Applications of Level Bundle method and L-BFGS for Neural Network training

Bersani Michele [*], *Lusiani Federico* [†], *Marincioni Alessandro*[‡]

Computational Mathematics for AI and Data Analysis, Academic Year: 2020/2021

8 April 2021

1 Abstract

The fitting of a neural network model to a dataset is a common optimization problem in the field of Machine Learning. In this work, the problem is approached by the application of two optimization algorithms: Level Bundle method and L-BFGS. During the training on a regression task, they update the weights of their respective neural networks.

2 Introduction

2.1 Notation

Neural Network. We consider a feed-forward neural network with L layers, defined by weight matrices W_l (whose last columns are bias vectors b_l) and activation functions ϕ_l for $l \in \{1 \cdots L\}$. The neural network defines a function $M_\theta : X \to Y$ parametric in θ, with $\theta = \left[vec(W_1)^\top, \cdots, vec(W_L)^\top\right]^\top$.

Let the dataset be an array of pairs $(d_i, y_i) \in X \times Y$ for $i \in \{1 \cdots D\}$. Given a loss function $\mathcal{L} : X \times Y \to \mathbb{R}$, we define the objective function $f(\theta)$ as:

$$f(\theta) = \sum_{i=1}^{D} \mathcal{L}(M_\theta(d_i), y_i) = \sum_{i=1}^{D} f(i; \theta) \tag{1}$$

where $f(i; \theta)$ is defined as the loss for the single data-point with index i. The gradient of $f(\theta)$ is given by:

$$\nabla f(\theta) = \sum_{i=1}^{D} \nabla_\theta \mathcal{L}(M_\theta(d_i), y_i) = \sum_{i=1}^{D} \nabla f(i; \theta) \tag{2}$$

[*]Università di Pisa, Master Degree in Computer Science, m.bersani@studenti.unipi.it

[†]Università di Pisa, Master Degree in Computer Science, f.lusiani@studenti.unipi.it

[‡]Università di Pisa, Master Degree in Computer Science, a.marincioni@studenti.unipi.it

Given the data-point of index i, we compute $M_\theta(d_i)$ with a *forward pass* of the network, and then $\nabla f(i;\theta)$ with a *backward pass* (or *backpropagation*), as described in Algorithm 1.

Regularization. Regularization is the introduction of an additional loss term $\mathcal{L}_{reg}(\theta)$, so that $f_{reg}(\theta) = f(\theta) + \mathcal{L}_{reg}(\theta)$ and $\nabla f_{reg}(\theta) = \nabla f(\theta) + \nabla L_{reg}(\theta)$. From now on we will simply refer to $f_{reg}(\theta)$ as $f(\theta)$.

Optimization Algorithms. Training a neural model reduces to finding (a good approximation of) a point θ^* that is a minimum (either locally or globally) for $f(\theta)$. In this work we present two algorithms to perform this search. When describing the algorithms, we refer to θ as $\mathbf{x}$ and $f(\theta)$ as $f(\mathbf{x})$. Therefore, note that when applying the algorithm to the neural model, $\mathbf{x}$ is the vector θ of the weights and the biases.
Since $f(\theta)$ is the loss of the neural model on the whole dataset (Equation 8), minimizing $f(\mathbf{x})$ is a form of batch-training.

Model implementations. If there are points where the loss function $\mathcal{L}$ or the activation functions ϕ_l are not differentiable, the same will hold for $f(\theta)$. This is the case for our first Model, M1. Specifically, M1 includes:

- the $ReLU$ as activation function,
- squared error with L1 regularization as loss function.

Conversely, if all the functions $\mathcal{L}$ and ϕ_l are differentiable in every point, we have that $f(\theta)$ is differentiable at every point. This is the case for our second Model, M2. Specifically, M2 includes:

- a *sigmoid* activation function,
- squared error with L2 regularization as loss function.

Gradient and Sub-gradient. The forward and backward pass of the model (see Algorithm 1) requires the derivatives $\frac{\partial \mathcal{L}(\mathbf{out}_L, y)}{\partial \mathbf{out}_L}$ and $\phi_l'(\mathbf{h}_l)$ to be computed. In the case of M1, these are actually *sub-derivatives* (even though we use the same notation). Computing sub-derivatives in the backward-pass yields a final $\nabla f(\theta)$ that is actually a *sub-gradient* of f in θ.

3 Algorithms

In Section 3.1 we present the forward and backward pass of the neural model, which provides a mean to compute the gradient (or sub-gradient) $\nabla f(\theta)$ for a given θ. This quantity is used by the two optimization algorithms presented in this work:

- Level Bundle method (Section 3.2), applied to M1
- L-BFGS method (Section 3.3), applied to M2

3.1 Back-propagation

Given an input x, the forward pass of the model is used to compute the output $\mathbf{out}_L$ of the neural network.
Considering the corresponding label y, the backward pass computes the gradient of the loss function $\mathcal{L}$ w.r.t. $\mathbf{out}_L$. From that, it computes the gradients $\mathcal{D}W_l$ for $l \in [L \cdots 1]$, which concatenated form $\nabla f(\theta)$ (as defined before, θ is the concatenation of all the weights).

Algorithm 1 Forward and backward pass of a NN for a single data point

Given input (d, y), weights (and biases) $\mathbf{W_l}$ and activations ϕ_l for $l \in [1, L]$
Forward pass:
$\mathbf{out}_0 = d$
for all $l = 1, \cdots, L$ **do**
    $\mathbf{out}_{l-1} = (\mathbf{out}_{l-1}, 1)$\{ Append 1 to vector, needed for the bias\}
    $\mathbf{h}_l = \mathbf{W}_l \mathbf{out}_{l-1}$
    $\mathbf{out}_l = \phi_l(\mathbf{h}_l)$
end for
Backward pass:
$\mathcal{D}\mathbf{out}_L \leftarrow \frac{\partial \mathcal{L}(z, y)}{\partial z}|_{z=\mathbf{out}_L}$
for all $l = L, \cdots, 1$ **do**
    $\mathbf{g}_l = \mathcal{D}\mathbf{out}_l \odot \phi_l'(\mathbf{h}_l)$
    $\mathcal{D}\mathbf{W}_l = \mathbf{g}_l \mathbf{out}_{l-1}^\top$
    $\mathcal{D}\mathbf{out}_{l-1} = \mathbf{W_l}^\top \mathbf{g}_l$
end for

3.2 Level Bundle method

This method operates on M1. The function to optimize is not differentiable due to the activation function and the presence of L1 regularization.

