Neuradillo

A neural network simulator

Giovanni Sorice
Francesco Corti

f.corti3@studenti.unipi.it
g.sorice@studenti.unipi.it

Computational Mathematics for Learning and Data Analysis, A.Y. 2019-2020

Date: *2/08/2020*

Type of project: ML project

Abstract

(M1) is a neural network with topology and activation function of your choice, provided it is differentiable, but mandatory L_1 regularization.

(M2) is a neural network with topology and activation function of your choice but mandatory L_2 regularization.

(A1) is a standard momentum descent approach.

(A2) an algorithm for L_2 regularization of the class of limited-memory quasi-Newton methods.

(A3) an algorithm for L_1 regularization of the class of bundle methods.

1 Introduction

At the beginning we provide a short description of the problem. Next, we talk about the implemented method to find the gradient and the activation function used in the experiments. In the end, we give some information about the used regularization method.

1.1 Neural network

Let M be a neural network with a specific topology. The main goal of this project was to learn and develop three different optimization methods for M. We exploit:

- Standard momentum descent approach;
- L-BFGS algorithm of the class of limited-memory quasi-Newton methods for L2 regularization;
- Proximal Bundle Method algorithm of the class of bundle methods for L1 regularization;

1.2 Backpropagation

The backpropagation algorithm can be divided into two phases (as mentioned in [4]):

1. Compute the network's gradient that is the derivative of the cost function $\nabla_\theta J(\theta)$, with θ representing the ANN's parameters (weights and bias). The algorithm used to compute the gradient is the *back-propagation* described by algorithm 1 and 2;
2. Use the knowledge of the gradient to do the next step using one of the optimizers chosen;

The computation of the gradient is divided in two parts that are *forward* and *backward*. In the *forward* phase the input matrix x flow through the network ending in the output layer that computes the output *h(x)*. This later is compared with the desired vector values $\widehat{y}$:

Algorithm 1 Forward propagation

```
1: procedure FORWARD PROPAGATION
2:     $\mathbf{h}_0 = \mathbf{x}$
3:     for $k = 1, \ldots, l$ do
4:         $\mathbf{a}_k = \mathbf{b}_k + \mathbf{W}_k \mathbf{h}_{k-1}$
5:         $\mathbf{h}_k = f(\mathbf{a}_k)$
6:     end for
7:     $\mathbf{h}(\mathbf{x}) = \mathbf{h}_l$
8:     $J = L(\mathbf{h}(\mathbf{x}), \mathbf{y}) + \lambda\Omega(\theta)$
9: end procedure
```

In our case, J is the *Mean Squared Error* function. Since the neural network is a composition of functions the Chain Rule is used to compute the partial derivative of the weights that compose it.

Algorithm 2 Backward computation

```
procedure Backward propagation
    $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\mathbf{h}(\mathbf{x}), \mathbf{y})$
    for $k = l, l-1, \ldots, 1$ do
        $\mathbf{g} \leftarrow \nabla_{\mathbf{a}_k} J = \mathbf{g} \odot f'(\mathbf{a}_k)$
        $\nabla_{\mathbf{b}_k} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}_k} \Omega(\theta)$
        $\nabla_{\mathbf{W}_k} J = \mathbf{g}\mathbf{h}_{k-1}^T + \lambda \nabla_{\mathbf{W}_k} \Omega(\theta)$
        $\mathbf{g} = \nabla_{\mathbf{h}_{k-1}} J = \mathbf{W}_k^T \mathbf{g}$
    end for
end procedure
```

1.3 Activation function

The activation function of a node defines its output given an input or a set of inputs. Properties of this function are:

- Nonlinear;
- Range;
- Continuously differentiable;
- Monotonic;
- Smooth functions;
- Approximates identity near the origin;

For the aim of our project, we chose to use two different activation function:

- Sigmoid (or standard logistic function):
  - It is defined between (0,1);

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$
$$f'(x) = \sigma'(x) = f(x)(1-f(x)),$$

- TanH;
  - It is defined between (-1,1);

$$f(x) = \sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$f'(x) = \sigma'(x) = 1 - f(x)^2,$$

The purpose of the nonlinearity is to ensure that the neural network is a universal function approximator.

