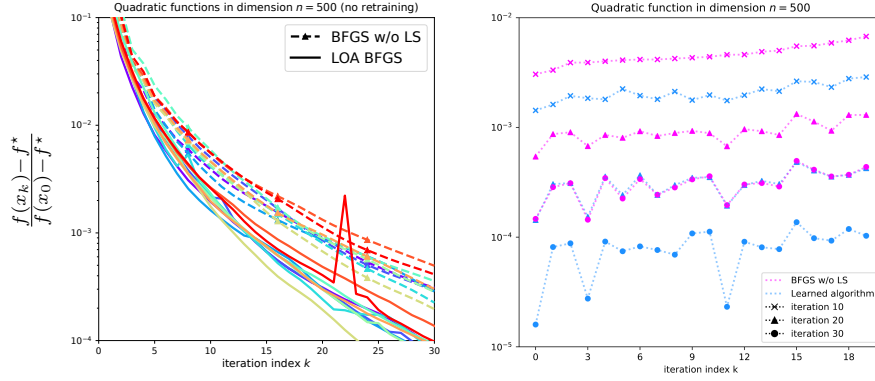


Figure 5: Quadratic problems similar to that of the training setting but used in dimension $n = 500$, *without retraining*. Left: sub-optimality gap against iterations. Right: Relative sub-optimality gap for each problem at several stages of the optimization process.

$n = 100$. We note that the improvement is not as dramatic as in dimension $n = 100$, yet we managed to transfer good performance in much larger problem than those of the training set, which was our main goal in this experiment.

Compatibility with line-search Like Newton’s method, one usually wants to use QN methods with a step-size γ as close as possible to 1. This may however cause numerical instabilities (*e.g.*, in the logistic regression problems). Therefore, QN algorithms, including BFGS are often used with line-search strategy (adapting the step-size based on some rules). It is thus important to evidence that Algorithm 2 performs well when used with line-search strategies, *despite having been trained with fixed step-sizes*. The results in Figure 6 show that Algorithm 2 significantly outperforms BFGS with line-search on almost all problems. This was also the case on the logistic regression problems of Figure 4 where we used line-search for Newton’s method, BFGS and our algorithm. Our LOA thus appears to be highly compatible with line-search.

Benefits of our initialization strategy As mentioned in Section 6, we can easily find a closed-form initialization of the model \mathcal{M}_θ such that Algorithm 2 coincides with BFGS before training. This dramatically stabilizes the training process as shown on Figure 7 where, without this initialization strategy, the average train loss remains large (despite having tuned the learning rate specifically for that setting). This can be explained by the fact that for a random initialization, the value $f(x_K)$ produced by the algorithm will usually be very large, making it necessary to train with small learning rates, whereas with our strategy we start in a more stable region as evidenced by the smaller oscillations in early training, allowing larger learning rates.