This method builds a cutting plane model to describe the function to optimize. At each step the model is updated using the value of the function and one of its subgradients. A target level is chosen, and the point $\mathbf{x_{i+1}}$ is computed to achieve such value on the model.

We implement the method described in [1], which is summarized in Algorithm 2.

At iteration i, $\mathbf{x_i^*}$ is the current minima, where the function value (i.e. the current minimum) is $f^*(i) = f(\mathbf{x_i^*})$. Note that at this point the model is defined by i hyperplanes, since we do not perform pruning of the bundle. The minimum of f on the current model is referred as $f_*(i)$.

The cycle stops when $\Delta(i) = f^*(i) - f_*(i)$ falls below a desired ϵ. If f is convex, ϵ is always positive; while during non-convex optimization, $\epsilon < 0$ establishes a failure and is eventually handled restarting the method and deleting the bundle or (through more complex approaches) pruning the bundle.

Algorithm 2 Level Bundle Method

1: Choose $\mathbf{x_1} \in Q$ where Q is a nonempty closed set, $\lambda \in (0, 1)$
2: **while** $\Delta(i) > \epsilon$ **do**
3: Compute $f(\mathbf{x_i})$, $g(\mathbf{x_i})$
4: Update $f^*(i)$ and $\mathbf{x_i^*}$
5: Update the model {See Equation 3}
6: Minimize the model (LP) to obtain $f_*(i)$
7: Compute $\Delta(i) = f^*(i) - f_*(i)$
8: Compute the target level $l(i) = f_*(i) + \lambda\Delta(i)$
9: Project $\mathbf{x_i}$ on the chosen level set of the model to obtain $\mathbf{x_{i+1}}$
10: **end while**

3.2.1 Minimizing the cutting plane model

The cutting plane model at iteration i is defined as:

$$f_\beta^i(\mathbf{x}) = \max\{f(\mathbf{x_j}) + g(\mathbf{x_j})^T(\mathbf{x} - \mathbf{x_j}) \;\mid\; 1 \leq j \leq i\} \tag{3}$$

where $f(\mathbf{x_j})$ and $g(\mathbf{x_j})$ are respectively the value of the function and a subgradient of the function computed at iteration j.

At step 6, the algorithm is required to find the minimum value of such model. This problem can be formulated as a linear program:

$$\begin{aligned} \text{find } (\mathbf{x}, y) \text{ which minimize} \quad & y \\ \text{subject to} \quad & g(\mathbf{x_1})^T\mathbf{x} - y \leq g(\mathbf{x_1})^T\mathbf{x_1} - f(\mathbf{x_1}) \\ & \vdots \\ & g(\mathbf{x_i})^T\mathbf{x} - y \leq g(\mathbf{x_i})^T\mathbf{x_i} - f(\mathbf{x_i}) \\ & \mathbf{x} \in Q \end{aligned}$$

The constraints have been rewritten to fit the standard LP representation. They are equivalent to considering only the points in the epigraph of the cutting plane model

(Equation 4). We want to find the smallest y in this set, which will always be a point on the model itself.

$$y \geq f(\mathbf{x_j}) + g(\mathbf{x_j})^T(\mathbf{x} - \mathbf{x_j}) \qquad \text{for } j = 1, 2, \ldots, i \tag{4}$$

3.2.2 Projecting on the level set

At iteration i, the level set of the model is defined as $l(i) = f_*(i) + \lambda\Delta(i)$. The update rule of the algorithm requires us to find the projection of a point $\mathbf{x_i}$ on $l(i)$, within the boundaries of the closed convex set Q.

$$\mathbf{x_{i+1}} = \pi(\mathbf{x_i}, \{\mathbf{x} \mid \mathbf{x} \in Q, f_i(\mathbf{x}) \leq l(i)\}). \tag{5}$$

With an appropriate choice of Q, this problem can be formulated as a quadratic program:

$$\begin{aligned} \text{minimize} \quad & \|\mathbf{x} - \mathbf{x_i}\|^2 \\ \text{subject to} \quad & \mathbf{x} \in Q \\ & g(\mathbf{x_1})^T\mathbf{x} \leq g(\mathbf{x_1})^T\mathbf{x_1} - f(\mathbf{x_1}) + l(1) \\ & \vdots \\ & g(\mathbf{x_i})^T\mathbf{x} \leq g(\mathbf{x_i})^T\mathbf{x_i} - f(\mathbf{x_i}) + l(i) \end{aligned}$$

Minimizing the square norm of the difference of the points is equivalent to minimizing the following sum:

$$\|\mathbf{x} - \mathbf{x_i}\|_2^2 = \mathbf{x^T x} - 2\mathbf{x^T x_i} + \mathbf{x_i^T x_i} \tag{6}$$

Therefore, calling n the dimension of $\mathbf{x}$, defining the corresponding n by n matrix and n dimensional vector of the quadratic function that we are minimizing is trivial.

Depending on the choice of Q, it may or may not be possible to express the problem as a QP. Choosing any translated hyperrectangle as Q ensures the problem to be easily expressed as a set of inequalities on $\mathbf{x}$.

3.3 L-BFGS

This algorithm operates on M2, thus the objective function is continuous and differentiable in all its domain.
At each iteration, it takes a step $s = -\alpha B\nabla f$, where α is the stepsize and B is the inverse of the Hessian at the current point. Being the number of weights n considerable, computing B at every iteration would be expensive. Quasi-Newton methods are exploited so that it is only estimated through a recursive process. The stepsize α is finally determined by Armijo-Wolfe Line Search (see Section 3.4).

