LEARNING THE STEP-SIZE POLICY FOR THE LIMITED-MEMORY BROYDEN-FLETCHER-GOLDFARB-SHANNO ALGORITHM

Anonymous authors
Paper under double-blind review

ABSTRACT

We consider the problem of how to learn a step-size policy for the Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm. This is a limited computational memory quasi-Newton method widely used for deterministic unconstrained optimization but currently avoided in large-scale problems for requiring step sizes to be provided at each iteration. Existing methodologies for the step size selection for L-BFGS use heuristic tuning of design parameters and massive re-evaluations of the objective function and gradient to find appropriate step-lengths. We propose a neural network architecture with local information of the current iterate as the input. The step-length policy is learned from data of similar optimization problems, avoids additional evaluations of the objective function, and guarantees that the output step remains inside a pre-defined interval. The corresponding training procedure is formulated as a stochastic optimization problem using the backpropagation through time algorithm. The performance of the proposed method is evaluated on the MNIST database for handwritten digits. The results show that the proposed algorithm outperforms heuristically tuned optimizers such as ADAM and RMSprop in terms of computational time. It performs comparably to more computationally demanding L-BFGS with backtracking line search. The numerical results also show that the learned policy generalizes better to high-dimensional problems as compared to ADAM and RMSprop, highlighting its potential use in large-scale optimization.

1 INTRODUCTION

Consider the unconstrained optimization problem

$$\underset{\boldsymbol{x}}{\text{minimize}}\ f(\boldsymbol{x}) \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is an objective function that is differentiable for all $\boldsymbol{x} \in \mathbb{R}^n$, with n being the number of decision variables forming $\boldsymbol{x}$. Let $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_0)$ be the gradient of $f(\boldsymbol{x})$ evaluated at some $\boldsymbol{x}_0 \in \mathbb{R}^n$. A general quasi-Newton algorithm for solving this problem iterates

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \boldsymbol{H}_k \boldsymbol{g}_k \tag{2}$$

for an initial $\boldsymbol{x}_0 \in \mathbb{R}^n$. At the k-th iteration, $\boldsymbol{g}_k = \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_k)$ is the gradient, $\boldsymbol{H}_k$ is a positive-definite matrix satisfying the secant equation (Nocedal and Wright, 2006, p. 137) and t_k is the step size.

In this paper, we develop a policy that learns to suitably determine step sizes t_k when the product $\boldsymbol{H}_k \boldsymbol{g}_k$ is calculated by the Limited-Memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm (Liu and Nocedal, 1989). The main contributions of the paper are:

1. We propose a neural network architecture defining this policy taking as input local information of the current iterate. In contrast with more standard strategies, this policy is tuning-free and avoids re-evaluations of the objective function and gradients at each step. The training procedure is formulated as a stochastic optimization problem and can be performed by easily applying backpropagation through time (TBPTT).

2. Training classifiers in the MNIST database (LeCun et al., 1998), our approach is competitive against heuristically tuned optimization procedures. Our tests show that the proposed policy is not only able to outperform competitors such as ADAM and RMSprop in wall-clock time but also performs as good as L-BFGS with backtracking line searches, which is the gold standard.

3. The proposed policy generalized significantly better to higher-dimensional problems when compared to ADAM and RMSprop. This indicates that retraining the policy can be dismissed in some cases, characterizing that learning can be transferred.

This result is a step towards the development of optimization methods that exempts the designer from tuning control parameters as it will be motivated in Section 2. The remaining parts of this paper are organized as follows: Section 3 presents the classical L-BFGS algorithm and discuss some methodologies to determine step sizes; Section 4 contains the architecture for the proposed policy and also discussions on how it was implemented; Section 5 describes the training procedure; and, finally, Section 6 presents experiments using classifiers to operate on MNIST database. The notation is mainly standard. Scalars are plain lower-case letters, vectors are bold lower-case, and matrices are bold upper-case. The clip function is defined as $\mathrm{clip}_l^u(y) := \min(u, \max(l, y))$.

2 Motivation

Most algorithms used in artificial intelligence and statistics are based on optimization theory, which has widely collaborated for the success of machine learning applications in the last decades. However, this two-way bridge seems not to be currently leveraging its full potential in the other sense, that is, to learn how to automate optimization procedures. Indeed, performing satisfactory optimization, or solving learning problems, still relies upon the appropriate tuning of parameters of the chosen algorithm, which are often grouped with other hyper-parameters of the learning task. Despite the existence of several methodologies to obtain good values for these parameters (Bengio, 2000; Bergstra et al., 2011; Bergstra and Bengio, 2012; Snoek et al., 2015; Daniel et al., 2016; Dong et al., 2018), the seek for tuning-free algorithms that perform better than heuristically designed ones is of great interest among practitioner and theoreticians. Indeed, besides the generally-desirable faster convergence, the ready-to-use nature of such algorithms allows the user to focus his attention on other problem-level hyper-parameters while the optimization procedure is automatically performed, resulting in better time and effort allocation. As recent advancements of machine learning have helped automatize the solution of numberless problems, optimization theory should equally benefit from these, balancing the bridge flows.