1.4 Loss Function

The *Loss Function* is a function used to evaluate the performance of a model, given the $h(x)$ vector compute by the network and the desired vector values $\widehat{y}$ measures the average of the squares of the errors. There are several *Loss Function* used in machine learning algorithms but we are going to focus on the *Mean Squared Error* (MSE). This is obtained by the formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(h(x) - \widehat{y})_i^2 \tag{1}$$

where n represents the number of sample input data we passed into the model. The *Loss Function* can be represented as a composition of the Euclidean norm and the quadratic function

$$MSE = \frac{1}{n} \parallel h(x) - \widehat{y} \parallel_2^2 \tag{2}$$

Moreover, the training phase of a supervised machine learning algorithm can be seen as an optimization (in our case minimization) of the *Loss Function* by altering the weights of the network w. Since the purpose of neural networks is to build models that fit data, we want to minimize the *Loss Function* to have a good prediction on unseen data. This minimization process is done through optimization algorithms but to minimize the *Loss Function* it must have certain properties. The gradient for the MSE concerning x_i is defined as:

$$\nabla_{x_i} MSE(x, \hat{y}) = 2 * (h(x) - \hat{y})_i * h'(x) \tag{3}$$

The ∇w is equal to:

$$\nabla w = -\frac{\partial MSE(x,\hat{y})}{\partial w} = -\sum_{i=1}^{n}\frac{\partial MSE(x,\hat{y})_i}{\partial w} = \sum_{i=1}^{n} -\frac{\partial MSE(x,\hat{y})_i}{\partial w} = \sum_{i=1}^{n}\nabla_i w \tag{4}$$

The $\nabla_i w$ for a generic t is equal to:

$$\nabla_i w_t = -\frac{\partial MSE(x,\hat{y})_i}{\partial w_t} = -\frac{\partial MSE(x,\hat{y})_i}{\partial o_t} * \frac{\partial o_t}{\partial net_t} * \frac{\partial net_t}{\partial w_t}$$

where $o_t = f_t(net_t)$, f_t is the activation function at layer t, $net_t = \sum_{i=1}^{n} w_{t,i} * o_{t-1,i}$ and o_0 are the inputs.
So, $\frac{\partial net_t}{\partial w_{t,i}}$ is equal to:

$$\frac{\partial net_t}{\partial w_t} = \frac{\partial \sum_{r=1}^{n} w_{t,r} * o_{t-1,r}}{\partial w_{t,i}} = o_{t-1,i}$$

The term $\frac{\partial o_t}{\partial net_t}$ is equal to:

$$\frac{\partial o_t}{\partial net_t} = f'(net_t)$$

We define:

$$\delta_t = -\frac{\partial MSE(x,\hat{y})_i}{\partial o_t} * \frac{\partial o_t}{\partial net_t}$$

Now we have to study two different case for $\frac{\partial MSE(x,\hat{y})_i}{\partial o_t}$, when t is the output layer and when t is a hidden layer.
Case t = k (where k is the last layer):

$$\frac{\partial MSE(x,\hat{y})_i}{\partial o_k} = -\frac{1}{2} * \frac{\sum_{r=1}^{n} \partial((h(x) - \widehat{y})_r^2)}{\partial o_k} = -\frac{\sum_{r=1}^{n}(h(x) - \hat{y})_r * h'(x)\partial((h(x) - \widehat{y})_r)}{\partial o_k} = (h(x) - \hat{y}) * h'(x)$$