BFGS is the chosen implementation of Quasi-Newton strategy. In its traditional implementation, storing B requires $O(n^2)$ in space, but we exploit a limited-memory BFGS to achieve space complexity $O(n)$ and time complexity of $O(Mn)$ per iteration, where M is the memory parameter. This is obtained by replacing B with a series of products and sums between vectors of size n.

The modified L-BFGS. Nocedal et al. [6] prove the convergence of the BFGS algorithm, under a set of assumptions on f. One of them requires the level set $\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n \; s.t. \; f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$ to be convex, and the hessian H of f to be positive definite in $\mathcal{L}$: both not guaranteed in our case.
For this reason, we have chosen to implement the modified L-BFGS algorithm proposed by Xiao et al. [3], which introduces the two parameters C and μ, and guarantees global convergence under a smaller set of assumptions (see Section 4.2 for further details). Instead of using $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ to estimate B, this modified version uses $y_k^* = y_k + t_k s_k$, where $t_k = C\|\nabla f(x_k)\|^\mu$ (line 10 of Algorithm 3).
Notice that, compared to the formula given by Xiao et al., t_k does not present the term $max\{-\mathbf{s}^T\mathbf{y}/\|\mathbf{s}\|^2, 0\}$. This is because the Wolfe condition imposed in the line-search ensures $\mathbf{s}^T\mathbf{y} > 0$, fixing this term to zero.

Implementation details. In Algorithm 3, we implement the first L-BFGS method, proposed in 1980 by Nocedal [6], applying the modification proposed in [3] (line 10 of the algorithm).

A central operation of the process, described in Algorithm 4, is the function *Two-Loop-Recursion*, which computes the inner product $B\nabla f(\mathbf{x})$, where B is the estimation of H^{-1} at $\mathbf{x}$. Computing the exact equivalent of $B\nabla f(\mathbf{x})$ would require the complete history of s, y and ρ; because of memory constraints, we only keep memory of the last M iterations, thus computing an approximation.

3.4 Armijo-Wolfe Line Search

Once the direction $\mathbf{d}$ has been determined, a crucial part of the algorithm is finding a suitable step-size, computing as few function (and gradient) evaluations as possible. Armijo-Wolfe is selected as Line Search method for this application:

- Armijo condition (A) ensures that the succession of points $\mathbf{x}_i$ is contained into $\mathcal{L}$, the level set below $f(\mathbf{x}_0)$ (as reported in section 4.2 this is required for the convergence of the algorithm).

- Wolfe condition (W) guarantees that at each iteration i, $\mathbf{y}_i^\top \mathbf{s}_i > 0$; this product is the inverse of *rho* (see line 11 of Algorithm 3) and must be strictly positive for the algorithm to work properly (if negative, $\mathbf{d}$ is not always guaranteed to be a direction of descent, and the line-search might fail).

Algorithm 3 Modified L-BFGS Method

Given a starting point $\mathbf{x}_0$, the stopping threshold ϵ, the memory m, parameters $C > 0$ and $\mu \geq 0$
Initialize s, y, rho as empty queues
$k \leftarrow 0$
while $\|\nabla f(\mathbf{x}_k)\| > \epsilon \|\nabla f(\mathbf{x}_0)\|$ **do**
  $\mathbf{r} \leftarrow$ Two-Loop-Recursion$(\nabla f(\mathbf{x}_k))$
  Determine α_k with Armijo-Wolfe Line Search
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha_k \mathbf{r}$
  $\mathbf{s}_{new} \leftarrow \mathbf{x}_{k+1} - \mathbf{x}_k$
  $\mathbf{y}_{new} \leftarrow \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$
  $\mathbf{y}^*_{new} \leftarrow \mathbf{y}_{new} + C\|\nabla f(\mathbf{x}_k)\|^{\mu} \mathbf{s}_{new}$
  s.push($\mathbf{s}_{new}$), y.push($\mathbf{y}^*_{new}$), rho.push$\left(\frac{1}{\mathbf{s}_{new}{}^{\top} \mathbf{y}^*_{new}}\right)$
  if $k > m$ **then**
    s.pop(), y.pop(), rho.pop()
  end if
  $k \leftarrow k + 1$
end while
return $\mathbf{x}_k$ as optimum

Algorithm 4 Two Loop recursion

Given $\mathbf{q}$ as input vector and $B_0 \succ 0$
Given access to s, y, rho (each one with memory m)
for $i = m-1, m-2, \cdots, 0$ **do**
  $\eta_i \leftarrow \rho_i \mathbf{s}_i^{\top} \mathbf{q}$
  $\mathbf{q} \leftarrow \mathbf{q} - \eta_i \mathbf{y}_i$
end for
$\mathbf{r} \leftarrow B_0 \mathbf{q}$
for $i = 0, \cdots, m-2, m-1$ **do**
  $\beta \leftarrow \rho_i \mathbf{y}_i^{\top} \mathbf{r}$
  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{s}_i(\eta_i - \beta)$
end for

At iteration i, after computing the direction $\mathbf{d}$, we perform a line search on the function $\phi(\alpha) = f(\mathbf{x}_i + \alpha \mathbf{d})$, looking for a step-size $\alpha > 0$ that satisfies the Armijo and Wolfe conditions (AW). For (W), we actually require the Strong Wolfe condition.
The pseudo-code of the Line Search is described in Algorithms 5 and 6 (respectively Phase 1 and Phase 2). See [6] for a detailed explanation of how the algorithm works. We now provide a short description.

In the first phase, α_0 is initialized at 1 and then incremented (doubled) exponentially at each iteration ($\alpha_j = 1, 2, 4, 8, \dots$), until either (AW) is satisfied, or α_j provides an upper bound α^+ for the search. An upper bound α^+ is such that $\exists\, \alpha \in (0, \alpha^+)$ satisfying (AW).
An α_j provides an upper bound if at least one of the following conditions are met:

1. α_j violates the Armijo condition **2.** $\phi(\alpha_j) \geq \phi(\alpha_{j-1})$ **3.** $\phi'(\alpha_j) > 0$

Moreover, if α_j provides an upper bound α^+, then α_{j-1} provides a lower bound α^-, such that $\exists \alpha \in (\alpha^-, \alpha^+)$ satisfying (AW). If α^+ is α_0, α^- is set to 0.
At this point begins the second phase, where we apply a bisection method to the interval (α^-, α^+), until we find an α that satisfies AW (or the maximum number of f evaluations is reached).