From a wider viewpoint, most optimization problem requires the user to select an algorithm and tune it to some extent. Although intuition and knowledge about the problem can speed-up these processes, trial-and-error methodologies are often employed which can be a time-consuming and inefficient task. With that in mind, the concept of *Learned optimizers* has been gathering attention in the last few years and, basically, refers to optimization policies and routines that were learned by looking at instances of optimization problems, here called *tasks*. This idea was introduced by Li and Malik (2016) and Andrychowicz et al. (2016) building upon previous results of "learning to learn" or "meta-learning" (Thrun and Pratt, 1998; Hochreiter et al., 2001). In the former, the authors presented an optimization policy based on a neural network trained by reinforcement learning and taking as input the history of gradient vectors at previous iterations. The latter adopts a long short-term memory (LSTM) to achieve a similar task, but the learning is done by truncated backpropagation through time after unrolling the proposed optimizer for a certain number of steps. Subsequently, it was shown in Metz et al. (2019) how multilayer perceptrons (MLP), adequately trained using a combined gradient estimation method, can perform faster in wall-clock time compared to current algorithms of choice. Also within this scenario, in Xu et al. (2019) a reinforcement learning-based methodology to auto-learn an adaptive learning rate is presented. Following this same fashion, in this present paper, instead of completely learning an optimizer from data, we propose a mixture of these ideas into a classical optimization procedure. Thus, the resulting optimizer, composed by a combination of L-BFGS and the proposed policy, will be learned in a constrained domain that assures valuable mathematical properties. The idea is to leverage both frameworks, inheriting the theoretical aspects assured by optimization theory while learning a policy to rule out the hand-design of parameters.

Algorithm 1: L-BFGS algorithm

Input: $\boldsymbol{s}_i = \boldsymbol{x}_{i+1} - \boldsymbol{x}_i$, $\boldsymbol{y}_i = \boldsymbol{g}_{i+1} - \boldsymbol{g}_i$ and $\rho_i = 1/(\boldsymbol{s}_i^T \boldsymbol{y}_i)$ for all $i \in k-m, \dots, k-1$;
and current gradient $\boldsymbol{g}_k$,
Result: update direction $\boldsymbol{d}_k = -\boldsymbol{H}_k \boldsymbol{g}_k$

1 $\boldsymbol{q} \leftarrow \boldsymbol{g}_k$;
2 **for** $i = k-1, \dots, k-m$ **do**
3 $\quad \alpha_i \leftarrow \rho_i \boldsymbol{s}_i^T \boldsymbol{q}$;
4 $\quad \boldsymbol{q} \leftarrow \boldsymbol{q} - \alpha_i \boldsymbol{y}_i$;
5 **end**
6 $\gamma = (\boldsymbol{s}_{k-1}^T \boldsymbol{y}_{k-1})/(\boldsymbol{y}_{k-1}^T \boldsymbol{y}_{k-1})$;
7 $\boldsymbol{r} \leftarrow \gamma \boldsymbol{q}$;
8 **for** $i = k-m, \dots, k-1$ **do**
9 $\quad \beta \leftarrow \rho_i \boldsymbol{y}_i^T \boldsymbol{r}$;
10 $\quad \boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{s}_i(\alpha_i - \beta)$;
11 **end**
12 $\boldsymbol{d}_k \leftarrow -\boldsymbol{r}$;

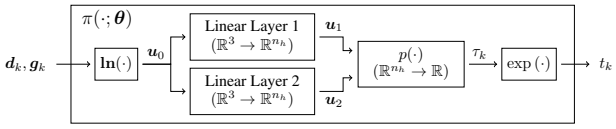

Figure 1: Neural network architecture for the proposed policy to generate step sizes t_k.

3 L-BFGS Algorithm

The L-BFGS algorithm was originally presented in Liu and Nocedal (1989) and is here transcribed into Algorithm 1. It is a quasi-Newton method derived from the BFGS algorithm (Nocedal and Wright, 2006) lowering space complexity from quadratic to linear in the problem dimension at the expense of precision. This algorithm calculates a descending direction in the search space taking into account an estimate of the inverse hessian matrix of $f(\boldsymbol{x})$, given by $\boldsymbol{H}_k$. This matrix is not explicitly constructed but rather the product $\boldsymbol{d}_k := -\boldsymbol{H}_k \boldsymbol{g}_k$ is obtained from the past m values of $\boldsymbol{x}_k$ and $\boldsymbol{g}_k$, which have to be stored. If $f(\boldsymbol{x})$ is convex in $\boldsymbol{x}$, this algorithm is guaranteed to provide a descending update direction, but the same does not apply for non-convex objective functions. However, a simple way to circumvent this is by removing iterations i in lines 2 and 8 of Algorithm 1 such that $\rho_i \leq 0$ (Nocedal and Wright, 2006, p. 537).

A matter of great relevance within this scope is how to choose an appropriate step size t_k to apply the update rule in Eq. (2). To the best of our knowledge, it does not seem to exist a consensus on how to choose t_k in a general way for non-convex objective functions. The scaling factor γ in lines 6-7 of Algorithm 1 is known to assure that the step size $t_k = 1$ is accepted in most iterations but not always. For this reason, a *line search* (LS) procedure is often combined with L-BFGS to assure its convergence. Ideally, this should be performed by solving $t_k = \arg\min_{t>0} f(\boldsymbol{x}_k + t\boldsymbol{d}_k)$ but this exact approach is often too expensive to be adopted, motivating the use of inexact ones. An example is the *backtracking line search* (BTLS), which takes an initial length t_k for the step size and shrinks it repeatedly until the so-called sufficient decrease Wolf Condition $f(\boldsymbol{x}_k + t_k \boldsymbol{d}_k) \leq f(\boldsymbol{x}_k) + c_1 t_k \boldsymbol{g}_k^T \boldsymbol{d}_k$ is fulfilled, where $c_1 \in (0,1)$ is a control parameter to be tuned. Another parameter that has to be designed is the contraction factor $c_2 \in (0,1)$ that shrinks the step size, i.e., $t_k \leftarrow c_2 t_k$, see Nocedal and Wright (2006, p. 37). This method assures convergence to a local-minima at the cost of re-evaluating the objective function several times per iteration. This is a price that the user is, in some cases, willing to pay, but for large-dimensional problems this procedure is likely to become the bottle-neck of the optimization task.