So, δ_t is equal to:

$$\delta_k = (h(x) - \hat{y}) * h'(x) * f'(net_k)$$

Case t = j (where j is a hidden layer):

$$\frac{\partial MSE(x,\hat{y})_i}{\partial o_j} = -\frac{1}{2} * \sum_{r=j}^{k} \frac{\partial((h(x)-\widehat{y})_r^2)}{\partial o_r} = -\frac{1}{2} * \sum_{r=j}^{k} \frac{\partial((h(x)-\widehat{y})_r^2)}{\partial o_r} * \frac{\partial o_r}{\partial net_r} * \frac{\partial net_r}{\partial o_j}$$

$$= \sum_k \delta_k * w_{k,j} \tag{5}$$

Where:

$$\frac{\partial net_r}{\partial o_j} = \frac{\sum_{r=1}^{n} \partial w_{k,r} * o_r}{\partial o_j} = w_{k,j}$$

After the derivation we can state that in the output layer k the gradient is defined as:

$$\nabla w_k = (h(x) - \hat{y}) * h'(x) * f'(net_k) * o_{t-1,i} \tag{6}$$

and in the hidden layer j is defined as:

$$\nabla w_j = (\sum_k \delta_k * w_{k,j}) * f'(net_j) * o_{t-1,i} \tag{7}$$

1.4.1 Loss Function properties

In an optimization problem given X any set and $f : X \to \mathbb{R}$ any function we want

$$(P) \quad f_* = min\{f(x) : x \in X\} \tag{8}$$

where X is the feasible region, f is the objective function and $v(P) = f_*$ is the optimal value. In our case $X \subseteq \mathbb{R}^n$ and we want to be sure that exists an optimal solution.

So we want to find an optimal solution: $x_* \in X$ such that $f(x_*) = f_*$ but this can be impossible for many reasons. For the *Bolzano-Weierstrass* theorem to ensure that our function has an optimal solution we need that $X \subseteq \mathbb{R}^n$ is compact and f is continuous (or lower semi-continuous) and differentiable.

- **Continuity**: A function f is *Lipschitz continuous* on its domain S if $\exists L > 0$ such that:

  $$|f(x) - f(y)| \leq L \parallel x - y \parallel \quad \forall x, y \in S, \tag{9}$$

  more formally a function is *Locally Lipschitz Continuous* at x if $\exists \varepsilon > 0$ s.t $S \in \beta(x, \varepsilon)$ and it is *Global Lipschitz Continuous* if f is *Locally Lipschitz Continuous* on all the space S, in our case R^n. Neural networks are a series of function composition:

  $$h(x) = \phi_k(b_k + \sum_j w_{kj}\phi_j(b_j + \sum_i w_{ji}x)) \tag{10}$$

  where x is the input and ϕ_i is an activation function. Theorem 12.6 of [7] say that: Let f_1 be Lipschitz continuous on a set I_1 with Lipschitz constant L_1 and f_2 be Lipschitz continuous on I_2 with Lipschitz constant L_2 such that $f_1(I_1) \subset I_2$. Then the composite function= $f_1 \circ f_2$ is Lipschitz continuous on I_1 with Lipschitz constant $L1 * L2$.
  In our case MSE is a composition of the Euclidean norm, that is quadratic and Lipschitz continuous function, and the output function, that is the composition between the used activation functions that are Lipschitz continuous. So, to ensure that the Theorem 12.6 of [7] is valid in our case, the computed value of the output function has to live in a bounded set. As mentioned in §1.3 each layer can use a *sigmoid* that bounds the output between (0,1) or a *tanH* that bounds the output between (-1,+1) as the activation function. Since every layer's output is bounded and the Euclidean norm receive an input in a bounded interval then for Theorem 12.6 of [7] the MSE is a Lipschitz continuous function.