Algorithm 5 AW line-search (Phase 1)

Given starting point $\mathbf{x}$ and direction $\mathbf{d}$
Given parameters m_1 and m_2
$\alpha_0 \leftarrow 1$
$j \leftarrow 0$
while max. f evaluations not reached **do**
  $\phi(\alpha_j) \leftarrow f(\mathbf{x} + \alpha_j \mathbf{d})$
  $\phi'(\alpha_j) \leftarrow \mathbf{d}^\top \nabla f(\mathbf{x} + \alpha_j \mathbf{d})$
  $A \leftarrow \phi(\alpha_j) \leq \phi(0) + m_1 \alpha_j \phi'(0)$
  $W \leftarrow |\phi'(\alpha_j)| \leq m_2 |\phi'(0)|$
  if A and W **then**
    return α_j
  else if $\neg A$ or $\phi(\alpha_j) \geq \phi(\alpha_{j-1})$ **then**
    return **zoom**(α_{j-1}, α_j)
  else if $\phi'(\alpha_j) \geq 0$ **then**
    return **zoom**(α_j, α_{j-1})
  end if
  $\alpha_j \leftarrow 2\alpha_j$
  $j \leftarrow j + 1$
end while

4 Expected performances

4.1 Level bundle method

The manual of Lemaréchal et al. [1] provides an upper bound to the number of iterations M needed to obtain a certain gap $\Delta = f^* - f_*$ on a convex function.

Algorithm 6 AW zoom function (Phase 2)

Given arguments $(\alpha_{lo}, \alpha_{hi})$ and access to line-search arguments
while max. f evaluations not reached **do**
  $\alpha_m \leftarrow (\alpha_{lo} + \alpha_{hi})/2$
  $\phi(\alpha_m) \leftarrow f(\mathbf{x} + \alpha_m \mathbf{d})$
  $\phi'(\alpha_m) \leftarrow \mathbf{d}^\top \nabla f(\mathbf{x} + \alpha_m \mathbf{d})$
  $A \leftarrow \phi(\alpha_m) \leq \phi(0) + m_1 \alpha_m \phi'(0)$
  $W \leftarrow |\phi'(\alpha_m)| \leq m_2 |\phi'(0)|$
  if A and W **then**
    return α_m
  else if $\neg A$ or $\phi(\alpha_m) \geq \phi(\alpha_{lo})$ **then**
    $\alpha_{hi} \leftarrow \alpha_m$
  else
    if $\phi(\alpha_m)(\alpha_{hi} - \alpha_{lo}) \geq 0$ **then**
      $\alpha_{hi} \leftarrow \alpha_{low}$
    end if
    $\alpha_{lo} \leftarrow \alpha_m$
  end if
end while

Defining L as the Lipschitz constant on f and q the side of the hypercube Q, it is proven that:

$$M(\Delta) \leq c \left(\frac{Lq\sqrt{n}}{\epsilon} \right)^2 \tag{7}$$

where c is a constant depending only on λ. Note that $q\sqrt{n}$ is the diameter of the smallest ball containing the n-dimensional hypercube Q.

Since the loss function on M1 is clearly non convex, the upper bound to M and therefore the convergence of the method are not guaranteed by theory.

4.2 L-BFGS

Xiao et al. [3] prove that global convergence ($\lim_{x_k \to \infty} \|\nabla f(x_k)\| < \epsilon$ for any starting x_0) is guaranteed for the modified L-BFGS algorithm, under the following set of assumptions:

(1) The level set $\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n \; s.t. \; f(\mathbf{x}) < f(\mathbf{x}_0)\}$ is bounded.

(2) f is twice continuously differentiable $\forall \mathbf{x} \in \mathcal{L}$;

(3) ∇f is Lipschitz continuous $\forall \mathbf{x} \in \mathcal{L}$.

In our experiments, these assumptions can be proven.

Proof of (1)

The formulation of the loss function ensures that $f(\mathbf{x}) > 0 \; \forall \mathbf{x} \in \mathbb{R}^n$.

therefore, given λ the L2 regularization parameter and defining $M_1 = \sqrt{\frac{f(\mathbf{x}_0)}{\lambda}}$, we have $\|\mathbf{x}\| < M_1 \;\; \forall \mathbf{x} \in \mathcal{L} \Rightarrow \mathcal{L} \subset \mathcal{B} = \{\mathbf{x} \in \mathbb{R}^n \; s.t. \; \|\mathbf{x}\| < M_1\}$

As a subset of a limited set, $\mathcal{L}$ is limited as well.

Proof of (2)

With squared error loss and L2 regularization, the formulation of $f(\theta)$ is:

$$f(\theta) = \sum_{i=1}^{D} \|M_\theta(d_i) - y_i\|_2^2 + \lambda \|\theta\|_2^2 \tag{8}$$

M is a function infinitely differentiable in both of its inputs θ and d_i, since it is a composition of infinitely differentiable functions: sum, product, sigmoid and identity activation functions. The same can be said for the squared 2-norm function.

Therefore f is infinitely differentiable in θ, being the sum of two infinitely differentiable functions. This guarantees of course Assumption (2).

Proof of (3)

In this proof we will refer to ∇f as g. We have proven above that $\mathcal{L} \subset \mathcal{B} = \{\mathbf{x} \in \mathbb{R}^n \; s.t. \; \|\mathbf{x}\| < M_1\}$. Therefore, to prove that g is Lipschitz continuous in the level set $\mathcal{L}$, it is sufficient to prove that g is Lipschitz continuous on $\mathcal{B}$.
Since $f \in C^\infty$, $g \in C^\infty$ as well. The Lipschitz continuity of g in $\mathcal{B}$ requires that $\exists L > 0 \; s.t. \; \|g(\mathbf{x}_0) - g(\mathbf{x}_1)\| \leq L \|\mathbf{x}_0 - \mathbf{x}_1\| \quad \forall \mathbf{x}_0, \mathbf{x}_1 \in \mathcal{B}$.