In the context of stochastic optimization, many modified versions of Algorithm 1 together with methodologies for choosing t_k are available (Moritz et al., 2016; Zhou et al., 2017; Bollapragada et al., 2018; Wills and Schön, 2018), but for sake of simplicity, our work will deal exclusively with deterministic non-linear optimization problems.

4 Learned policy for selecting step sizes

Our policy is defined as $t_k = \pi(\boldsymbol{d}_k, \boldsymbol{g}_k; \boldsymbol{\theta})$ and selects an adequate step size for L-BFGS but neither relying on any parameter tuning nor requiring additional evaluations of the objective function.

Instead, its parameters that are represented by $\boldsymbol{\theta}$ should be learned from data. Let us, from now on, define this policy combined with Algorithm 1 as the *L-BFGS-π approach*. The architecture of the policy $\pi(\boldsymbol{d}_k, \boldsymbol{g}_k; \boldsymbol{\theta})$ is shown in Fig. 1. To allow the policy to be independent from the problem size n, only the inner products $\boldsymbol{d}_k^T\boldsymbol{d}_k$, $\boldsymbol{g}_k^T\boldsymbol{g}_k$ and $\boldsymbol{d}_k^T\boldsymbol{g}_k$ are used. These values define the input $\boldsymbol{u}_0 = [\ln(\boldsymbol{d}_k^T\boldsymbol{d}_k) \quad \ln(\boldsymbol{g}_k^T\boldsymbol{g}_k) \quad \ln(-\boldsymbol{d}_k^T\boldsymbol{g}_k)]^T =: \mathbf{ln}(\boldsymbol{d}_k, \boldsymbol{g}_k)$ to two parallel input layers, which are fully connected linear layers that lift information in $\boldsymbol{u}_0$ to a higher dimensional spaces $\mathbb{R}^{n_h}$ (in our tests, we adopted $n_h = 6$). Their outputs, as usual, are defined as $\boldsymbol{u}_1 = \boldsymbol{W}_{01}\boldsymbol{u}_0 + \boldsymbol{b}_{01}$ and $\boldsymbol{u}_2 = \boldsymbol{W}_{02}\boldsymbol{u}_0 + \boldsymbol{b}_{02}$. Notice that $-\boldsymbol{g}_k^T\boldsymbol{d}_k = \boldsymbol{g}_k^T\boldsymbol{H}_k\boldsymbol{g}_k > 0$ as $\boldsymbol{H}_k$ is positive definite, assuring that the last element of $\boldsymbol{u}_0$ is well-defined as a real number. The logarithm operation was adopted to let the linear layers evaluate products and divisions between powers of the inputs by simply summing and subtracting them. Moreover, as the inputs and outputs are all positive, working in the logarithmic vector space allows us to use a wider range of numerical values. Subsequently, let us define the normalized vectors $\bar{\boldsymbol{u}}_1 = \boldsymbol{u}_1/\|\boldsymbol{u}_2\|$ and $\bar{\boldsymbol{u}}_2 = \boldsymbol{u}_2/\|\boldsymbol{u}_2\|$ to calculate the scalar projection of $\bar{\boldsymbol{u}}_1$ onto $\bar{\boldsymbol{u}}_2$ and clip the result to some interval $[\tau_m, \tau_M]$, yielding the *log-step size*

$$\tau_k = \text{clip}_{\tau_m}^{\tau_M}\left(\bar{\boldsymbol{u}}_2^T\bar{\boldsymbol{u}}_1\right) =: p(\boldsymbol{u}_1, \boldsymbol{u}_2) \tag{3}$$

Finally, the selected step size is obtained as $t_k = e^{\tau_k}$. To geometrically interpret this, we sketch three different scenarios in Fig. 2. The dashed lines represent orthogonal axes spanned by some arbitrary $\bar{\boldsymbol{u}}_2$ and the gray strip represents the interval $[\tau_m, \tau_M]$ along the direction of $\bar{\boldsymbol{u}}_2$ whence τ_k should be taken. When the Linear Layer 1 maps $\boldsymbol{u}_0$ into $\boldsymbol{u}_1'$, the scalar projection of $\bar{\boldsymbol{u}}_1'$ onto $\bar{\boldsymbol{u}}_2$ is beyond the maximal τ_M, so τ_k is clipped to it. In the same way, for $\bar{\boldsymbol{u}}_1'''$ the step size will be the minimal one $t_k = e^{\tau_m}$ whereas for the intermediate $\bar{\boldsymbol{u}}_1''$ we have $\tau_k \in (\tau_m, \tau_M)$. The two layers, jointly trained, should learn how to position $\bar{\boldsymbol{u}}_1$ and $\bar{\boldsymbol{u}}_2$ in the lifted space to represent important directional information of $\boldsymbol{d}_k$ and $\boldsymbol{g}_k$ by looking at similar optimization tasks, being thus able to produced suitable step sizes.