- **Differentiability**: the network uses *sigmoid* (§1.3) and *hyperbolic tangent* (§1.3) as activation functions. These functions are continuous and twice differentiable with bounded Lipschitz continuous derivative. So *Mean Squared Error* is a differentiable function since the network is a composition of continuously differentiable functions. Moreover, in our case the gradient has to be Lipschitz continuous for the Momentum Descent Approach and for proving the superlinear convergence of L-BFGS, instead for the Proximal Bundle Method we do not need this property to converge to local minima. This implies the following discussion. The Theorem 12.4 of [7] states that if f_1 and f_2 are Lipschitz continuous on a bounded interval I then $f_1 * f_2$ is Lipschitz continuous on I. For this theorem the gradient of our loss function is Lipschitz continuous if we use only activation functions with Lipschitz derivative and we restrict the weights to a bounded set. Also in derivation 4, which leads to 6 and 7, we can see that the gradient is bounded if the weights are restricted in a bounded set because it is a composition between values (weights and activation function output) and derivatives of the activation function that are bounded. The weights of the network are initialized using a uniform distribution, taken in the range $[1, -1]$. We can observe that the weights are in a compact set if the algorithm guarantees a monotone sequence of objective values (i.e. $f(x_i)$) when it is converging to local minima. The Momentum Descent Approach produces a monotone sequence of objective values when the step length (also known as learning rate) is sufficiently small (this is verified in [1] §2.2 paragraph "Search directions for line search methods"). In our implementation, we use a grid search with fixed step size, so with it, we try to find a sufficiently small step length that produces a monotone sequence of objective values. Also, L-BFGS method produces a monotone sequence of objective values when an Armijo-Wolfe line search is used to find the right step length, moreover this line-search ensure that the *curvature condition* is satisfied (more information at 28). We define the objective value as $f(w) = Loss(w) + \lambda||w||$ where $Loss(w)$ is a non negative function, $\lambda > 0$ and $||w||$ is the regularization term. We can state that $f(w_0) = Loss(w_0) + \lambda||w_0||$ where w_0 is the initial point. We can assert that if the algorithm guarantees a monotonic sequence of objective values (that is our case for both Momentum Descent Approach with a sufficiently small step length and L-BFGS since Armijo-Wolfe line search is used) then $f(w_i) = Loss(w_i) + \lambda||w_i|| \leq f(w_0) \, \forall i$. It follows that $||w_i|| \leq f(w_0) - Loss(w_i) \leq f(w_0)$ because $Loss(w_i) \geq 0$. For this reason w_i lives in a compact set. Also, there exist variants of Nesterov Descent Approach that guarantees a monotonic sequence of objective values (see §2.1). But, the Nesterov Descent Approach, that we have implemented, does not guarantee the production of a monotone sequence of objective values. We found out during the experimental phase that in the problems that we chose, the weights of the Nesterov Descent Approach lives in a compact set and this is all we need. Therefore, after analysing the results of the experimental phase, we can state that in our problems the gradient is Lipschitz continuous also for the Nesterov Descent Approach.
- **Convexity**: all the functions used as activation function (*sigmoid* and *tanH*) are not convex functions. Since our MSE (*Loss Function*) is obtained combining these not convex functions MSE is not convex.

1.5 Regularization

In machine learning, the regularization is used to ensure a trade-off between accuracy in the training set and complexity of the model. We implemented and used two types of regularization, L1 and L2. They are implemented adding at the objective Loss Function a penalty term multiplied by a lambda parameter.

$$\mathbf{W} \in \mathbb{R}^n \; L(\mathbf{W}) + \lambda\Omega(\mathbf{W})$$

L1 regularization Usually named as Lasso regression, defined as: $\Omega(\mathbf{W}) = \sum_{i=1}^{k} |w_i| = \|\mathbf{W}\|_1$.

L2 regularization Usually named as Ridge regression, defined as: $\Omega(\mathbf{W}) = \sum_{i=1}^{k} w_i^2 = \|\mathbf{W}\|_2^2$.