Given any pair of $\mathbf{x}_0$ and $\mathbf{x}_1 \in \mathcal{B}$, let us define $\varphi(\alpha) : [0, 1] \to \mathbb{R}^n$ as follows:

$$\varphi(\alpha) = g(\mathbf{x}_0 + \alpha(\mathbf{x}_1 - \mathbf{x}_0)) \tag{9}$$

Since $\varphi \in C^\infty$, the *Mean Value Theorem* ensures that $\forall(\alpha_0, \alpha_1), \; \exists \alpha^* \in [\alpha_0, \alpha_1]$ s.t.

$$\frac{\|\varphi(\alpha_0) - \varphi(\alpha_1)\|}{\|\alpha_0 - \alpha_1\|} = \|\varphi'(\alpha^*)\|$$

If we select $\alpha_0 = 0$ and $\alpha_1 = 1$, this equation becomes

$$\|\varphi(0) - \varphi(1)\| = \|\varphi'(\alpha^*)\| \qquad \text{with } \alpha^* \in [0, 1] \tag{10}$$

Let us recall that, by the definition of φ, $\|\varphi(0) - \varphi(1)\| = \|g(\mathbf{x}_0) - g(\mathbf{x}_1)\|$. Moreover:

$$\begin{aligned}\varphi'(\alpha^*) &= \nabla g(\mathbf{x}_0 + \alpha^*(\mathbf{x}_1 - \mathbf{x}_0))(\mathbf{x}_1 - \mathbf{x}_0) && \text{by chain-rule} \\ &= \nabla g(\mathbf{x}^*)(\mathbf{x}_1 - \mathbf{x}_0) \\ &= \nabla^2 f(\mathbf{x}^*)(\mathbf{x}_1 - \mathbf{x}_0) \end{aligned} \tag{11}$$

Substituting in (10), we obtain

$$\|g(\mathbf{x}_1) - g(\mathbf{x}_0)\| = \left\|\nabla^2 f(\mathbf{x}^*)(\mathbf{x}_1 - \mathbf{x}_0)\right\|$$

By the triangular inequality and defining $k = \|\nabla^2 f(\mathbf{x}^*)\|$,

$$\|g(\mathbf{x}_0) - g(\mathbf{x}_1)\| \leq k \|\mathbf{x}_1 - \mathbf{x}_0\|$$

Since $\mathbf{x}^*$ is an interpolation of two points contained in $\mathcal{B}$, then $\mathbf{x}^* \in \mathcal{B}$, since $\mathcal{B}$ is convex. Moreover, since $\mathcal{B}$ is bounded, and $f \in C^\infty$, then k is bounded and the proof is complete.

Superlinear convergence. This modified L-BFGS algorithm is the limited-memory implementation of the modified BFGS algorithm proposed by Li and Fukushima in [5]. Li et al. prove that if the sequence of points $\{x_k\}$ given by the modified BFGS algorithm converges to a point x^*, such that $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*) \succcurlyeq 0$ (that is, x^* is a strict local minima), then the convergence rate is superlinear.
This property is not proven for the limited-memory implementation, and therefore we cannot guarantee a specific rate of convergence, even under the aforementioned conditions. Nevertheless, L-BFGS appears to be one of the best compromises between efficiency and convergence speed currently available in Machine Learning[2].

5 The experiments

M1 and M2, with the respective optimization algorithms, will be trained for a regression task on a dataset of 100 patterns, using respectively L1 and L2 regularization (both weighted with $\alpha = 10^{-4}$). Each pattern $(\mathbf{x}, \mathbf{y})$ consists of ten input scalars $(x_1, x_2, \ldots, x_{10})$ and two output scalars (y_1, y_2).

To explore several combinations of optimization hyperparameters, both algorithms are tested on the same network topology and same dataset. For every configuration, the reported results are the average over three different weights initializations.

A Stochastic Gradient Descent, driven by ADAM, is run on both M1 and M2 in order to set a comparative benchmark for the performances of Bundle Method and L-BFGS.

Convergence criteria. Bundle and L-BFGS use two different convergence criteria to decide when to stop (see respectively Section 3.2 and 3.3), both of which are tuned by a parameter ϵ.
When the experiments aim to tuning the hyper-parameters of the two algorithms, we use a greater (more "relaxed") value of ϵ ($\epsilon = 10^{-2}$ for Bundle and $\epsilon = 10^{-4}$ for L-BFGS), in order to obtain shorter runs and more feasible computing times. When comparing the two algorithms to ADAM, we use a smaller value ($\epsilon = 10^{-6}$ for Bundle and $\epsilon = 10^{-10}$ for L-BFGS).

5.1 Experiments on Bundle Method

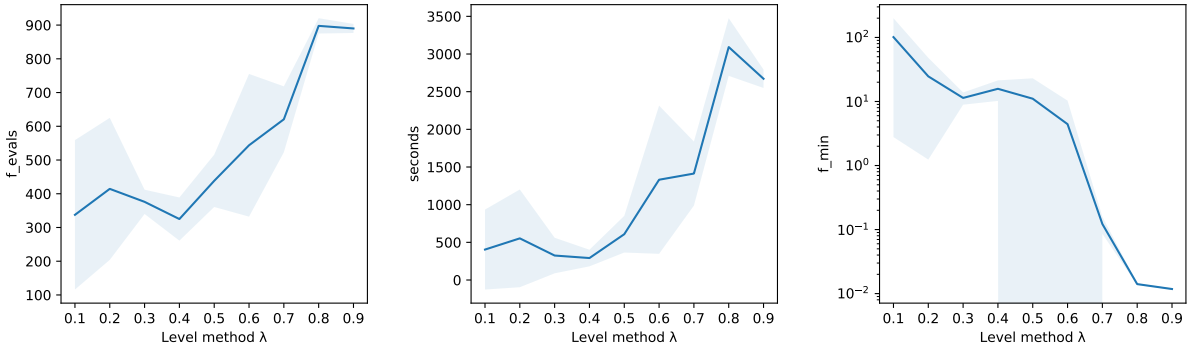
A first experimenting phase regards the solvers of LP and QP. Many state-of-art solvers are explored and MOSEK proves to be the fastest. The following experiments will use this solver.