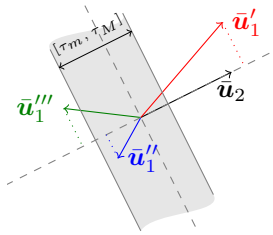

Figure 2: Geometric representation of the scalar projection and clip procedures for 3 cases.

This approach is powerful enough to capture interesting mathematical local properties of this problem. As an instance, it could calculate $\cos\phi_k = -\boldsymbol{d}_k^T\boldsymbol{g}_k/\sqrt{\boldsymbol{d}_k^T\boldsymbol{d}_k\boldsymbol{g}_k^T\boldsymbol{g}_k}$, where ϕ_k is the angle formed between $\boldsymbol{d}_k$ and the steepest descend direction $-\boldsymbol{g}_k$, by letting parameters $\boldsymbol{\theta} := (\boldsymbol{W}_{01}, \boldsymbol{b}_{01}, \boldsymbol{W}_{02}, \boldsymbol{b}_{02})$ and limits τ_m and τ_M be given as described in Appendix A. Indeed, the considered inner products forming $\boldsymbol{u}_0$ are also employed in many procedures for determining step sizes, for example, in the sufficient decrease Wolfe condition for backtracking line search, which makes our policy comparable to them in the sense that $\pi(\cdot; \boldsymbol{\theta})$ does not require additional information to operate.

However, the clip function is not suitable for training given that it is non-differentiable and gradients cannot be backpropagated through it. Fortunately, the clip operation (3) can be cast as a convex optimization problem

$$\tau_k = \arg\min_{\tau\in\mathbb{R}} \|\boldsymbol{u}_2\tau - \boldsymbol{u}_1\|^2 \tag{4}$$

$$\text{s.t.} \quad \tau_m \leq \tau \leq \tau_M \tag{5}$$

allowing τ_k to be calculated by a convex optimization layer, defined here as a *CVX Layer*, (Agrawal et al., 2019). This last layer can output the solution to a parameter-dependent convex optimization problem. For the special case where a solution is not differentiable with respect to the input (e.g., in our case when an inequality constraint is active), the automatic differentiation procedure delivers an heuristic quantity that can be employed as a gradient. The use of a CVX Layer is therefore convenient for training our policy but, on the other hand, using Eq. (3) in its place when applying the already-trained policy significantly speeds up the step size evaluation, compared to solving (4).

It is important to remark that this policy is defined as independent from both the memory length m of Algorithm 1 and the problem dimension n. Additionally, the lower and upper limits for the log-step size are τ_m and τ_M, respectively, and can also be learned. In this work, however, we chose $\tau_m = -3$ and $\tau_M = 0$, letting $t_k \in [0.0497, 1]$. This interval is comprehensive enough to let our method be compared in a fair way to backtracking line searches.

5 TRAINING THE POLICY

The L-BFGS-π procedure can be trained by *truncated backpropagation through time* (TBPTT), in a similar way to Andrychowicz et al. (2016). From this point on, training the optimizer is referred to as the *outer optimization problem* whereas an instance of a task in the form of (1) is called the *inner optimization problem*. Therefore, this outer problem is defined as

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\ F(\boldsymbol{\theta}) \coloneqq \mathbb{E}_{\boldsymbol{x}_0 \sim \mathbb{R}^n} \mathbb{E}_{f \sim \mathcal{T}} \left(\sum_{k=1}^{K} w_k f(\boldsymbol{x}_k) \right) \tag{6}$$

$$\text{s.t.}\ \ \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \pi(\boldsymbol{d}_k, \boldsymbol{g}_k; \boldsymbol{\theta}) \boldsymbol{d}_k \tag{7}$$

where $\boldsymbol{d}_k$ is given by Algorithm 1, $K \in \mathbb{N}$ is the truncated horizon over which optimization steps are unrolled, $w_k,\ k = 1, \ldots, K$ are weight scalars, herein considered $w_k = 1$, and $\mathcal{T}$ is some set of tasks formed by inner objective functions $f(\boldsymbol{x})$ to be optimized. In (6), the innermost expected value is approximated by sampling tasks within a training set $\mathcal{T}_{train}$, one at a time, and unrolling the optimization for K inner steps for some random $\boldsymbol{x}_0$ with i.i.d. components. One outer optimization step consists of, performing K inner steps, computing a gradient for the outer optimization problem $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ and updating $\boldsymbol{\theta}$, in our case, by ADADELTA with learning rate equals 1 (Zeiler, 2012). To assure that different orders of magnitude of $\boldsymbol{x}$ are seen during this training, we set the initial point for the next outer step to be the last iterate from the previous one, i.e., $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}_K$, and perform another whole outer optimization step. This is repeated for T outer steps or until $\|\boldsymbol{g}_k\| < \epsilon = 10^{-10}$, when a new random $\boldsymbol{x}_0$ is then sampled. Backpropagation to calculate $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ happens through all operations with exception of the inner gradient evaluation $\boldsymbol{g}_k$, which is considered an external input. Double floating-point precision is used to assure accurate results in the comparisons.