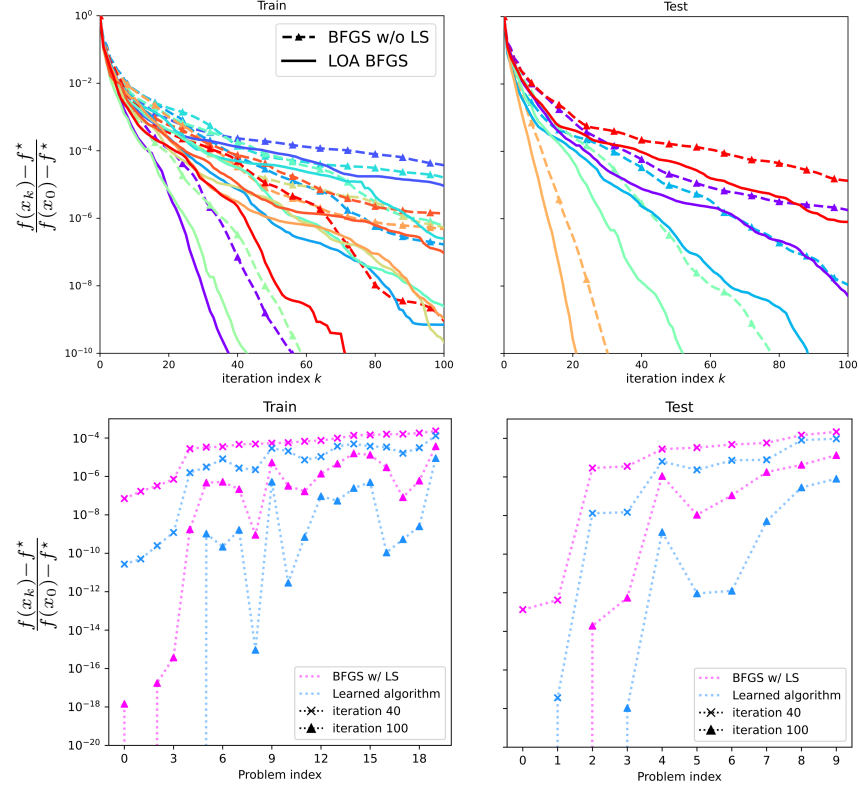


Figure 6: Same experiment as Figure 3 but we evaluate the performance of the algorithm used with line-search, *without retraining* it. Top row: relative sub-optimality gap against iterations on the training and test sets. Each color represents a different problem. Bottom: relative sub-optimality gap for each problem after 40 and 100 iterations.

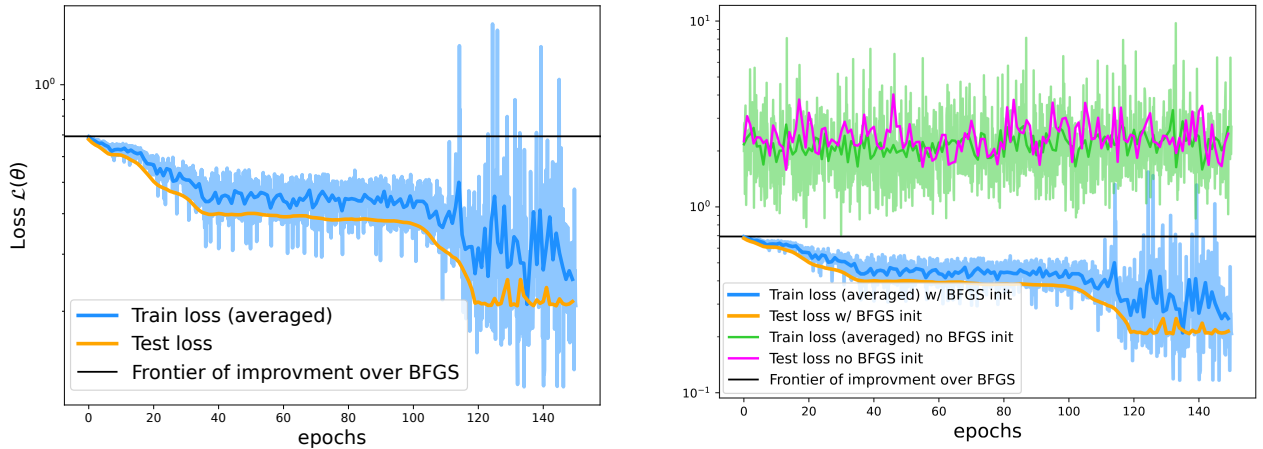


Figure 7: Left: evolution of the training loss and test loss during the training of the model of our algorithm. The blue area shows the value of the stochastic loss and the blue curve represents the average over one epoch. The test loss is computed after each epoch. The black line corresponds to $\log(2)$, any value of $\mathcal{L}(\theta)$ below this line corresponds to an improvement compared to vanilla BFGS. Right: Same figure as the left-hand side but comparison with the case where we did not initialize the model to coincide with BFGS, causing instability in training.