5.1.1 Hyperparameters selection

A crucial hyperparameter for this algorithm is λ, which determines the distance of the level set $l(i)$ from the minimum of the bundle, $f_*(i)$. The first part of the experiment explores different values of λ, aiming at minimizing the iterations (and therefore the number of planes) needed to converge, given a certain ϵ. Once optimized on a (10,20,20,2) network, we assume λ to be a good value for all the architectures of the following experiment.

Performance on λ: The Level Bundle method is applied to minimize $f(x)$ using values of λ from 0.1 to 0.9. Figure 1 shows the mean and the standard deviation of the results for 3 different runs. From the plots we observe that greater values of λ (0.7 to 0.9) correspond to more function evaluations and longer *runtime*. However, greater values of λ also yield better minima of the target function: we have therefore a trade-off between the *runtime* of the algorithm and the quality of the minima that it can find.

It is also interesting to note that with λ between 0.1 and 0.4, the algorithm always fails one time out of three. The results shown in the plots ignore those runs and compute



(a) n. of f_{evals} (b) *runtime* (c) $min(f)$ achieved

Figure 1: Level method: Bundle performance as function of λ (3 runs)

mean and standard deviation on the successful runs. The reason of these failures is always the same: the value of the gap $\Delta(i)$ computed by the algorithm becomes negative, meaning that $f^*(i) < f_*(i)$ for the current iteration i. In the case of convex functions, the gap is guaranteed to be positive; but our target function is not. This means that the gap can eventually become negative, forcing the algorithm to stop and preventing us from converging to a minimum.

In the end, $\lambda = 0.7$ seems to be a good trade-off between the need to successfully reach a good minimum of the objective function and the number of algorithm steps required.

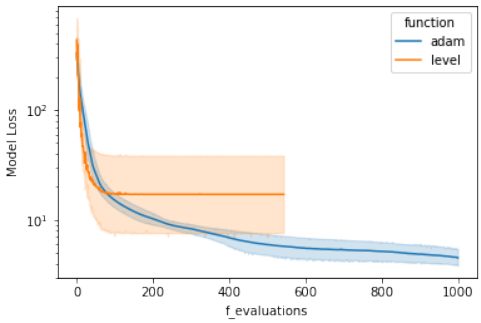
5.1.2 Bundle vs ADAM

Since every step of the Bundle method requires the machine to solve a LP and a QP, it is clear that the dimension of the problem (i.e. the number of the weights in the network) affects both the number of steps needed to converge and the time of each step. To analyze the dependence between the *runtime* and the dimension of the network, the second phase of the experiment explores the same algorithm with different numbers of nodes for the hidden layers:

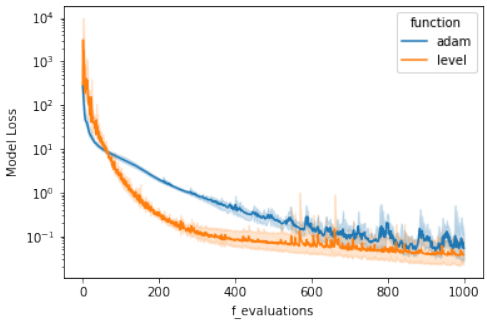
M1.a	**M1.b**	**M1.c**
10 5 5 2	10 20 20 2	10 50 50 2

These configurations are also leveraged to compare Bundle Method with Stochastic Gradient Descent, performed by Adaptive Moment Estimation (ADAM).

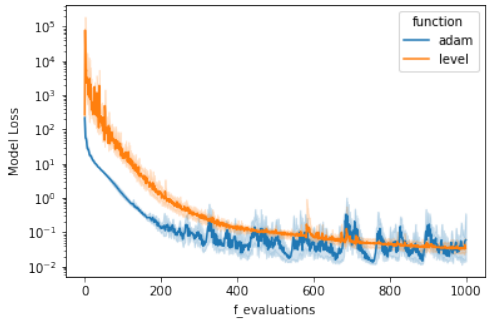
Figure 2 shows that, apart from M1.a, where the gap became negative and the algorithm stopped, the values of f found by Bundle method and ADAM are quite similar. Nevertheless, the running times reported in Table 1 prove that Level Bundle method is much slower than ADAM, about two orders of magnitude. Solving the LP and the QP



(a) M1.a



(b) M1.b



(c) M1.c

Figure 2: Bundle ($\lambda = 0.7$) vs ADAM: f value over f_{evals}(3 runs per plot)

at each iteration is probably the computational bottleneck and the solvers were selected between various state-of-art candidates. This is why ADAM seems outperforming Level Bundle method on M1.

A slight difference regards the smoothness of the curves: in the first phase of the method, Bundle is less regular than ADAM; whereas in the second part of the plot, maybe where the model is more refined, it gets more regular than ADAM.

Solver	M1.a (500 f_{evals})	M1.b (1000 f_{evals})	M1.c (1000 f_{evals})
Level Bundle	58.4s (±34s)	4338.7s (±407s)	13947.3 (±733)
ADAM	6.2s (±0.01s)	12.5s (±0.05s)	13.8s (±0.03s)

Table 1: Level Bundle vs ADAM: *runtime* comparisons on M1 (3 runs per cell)

5.2 Experiments on L-BFGS

5.2.1 Hyperparameters selection

In the first part of experiments on L-BFGS, we focus on testing the performance of the algorithm varying some hyperparameters, with the goal of selecting the best-performing values. During this tuning phase, ϵ is relaxed to 10^{-4}, obtaining shorter *runtimes* and hence more affordable runs.

Although a proper search would require an extensive grid-search on all the hyperparameters, for simplicity we first perform a grid-search on m_1 and m_2 keeping the memory M fixed, and then test the performance of the algorithm for different values of M. Moreover, during this hyperparameters tuning, the network topology of M2 is kept fixed to $[10, 20, 20, 2]$ neurons per layer.