6 EXAMPLE: TRAINING A CLASSIFIER ON MNIST

All tests were carried out with the aid of `PyTorch` (Paszke et al., 2019) to backpropagate gradients and of `cvxpylayers` (Agrawal et al., 2019) to implement the CVX layers. They were run on an Intel Xeon E5-2690v4 equipped with an NVidia K80. First, we carried out tests on convex optimization problems, namely quadratic functions and logistic regression problems, but no significant difference was noticed and these were, therefore, omitted.

In this example, we trained an MLP with $n_l = 1$ hidden layer with $n_u = 20$ units and sigmoid activation functions to classify images of digits in MNIST database (LeCun et al., 1998). We used a full-batch gradient at every iteration, even though stochastic optimization is generally the most common strategy employed in similar tasks. However, our main interest in this example is to analyze the optimization problem itself and how our deterministic algorithm performs.

The $n = 16{,}280$ parameters defining an MLP are concatenated in $\boldsymbol{x}$ and $f(\boldsymbol{x})$ is the associated cross-entropy loss function for a given set of images. This is known to be a non-convex optimization problem, mainly because of the presence of non-linear activation functions. A training set of tasks $\mathcal{T}_{train}$ was constructed by randomly grouping images in MNIST training set into 60 batches of $N = 1{,}000$ images. For each of these batches one initial condition $\boldsymbol{x}_0$ was sampled, which altogether compose $\mathcal{T}_{train}$ with 60 tasks. The policy $\pi(\cdot; \theta)$ was trained for 10 epochs, $K = 50$ and $T = 16$, and its performance was compared to other methods, namely, L-BFGS with a BTLS, ADAM and RMSProp. Some remarks on long-training effects are discussed in Appendix B. For running Algorithm 1, we selected $m = 5$. The learning rates of ADAM and RMSProp were heuristically tuned to yield fast convergence by exhaustive search within the set $\left\{ i \times 10^j \ :\ i \in \{1, 3\},\ j \in \{-3, \ldots, -1\} \right\}$ and the values 0.03 and 0.01 were used, respectively. The BTLS parameters c_1 and c_2 were searched in the set $\{0.25, 0.5, 0.75\}^2$ and $c_1 = 0.25,\ c_2 = 0.5$ were chosen, associated to the best results. The initial step size for the BTLS was $t_k = 1$. The following comparisons were performed in a test set of tasks $\mathcal{T}_{test}$ built similarly to $\mathcal{T}_{train}$ but considering all images in the MNIST test set split into 10 batches of $N = 1{,}000$ images, and 100 random samples of $\boldsymbol{x}_0$ were generated for each batch, resulting in $1{,}000$ tasks.

The objective function value for three selected tasks is shown in the upper plots of Fig. 3 along with the correspondent selected step sizes by $\pi(\cdot; \boldsymbol{\theta})$ and the BTLS, on the bottom ones. For Task 1, L-BFGS-π was successful in attaining lower values for $f(\boldsymbol{x})$ when compared to the other algorithms. The spikes represent steps at which $\pi(\cdot; \boldsymbol{\theta})$ took a step size that did not provided a decrease. For

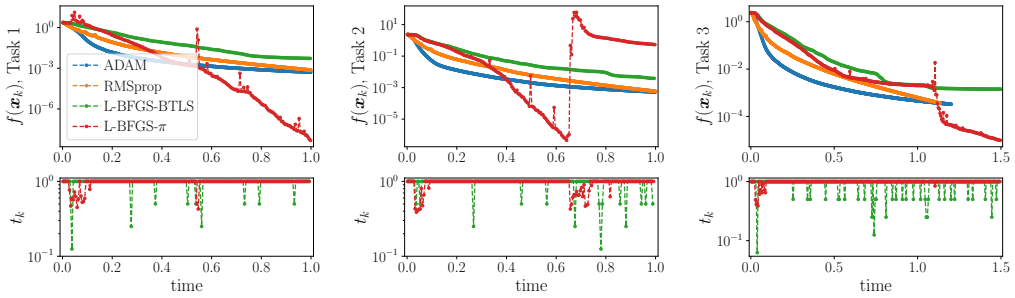

Figure 3: Objective function $f(\boldsymbol{x}_k)$ at the current iterate with respect to wall-clock time for all optimizers and 3 selected tasks (top) and correspondent step sizes for $\pi(\cdot, \boldsymbol{\theta})$ and BTLS (bottom).

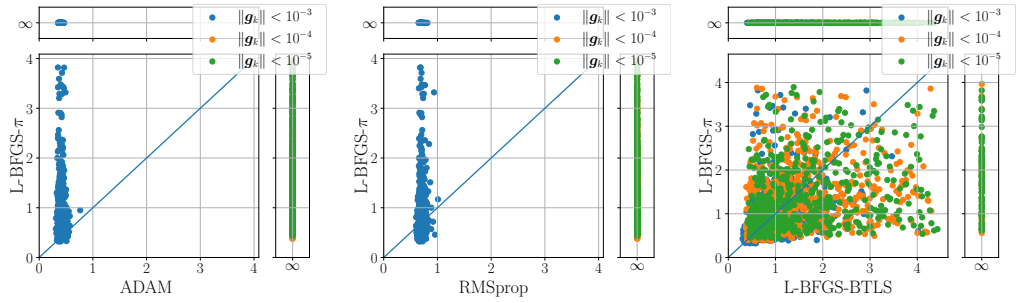

Figure 4: Time t_f (in seconds) at when the step criterion $\|\boldsymbol{g}_k\| < \varepsilon$ was reached for all tasks in $\mathcal{T}_{test}$ compared for different optimizers and $(n_l, n_u) = (1, 20)$.

some tasks, such as Task 2, this behavior led to a poor performance, leading the iterate to some local minimum that is not adequate. However, this same phenomenon was useful in Task 3 to help the optimizer escape from a region of slow convergence.