2 Method

2.1 Momentum descent approach

The momentum descent approach add to the deterministic gradient descent (GD) a velocity vector in directions of persistent reduction in the objective function across iterations for accelerating gradient descent.
This algorithm is a gradient-based optimization. The main idea of this type of algorithm is to minimize a loss function (or cost/error function) following the direction given by the gradient computed in the current point of the function.
What can be a problem in the gradient descent is the large number of examples that must be computed at each iteration. Instead, in the stochastic gradient descent (SGD) a small number of examples are computed in online training. However, the SGD is not as stable as the classical GD, but it requires only the evaluation of one example for each iteration. Form now on, when we talk about SGD, we refer to SGD with a subset of the training dataset, called mini-batch or batch SGD.

Momentum descent approach use momentum to accelerate learning. It introduces a variable $\mathbf{v}$ that take account of the speed and the direction at which the parameters move through parameters space. The intensity of momentum is determined by the hyperparameter $\alpha \in [0, 1)$. The update rule is the following:

$$\mathbf{v}_k = \alpha \mathbf{v}_{k-1} + \eta \nabla L(\mathbf{W}_k). \tag{11}$$

$$\mathbf{W}_k = \mathbf{W}_{k-1} + \mathbf{v}_k. \tag{12}$$

We would like to observe that the method choose a descent direction equals to $\frac{\partial f}{\partial d_i}(x_i) < 0$. This means that $\langle d_i, \nabla f(x_i) \rangle < 0$ and the $\cos(\theta_i) > 0$. The main idea is to take the information given by the directional derivatives on the way that our function decrease and follow these directions. So, the method follows a descent direction, the best case is to take the direction d_i with the same direction of $-\nabla f(x_i)$.
The Nesterov momentum is a variant of the classical Momentum, the main idea is to add a correction factor at the classical momentum, for this reason they differ in the place where the gradient is evaluated. With Nesterov momentum the rate of convergence goes from $O(\frac{1}{k})$ to $O(\frac{1}{k^2})$. The update rule for the velocity vector $\mathbf{v}$ is the following:

$$\mathbf{v}_k = \alpha \mathbf{v}_{k-1} + \eta \nabla L(\mathbf{W}_k + \alpha \mathbf{v}_{k-1}). \tag{13}$$

Also, there exists a variant of the Nesterov momentum that guarantees a monotonic sequence of objective values and it is named Nesterov Momentum Approach with restart. More information can be found in the paper [15]. This property could be useful to state that the gradient is Lipschitz continuous, but we did not have implemented it.

2.1.1 Algorithm

Algorithm 3 Momentum Descent Algorithm. The learning rate η, the α term and the maximum number of iterations are given.

```
Require: Learning rate $\eta$ and momentum parameter $\alpha$
Require: Maximum number of iteration and error threshold
 1: procedure MOMENTUM DESCENT
 2:     Initialize $\mathbf{W}$ and $\mathbf{v}$
 3:     $k \leftarrow 0$
 4:     while $k < max\_iterations$ && $error\_th < e$ do
 5:         if Nesterov Momentum then
 6:             $\tilde{\mathbf{W}} \leftarrow \mathbf{W} + \alpha \mathbf{v}$
 7:         end if
 8:         Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{n}\nabla \sum_i L(\tilde{\mathbf{W}})$
 9:         Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \mathbf{g}$
10:         Apply update: $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$
11:     end while
12: end procedure
```

2.1.2 Method convergence

Convergence of this algorithm is strictly connected to the choice of η, the learning rate parameter. It is important to gradually decrease η over iterations, to guarantee convergence. A sufficient condition to ensure convergence of momentum descent is that

$$\sum_k \eta_k = \infty \quad \text{and} \quad \sum_k \eta_k^2 < \infty \tag{14}$$

The convergence rate is of $O(\frac{1}{\sqrt{k}})$ (after k steps) when the algorithm is applied to a convex problem. Moreover, if we assume that the Hessian matrix $\nabla^2 f(x_*)$ is positive definite, means that the loss function is a strongly convex function and the convergence rate is $O(\frac{1}{k})$. [11]