We do not perform a grid search on hyper-parameters C and μ, selecting instead the default values proposed by Xiao et al. in [3] ($C = 10^{-7}$ and $\mu = 0$). Looking at their work, these configuration is reported as generally fine for all applications.

Grid-search on m_1 and m_2: Each L-BFGS step is driven by Armijo-Wolfe Line Search, with the two hyperparameters m_1 and m_2 (see Section 3.4). We perform a simple grid search to find their best combination, comparing f_{eval}: the number of evaluation of f and its gradient. The results are shown in Figure 3. Note that for $m_1 = 0.0001$ and $m_2 = 0.5$ no value is shown, since the algorithm failed on 2 of the 3 runs. In this case the failures were caused by unterminated Line Searches: given a certain direction, the algorithm did not find any point satisfying Armijo-Wolfe conditions.

In conclusion, $m_1 = 0.01$ and $m_2 = 0.7$ are selected as Armijo-Wolfe coefficients.

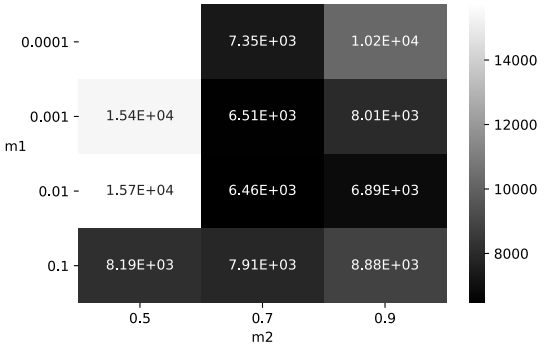


Figure 3: LBFGS: mean f_{evals} in function of $m1$, $m2$ (3 runs, $M = 10$)

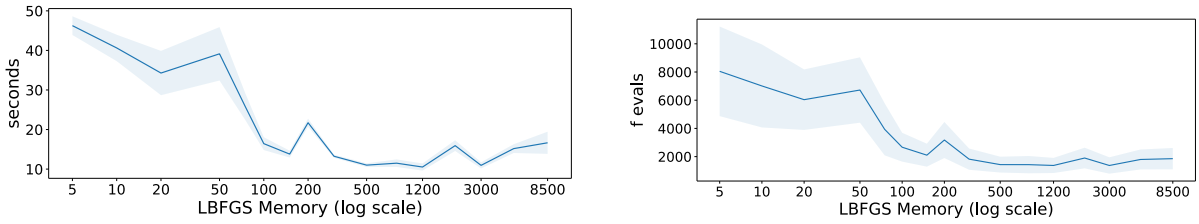
The impact of memory M on the performance: After tuning m_1 and m_2, the experiment focuses on M, the memory limit for L-BFGS. Higher M should generally

reduce f_{evals}, but should also slow down each evaluation by linearly increasing the number of vector summations and multiplications inside the Two Loop recursion. Therefore, aside of f_{evals}, also the *runtime* is evaluated to solve this trade-off. The general results are shown in Figure 4 and the top 4 performing values of M (both in terms of *runtime* and f_{evals}) are displayed in further detail in Table 2.

M	m1	m2	f_evals_m	f_evals_std	runtime_m	runtime_std
500	0.01	0.7	1433.3	562.5	11.0 s	0.468 s
800	0.01	0.7	1431.0	612.9	11.4 s	0.956 s
1200	0.01	0.7	1373.3	536.4	10.5 s	0.904 s
3000	0.01	0.7	1371.3	580.7	10.9 s	0.787 s

Table 2: LBFGS: top 4 performing values of M

As observable, in our case the time needed to compute f and its gradient seems to dominate over the two-loop recursion: even for large values of M, the *runtime* of the algorithm is essentially determined by the number of f_{evals}. At the end of the day, $M = 500$ seems to be the best compromise in terms of *runtime*, f evaluations and memory usage.



(a) *runtime* (b) n. of f_{evals}

Figure 4: LBFGS performance as function of M (3 runs, $m_1 = 0.01$, $m_2 = 0.7$)

Figure 5 reports the evolution of $\|\nabla f\|$ during a single run with $M = 500, m1 = 0.01, m2 = 0.7$.

5.2.2 L-BFGS vs ADAM

Stochastic Gradient Descent, performed by Adaptive Moment Estimation (ADAM), sets a benchmark for evaluating L-BFGS performance. As for Level Method, we test both the methods varying the dimension of the network, in particular the number of nodes for the hidden layers:

M2.a	M2.b	M2.c
10 5 5 2	10 20 20 2	10 50 50 2

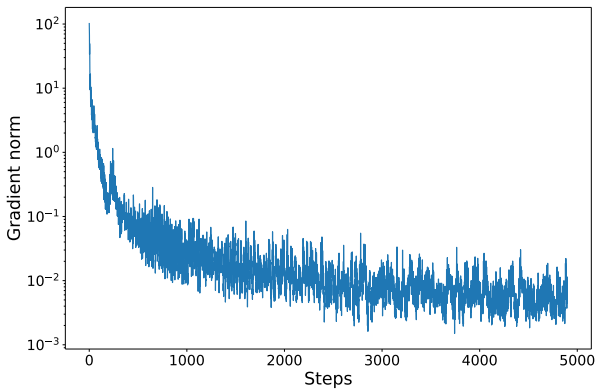


Figure 5: $\|\nabla f\|$ as function of LBFGS steps

ADAM is implemented with the standard hyperparameters for Machine Learning applications by Kingma and Ba [4] ($\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$).