For each individual task, the first instant of time t_f at which the optimization procedure attained some precision-based stop criteria $\|\boldsymbol{g}_k\| < \varepsilon$ for different values of ε was computed for all four optimization procedures, and a comparison between our methodology and others is shown in Fig. 4. These plots compare algorithms two-by-two and the way to interpret them is by observing that a point above the blue line represents a task for which the algorithm associated to the x axes reached the precision criterion faster, and *vice-versa*. The two subplots, on the right and on the top, are use to represent tasks for which the given precision was reached by one of the algorithms but not by the other. Tasks for which the criterion was not reached by both algorithms are not displayed. Notice that better precision values are reached by our approach when compared to ADAM and RMSProp but a similar performance was obtained when compared to the heuristically designed backtracking line search method, which is the gold standard.

Additionally, Table 1 presents the percentage of times that L-BFGS-π reached the defined precision before other methods, characterizing a "win", and that both methods reached the precision at the exact same time (which is very unlikely) or have not reached this precision after $K = 800$ inner steps, denoting a "tie". To investigate whether our policy is able to generalize and perform well on higher-dimension problems, we also present these results for (n_l, n_u) equals to $(2, 400)$ and $(4, 800)$, characterizing problems of size $n = 637,600$ and $n = 128,317,600$ respectively. Different values of ε were considered as smaller values for $\|\boldsymbol{g}_k\|$ were reached for these two last cases. RMSProp presented the poorest performance compared to L-BFGS-π while L-BFGS-BTLS and ADAM produced comparable results but, for $(n_l, n_u) = (4, 800)$, ADAM seems to attain the stop criterion much faster than L-BFGS-π. In spite of that, given the non-convexity of this problem, it is important to observe what were the minimum values obtained for $f(\boldsymbol{x})$ by each algorithm. As the proposed policy does not assure a decreasing step size at each iteration, instead of the final value $f(\boldsymbol{x}_k)$ we looked at $f_* := \min_k f(\boldsymbol{x}_k)$, which can easily be stored and updated during the optimiza-

Table 1: Percentages of wins (W) and ties (T) of L-BFGS-π against other algorithms on tasks in $\mathcal{T}_{test}$ with respect to which one attained the stop criterion $\|\boldsymbol{g}_k\| < \varepsilon$ first for different (n_l, n_u).

	$(n_l, n_u) = (1, 20)$			$(n_l, n_u) = (2, 400)$			$(n_l, n_u) = (4, 800)$		
Competitor	ε	W (%)	T (%)	ε	W (%)	T (%)	ε	W (%)	T (%)
ADAM	10^{-3}	4.6	0	10^{-5}	14.4	0.1	10^{-5}	0.3	0
	10^{-4}	83.8	16.2	10^{-6}	37.7	0.1	10^{-6}	0.3	0.1
	10^{-5}	66.6	33.4	10^{-7}	65.7	0.1	10^{-7}	0.7	0.1
RMSProp	10^{-3}	47.8	0	10^{-5}	97.4	2.6	10^{-5}	94.0	6.0
	10^{-4}	83.8	16.2	10^{-6}	97.4	2.6	10^{-6}	94.0	6.0
	10^{-5}	66.6	33.4	10^{-7}	97.4	2.6	10^{-7}	94.0	6.0
L-BFGS-BTLS	10^{-3}	54.1	0.1	10^{-5}	69.3	0	10^{-5}	70.4	0
	10^{-4}	49.0	2.1	10^{-6}	68.4	0	10^{-6}	69.5	0
	10^{-5}	41.3	7.9	10^{-7}	69.7	0	10^{-7}	68.4	0

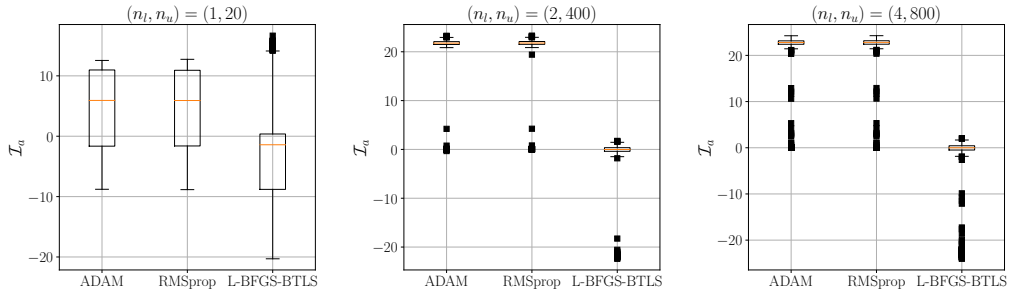

Figure 5: Box plots for values of $\mathcal{I}_a$ where a is the algorithm in the x-axis encountered for all tasks in $\mathcal{T}_{test}$ by different algorithms and pairs (n_l, n_u).