Our objective function is not convex, so we need more assumptions to ensure the convergence. D.P. Bertsekas proved the following, which appears as proposition 1.2.4 of [12].
Let $\{x_k\}$ be a sequence generated by a gradient method (in our specific case a Momentum descent approach with a direction d_k defined as stated in §2.1) $x_{k+1} = x_k + \alpha_k d_k$. Assume that for some constant L>0 we have

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|, \forall x, y \in R^n \tag{15}$$

and that there exist positive scalars c_1, c_2 such that for all k the Armijo-Wolfe conditions are respected. Suppose also that

$$\eta_k \to 0 \enspace , \enspace \sum_k \eta_k = \infty \tag{16}$$

Then either $f(x_k) \to -\infty$ or else $\{f(x_k)\}$ converges to a finite value and $\nabla f(x_k) \to 0$. Furthermore, every limit point of $\{x_k\}$ is a stationary point of f. The first fundamentals observation is that the loss function that we choose is a quadratic and it is two times Lipschitz continue and derivable (as we said in §1.4.1).

As stated in [1] in Chapter 3 the Armijo-Wolfe conditions are defined as:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, \tag{17}$$

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k, \tag{18}$$

Also, our implementation does not use a line search to decide the step length and this means that the Bertsekas results are not valid in our case.

Indeed, we have implemented a grid search for the choice of the fixed step length. Grid search is the traditional method of hyperparameters optimization. It simply makes a complete search over a given subset of the hyperparameters space of the training algorithm. In our case, we used as hyperparameters the step length and the momentum rate. Grid search suffers from high dimensional spaces, but in our case, the dimensional space generated from the two hyperparameters is not too high. In theory, you have to tests all possible permutation combinations of hyperparameters of given Machine Learning algorithm, in practice, grid search is used to create different size of hyperparameters permutation combinations that gradually became more fine-grained.

We cannot state anything a priori regarding the convergence of the Nesterov Momentum Method applied to a non-convex problem with fixed step size found through the grid search method. But after analysing the results of the experimental phase, we can state that in our problems Nesterov Descent Approach has a linear convergence.

2.2 Limited-memory quasi-Newton methods

Limited-memory Quasi-Newton methods are useful for solving large unconstrained optimization problems in which Hessian matrices cannot be computed at a reasonable cost. More formally to minimize a function using the second order information, we create a quadratic model that approximate the function at the current x_k:

$$m_k(p) = L_k(p) + \frac{1}{2}(p-k)^T \nabla^2 f(k)(p-k). \tag{19}$$

where $L_k(p)$ is the first order model and the remaining is the second order term. Then we minimize the function using the information given by the model m_k and the direction compute as

$$d_k \leftarrow -[\nabla^2 f(x_k)]^{-1} \nabla f(x_k). \tag{20}$$

Theoretically the Hessian matrix $\nabla^2 f(k)$ needs to be recomputed at every iteration. Instead in the Quasi-Netwon method the Hessian is substituted by an approximation H_k to reduce the cost.

The main idea of L-BFGS is to use the curvature information from the m previously step to construct B_k, the inverse of the Hessian (H_k^{-1}), to avoid using $O(n^2)$ storage. The information about B_k is stored inside vectors of length n. Earlier iterations parameters are discarded in the interest of saving storage.

2.2.1 Limited memory BFGS

In the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) each step use the information given by the gradient ∇f_k and the approximated inverse of the Hessian B_k to minimize a function updating its parameters x_k. Each step has the form:

$$x_{k+1} = x_k - \alpha_k B_k \nabla f_k \tag{21}$$

where α_k is the step length.