L-BFGS is tested with the hyperparameters discussed in Section 5.2.1:

M	$m1$	$m2$	C	μ	ϵ
500	0.01	0.7	10^{-7}	0	10^{-10}

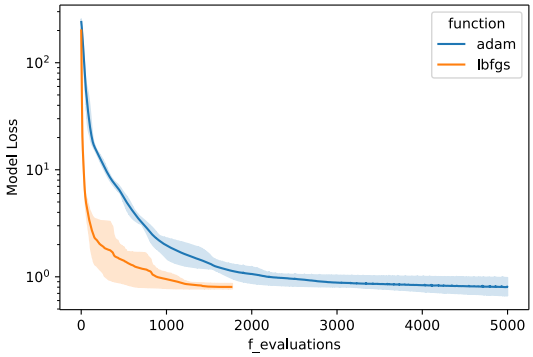
The plots in Figure 6 evidence a significant difference in the monotonicity of the curves: L-BFGS includes Armijo condition, which implies that $f(\mathbf{x_{i+1}}) < f(\mathbf{x_i}) \ \ \forall i$. On the other side, Stochastic Gradient Descent does not guarantee monotonicity, in fact its curves are visibly irregular. If wishing to remove irregularities, at every iteration one can consider f as the current $min(f)$ reached by the method; but this remains a meaningful difference between the two methods.

While minimizing f on M2.a, L-BFGS converged after ~ 1800 f_{evals}, because $\|\nabla f(\mathbf{x}_i)\|$ was more than 10 orders of magnitude below $\|\nabla f(\mathbf{x}_0)\|$. It could very well be that $f = 0.8$ is close to the global minimum, considering the small dimension of the net and how well we expect it to perform on the dataset.

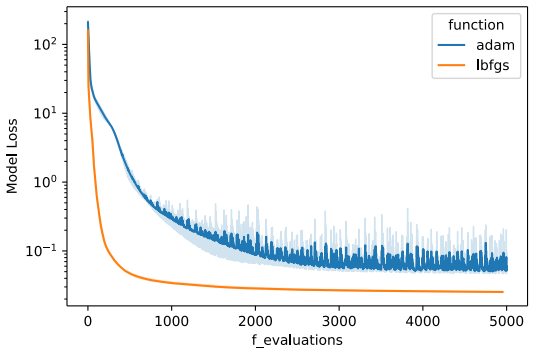
Despite three different weights initializations, L-BFGS shows an extremely low variance (not even visible in Figures 6.b and 6.c), which is for sure another appreciable feature.

If compared on the f_{evals} needed to reach a certain level of f, L-BFGS always outperforms ADAM. Nevertheless, the cost of each f_{eval} is higher for L-BFGS then for a simple SGD. To provide quantitative measurements of this time complexity, Table 3 reports the time (in seconds) needed by each algorithm to perform 5000 f_{evals}, regardless of the f value that is reached.

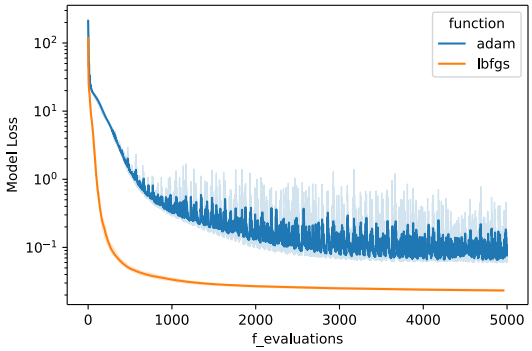
A rough estimation is that an L-BFGS iteration takes twice the time of an ADAM one. Considering the evident performance gap of Figure 6 (around one order of magnitude),



(a) M2.a



(b) M2.b



(c) M2.c

Figure 6: LBFGS vs ADAM: f value over f_{evals} (3 runs per plot)

the time per iteration seems a negligible factor. Even for time-constrained applications, L-BFGS appears better: stopping both the algorithms after a fixed time t_0, L-BFGS still finds lower values of f than SGD.

Solver	M2.b	M2.c
L-BFGS	93.1s ($\pm$0.49s)	122.7s ($\pm$1.7s)
ADAM	55.3s ($\pm$2.3s)	63.1s ($\pm$2.2s)

Table 3: L-BFGS vs ADAM: *runtime* for 5000 f_{evals} on M2 (3 runs per cell)

6 Conclusions

L-BFGS. The algorithm shows consistent performances throughout the tests, outperforming a widely used method such as ADAM SGD on minima reached, time taken and learning curve smoothness. We initially expected that values of M far greater than 20 could slow down significantly the algorithm; but the tests revealed the opposite, as the computational bottleneck seems to lie on the evaluation of f and ∇f. Furthermore, in

most of ML applications the dataset is made of thousands (or millions) of patterns. As the batch size increases, the Two-Loop recursion progressively loses computational impact w.r.t. computing f, leaving the user free to tune M without feasibility concerns.

As for the rate of convergence, our experiments never show a super-linear rate of convergence on the norm of the loss gradient. This does not surprise us: as discussed above, it is not guaranteed by the theory on our applications of the algorithm. This does not prove to be a problem in the case of ML applications, as the only concern in these cases is minimizing the loss function, which L-BFGS accomplishes quite well.

Level Bundle method. Although there is no theoretical guarantee for the algorithm to work properly on a non-convex objective function, the results show that the Level Bundle method can indeed be applied to such problems. With a proper choice of the hyperparameter λ, the algorithm terminates successfully on all the runs.

However, when compared to ADAM SGD, the algorithm proves to be too computationally heavy to justify its use, even employing state of the art solvers for the two master problems at each step. Looking at the number of f_{evals}, the two algorithms may seem to perform similarly; but looking at Table 1, we see that the *runtime* of the Bundle method is from 1 to 3 orders of magnitudes greater than ADAM SGD. Therefore, when compared on time, ADAM SGD is actually able to find better (smaller) values of the loss.

References

[1] Yurii Nesterov Claude Lemaréchal, Arkadii Nemirovskii. *New variants of bundle methods*, pages 111–147. 1995. https://doi.org/10.1007/BF01585555.

[2] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks, 2021.

[3] Yun hai Xiao, Ting feng Li, and Zeng-Xin Wei. Global convergence of a modified limited memory bfgs method for non-convex minimization, 2013.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[5] Dong-Hui Li and Masao Fukushima. A modified bfgs method and its global convergence in nonconvex minimization. *Journal of Computational and Applied Mathematics*, 129(1):15–35, 2001. Nonlinear Programming and Variational Inequalities.

[6] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, 1999.