tion. More than simply looking at the minimum values, we would like to verify whether L-BFGS-π attains lower values f_* when compared to other algorithms. To this end we present the index

$$\mathcal{I}_a(f) = \ln\left(f_*^{[a]} / f_*^{[\text{L-BFGS-}\pi]}\right)$$

where $f_*^{[a]}$ represents the minimum value reached by some algorithm a for $f(\boldsymbol{x})$. Hence, $\mathcal{I}_a(f) > 0$ implies that L-BFGS-π performs better than a in the task associated to $f(\boldsymbol{x})$ and its initial condition. Box plots of the obtained values for all tasks and each one of the other algorithms are presented in Fig. 5. In these plots we can notice that ADAM and RMSprop generalized very poorly to higher-dimension problems, indicating that some re-tuning is required. Also, L-BFGS-π had a similar performance to L-BFGS-BTLS, despite the presence of some outliers indicating cases where our policy reached bad local minima. This showed how the proposed policy was successful in learning to provide step sizes in a single shot that are as good as those generated by a heuristically designed line search, which benefits from the possibility of re-evaluating the objective function as much as needed.

7 Conclusions

In this work we demonstrate how to build and train a neural network to work as step-size policy for the L-BFGS algorithm. The step sizes provided by this policy are of the same quality as those of a backtracking line searches, hence making the latter superfluous. Moreover, L-BFGS with our step-size policy outperforms, in wall-clock time, ADAM and RMSprop with heuristically tuned parameters in training classifiers for the MNIST database. When generalizing to higher-dimension problems, the performance improvement is even more remarkable. In future work, we intend to extend this result for stochastic optimization, allowing us to learn policies to determine, for example, learning rates in other classic machine learning algorithms.

REFERENCES

A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, pages 9562–9574, 2019.

M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv preprint arXiv:1606.04474*, 2016.

Y. Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

R. Bollapragada, D. Mudigere, J. Nocedal, H.-J. M. Shi, and P. T. P. Tang. A progressive batching L-BFGS method for machine learning. *arXiv preprint arXiv:1802.05374*, 2018.

C. Daniel, J. Taylor, and S. Nowozin. Learning step size controllers for robust neural network training. In *AAAI*, pages 1519–1525, 2016.

X. Dong, J. Shen, W. Wang, Y. Liu, L. Shao, and F. Porikli. Hyperparameter optimization for tracking with continuous deep Q-learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 518–527, 2018.

S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

K. Li and J. Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

L. Metz, N. Maheswaranathan, J. Nixon, D. Freeman, and J. Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565, 2019.

P. Moritz, R. Nishihara, and M. Jordan. A linearly-convergent stochastic L-BFGS algorithm. In *Artificial Intelligence and Statistics*, pages 249–258, 2016.

J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. 2019.

J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams. Scalable bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pages 2171–2180, 2015.

S. Thrun and L. Pratt. *Learning to learn*. Springer Science & Business Media, 1998.

A. Wills and T. Schön. Stochastic quasi-newton with adaptive step lengths for large-scale problems. *arXiv preprint arXiv:1802.04310*, 2018.

Z. Xu, A. M. Dai, J. Kemp, and L. Metz. Learning an adaptive learning rate schedule. *arXiv preprint arXiv:1909.09712*, 2019.

M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

C. Zhou, W. Gao, and D. Goldfarb. Stochastic adaptive quasi-newton methods for minimizing expected values. In *International Conference on Machine Learning*, pages 4150–4159, 2017.

A PARAMETERS FOR THE POLICY TO CALCULATE $\cos\phi_k$

In Section 4, it was stated that the policy $\pi(\cdot;\boldsymbol{\theta})$ is able to calculate

$$\cos\phi_k = \frac{-\boldsymbol{d}_k^T\boldsymbol{g}_k}{\sqrt{\boldsymbol{d}_k^T\boldsymbol{d}_k\boldsymbol{g}_k^T\boldsymbol{g}_k}}. \tag{8}$$

In the simplest case, this can be done by letting

$$\boldsymbol{W}_{01} = \begin{bmatrix} -0.5 & -0.5 & 1 \\ & \mathbf{0} & \end{bmatrix}, \tag{9}$$

$\boldsymbol{b}_{01} = \mathbf{0}$, $\boldsymbol{W}_{02} = \mathbf{0}$ and $\boldsymbol{b}_{02} = [1\ 0\ \cdots\ 0]^T$, where $\mathbf{0}$ denotes a zero matrix or vector with appropriate dimensions. The optimization problem becomes

$$\tau_k = \arg\min_{\tau\in\mathbb{R}} \Big(\tau - \ln\big(\cos\phi_k\big)\Big)^2 \tag{10}$$

$$\text{s.t. } \tau_m \leq \tau \leq \tau_M \tag{11}$$

Recalling that $\cos\phi_k > 0$ as $-\boldsymbol{d}_k^T\boldsymbol{g}_k > 0$, letting $\tau_M = 0$ and τ_m small enough assures that $t_k = e^{t_k} = \cos\phi_k$. It is important to say that this specific step size might not be a good one, but this quantity can carry useful information when composing vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ as it characterizes the deviation between the update direction $\boldsymbol{d}_k$ and the steepest descend direction.

B FURTHER TRAINING OF THE POLICY

Table 2: Percentages of wins (W) and ties (T) of L-BFGS-π (trained until epoch 50) against other algorithms on tasks in $\mathcal{T}_{test}$ with respect to which one attained the stop criterion $\|\boldsymbol{g}_k\| < \varepsilon$ first for different (n_l, n_u).