In L-BFGS method to handle the complexity of storing an approximation of the Hessian, we save a certain number of vector m composed of

$$s_k = x_{k+1} - x_k = \alpha_k d_k, \qquad y_k = \nabla f_{k+1} - \nabla f_k. \tag{22}$$

We can now approximate the product $B_k \nabla f_k$ by doing multiple products and vector summation using the m vectors $\{s_i, y_i\}$. At step k after we compute the new iterate and the oldest vector $\{s_m, y_m\}$ is replaced with the new one.

We first choose some initial Hessian approximation B_k^0 that is allowed to vary from iteration to iteration and then we find B_k with

$$\begin{aligned} B_k &= (V_{k-1}^T \cdots V_{k-m}^T) B_k^0 (V_{k-m} \cdots V_{k-1}) \\ &+ \rho_{k-m} (V_{k-1}^T \cdots V_{k-m+1}^T) s_{k-m} s_{k-m}^T (V_{k-m+1} \cdots V_{k-1}) \\ &+ \rho_{k-m+1} (V_{k-1}^T \cdots V_{k-m+2}^T) s_{k-m+1} s_{k-m+1}^T (V_{k-m+2} \cdots V_{k-1}) \\ &+ \cdots \\ &+ \rho_{k-1} s_{k-1} s_{k-1}^T. \end{aligned} \tag{23}$$

where

$$\rho_k = \frac{1}{y_k^T s_k}, \qquad V_k = I - \rho_k y_k s_k^T, \tag{24}$$

formulas taken from [1].

Now we can derive an algorithm to compute the product $B_k \nabla f_k$

Algorithm 4 L-BFGS two loop recursion

```
    $q \leftarrow \nabla f_k$;
    for $i = k-1, k-2, \cdots, k-m$ do
3:      $\alpha_i \leftarrow \rho_i s_i^T q$;
        $q \leftarrow q - \alpha_i y_i$;
    end for
6:  $r \leftarrow B_k^0 q$;
    for $i = k-m, k-m+1, \cdots, k-1$ do
        $\beta \leftarrow \rho_i y_i^T r$;
9:      $r \leftarrow r + s_i(\alpha_i - \beta)$
    end for
    stop with result $B_k \nabla f_k = r$
```

A method for choosing B_k^0 that has been proved effective in practice is to set $B_k^0 = \gamma_k I$ where

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \tag{25}$$

γ_k is the scaling factor that attempts to estimate the size of the true Hessian matrix along the most recent search direction [1]. This choice helps to ensure that the search direction p_k is well scaled.

To keep the update of the parameters stable using the approximated inverse of the Hessian B_k, a condition based on the knowledge gained during the latest step is imposed. The new model must respect old information

$$\nabla m_{k+1}(x_k) = \nabla f(x_k), \tag{26}$$

that is ensured if the *secant equation*

$$B_{k+1}y_k = s_k \tag{27}$$

is satisfied. This is possible if s_k and y_k satisfy the *curvature condition*

$$s_k^T y_k > 0. \tag{28}$$

The *curvature condition* is satisfied when f is strongly convex but for nonconvex functions we need to restrict the step α_k used in the line search procedure to guarantee the satisfiability of it. Moreover the *curvature condition* is satisfied in the line search if we impose Wolfe conditions 17 and 18 or strong Wolfe conditions

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, \tag{29}$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq c_2 |\nabla f_k^T p_k|, \tag{30}$$

with $0<c_1<c_2<1$.

As shown in [1], we need to ensure that among all the symmetric matrix satisfying the secant equation, B_{k+1} is the closest to the current matrix B_k. This condition of closeness to B_k is resolved by solving the problem

$$B_{k+1} = \min_B \parallel B - B_k \parallel \tag{31}$$

with

$$B = B^T, \quad By_k = s_k \tag{32}$$

for $\parallel \cdot \parallel_F$ the formula to compute B_{k+1} is found.

2.2.2 Algorithm

Limited-memory variants of the quasi-Newton approach use Hessian approximations that can be stored compactly by using just few vectors of length n. These methods are fairly robust, inexpensive and easy to implement.