	$(n_l,n_u)=(1,20)$			$(n_l,n_u)=(2,400)$			$(n_l,n_u)=(4,800)$		
Competitor	ε	W (%)	T (%)	ε	W (%)	T (%)	ε	W (%)	T (%)
ADAM	10^{-3}	5.4	0	10^{-5}	6.1	0.0	10^{-5}	0.3	0
	10^{-4}	91.7	8.3	10^{-6}	10.9	0.0	10^{-6}	0.1	0.1
	10^{-5}	79.0	21.0	10^{-7}	21.0	0.0	10^{-7}	0.4	0.1
RMSProp	10^{-3}	57.7	0	10^{-5}	99.6	0.4	10^{-5}	98.2	1.8
	10^{-4}	91.7	8.3	10^{-6}	99.6	0.4	10^{-6}	97.8	2.2
	10^{-5}	79.0	21.0	10^{-7}	99.6	0.4	10^{-7}	97.7	2.3
L-BFGS-BTLS	10^{-3}	64.7	0	10^{-5}	14.6	0	10^{-5}	2.5	0
	10^{-4}	59.1	1.4	10^{-6}	15.4	0	10^{-6}	2.4	0
	10^{-5}	51.7	6.0	10^{-7}	16.2	0	10^{-7}	2.8	0

Here we investigate the effects of training the policy for more epochs. At the end of epoch 50, the model was again evaluated in the test set $\mathcal{T}_{train}$ for the same values of (n_l, n_u) used in Sec. 6. The updated values from Table 1 and Fig. 5 are displayed in Table 2 and Fig. 6, respectively. As expected, we can observe that our policy performed better than before for the values $(n_l, n_u) = (1, 20)$, which were used for training. Indeed, the box plots in Fig. 6 indicate that L-BFGS-π could find even better minimum values f_* than ADAM and RMSprop whereas it performed again similarly to the heuristically designed backtracking line search of L-BFGS-BTLS. Regarding its ability to generalize, it can still perform well in the higher-order problems, specially when compared to ADAM and RMSprop in Fig. 6, although Table 2 shows that it attained the stopping criteria after L-BFGS-BTLS in most cases. This indicates that our policy may risk overfitting to the training data and common techniques to avoid it, such as dropouts and regularization, should be considered in future work.

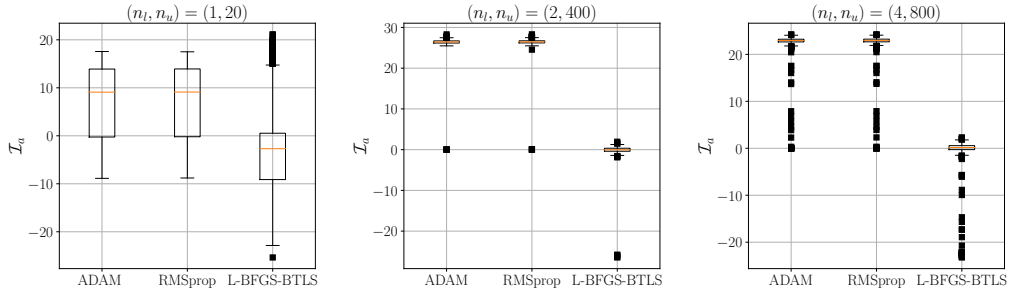

Figure 6: Box plots for values of $\mathcal{I}_a$ where a is the algorithm in the x-axis encountered for all tasks in $\mathcal{T}_{test}$ by different algorithms and pairs (n_l, n_u) when $\pi(\cdot, \theta)$ is trained until epoch 50.

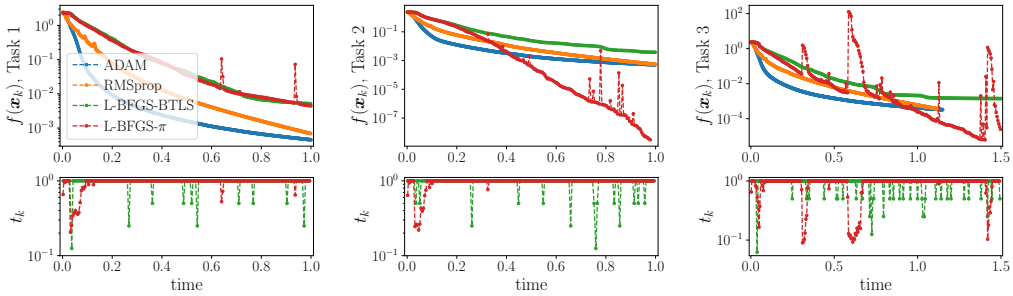

Figure 7: Objective function value $f(\boldsymbol{x}_k)$ for the current iterate with respect to wall-clock time for all optimizers and 3 selected tasks on the top and correspondent chosen step size for L-BFGS-π and L-BFGS-BTLS, on the bottom ($\pi(\cdot, \theta)$ was trained until epoch 50).

Finally, in Fig. 7 we present the updated values of $f(\boldsymbol{x}_k)$ for the three tasks highlighted in Fig. 3. Although the occurrence of the spike-like phenomenon could not be totally avoided by further training the policy, the frequency of tasks having at least one increase in the objective value within the first 0.1 seconds went down from 60.2 % (for epoch 10) to 32.0 % (for epoch 50). One example is Task 1 that, as it can be seen in Fig. 7, does not have a peak in the early iterations as it had in Fig. 3.