Algorithm 5 L-BFGS.

```
Require: Starting point $x_0$;
Require: Integer m>0;
Require: Maximum number of iteration and error threshold;
    procedure LBFGS
        Compute $k \leftarrow 0$;
        repeat
4:          Choose $B_k^0$;
            Compute $p_k \leftarrow B_k \nabla f_k$;
            Compute $x_{k+1} \leftarrow x_k + \alpha_k p_k$ where $\alpha_k$ is chosen to satisfy the Wolfe condition;
            if k>m then
8:              Discard the vector pair $\{s_{k-m}, y_{k-m}\}$ from storage;
            end if
            Compute and save $s_k \leftarrow x_{k+1} - x_k, y_k = \nabla f_{k+1} - f_k$;
            $k \leftarrow k + 1$
12:     until convergence.
    end procedure
```

Each iteration can be performed at cost of $O(mn)$ arithmetic operations, where m is the number of steps stored in memory.

Our implementation uses the update rule of BFGS-Level described in [3] chapter 3. The main idea is to avoid the second-order interactions between weights of different levels and consider a separate approximate Hessian matrix (in our case defined as B_k) for each level of the network. As written in [3] this approach reduces considerably the total size of the matrix to be computed, since the Hessian matrix of a neural network is sparse with respect to the output layer because there are no interactions between output neurons.

2.2.3 Algorithm Convergence

L-BFGS guarantee global convergence under specific assumptions:

1. The objective function f is twice continuously differentiable;
2. The approximated inverse Hessian B_k need to be positive definite to take the correct descent direction.

Assumption 1 is satisfied, since our loss function f is twice continuously differentiable as we said in 1.4.1.

Assumption 2 is satisfied because L-BFGS computes an approximation of the inverse of the Hessian B_k that is directly multiplied with the gradient ∇f_k to obtain a direction p_k (see algorithm 4), so methods that manually modify the Hessian to make it positive definite are not used. Indeed, to produce a descent direction we need to guarantee that B_{k+1} will be positive definite. This is true whenever B_k (past approximation) is positive definite, as shown in [1] section 6.1 (subsection "Properties of the BFGS method"). Produce B_{k+1} that is correctly defined is ensured whenever the initial approximation B_0 (first B_k) is positive definite and the *secant equation* is satisfied for every k step of the minimization process (which is our case since B_0 it is initialized as positive definite and the secant equation is checked at every iteration).

Also if the α_k (step length) is computed by an inexact line search that satisfies the Wolfe conditions and the first step tried has always value equal to 1, then the search direction of a quasi-Newton method approximates the Newton direction well enough so that the step size α_k will satisfy the Wolfe conditions as the iterates converge to the solution (as reported in [1] chapter 3.3 paragraph "Quasi-Newton method"). This presents a condition that if it is satisfied produces a superlinearly convergent iteration for all x_k in the proximity of a solution x_* in a convex case. But in our case, since the function to be optimized is non-convex we need to make additional assumptions. We first present the assumptions needed to obtain a superlinear convergence in the convex case, after that the additional assumptions needed for the non-convex case are discussed.

Theorem 3.6 of [1] states: suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable. Consider the iteration $x_{k+1} = x_k + \alpha_k d_k$, where d_k is a descent direction and α_k satisfies the Wolfe conditions with $c_1 <= \frac{1}{2}$ (which is our case). If the sequence $\{x_k\}$ converges to a point x_* such that $\nabla f(x_*) = 0$ and $\nabla^2 f(x_*)$ is positive definite, and if the search direction satisfies:

$$\lim_{k\to\infty} \frac{\|(\nabla f_k + \nabla^2 f_k d_k)\|}{\|d_k\|} = 0. \tag{33}$$

then

- the step length $\alpha_k = 1$ is admissible for all k greater than a certain index k_0;
- if $\alpha_k = 1$ for all $k > k_0, x_k$ converges to x_* superlinearly.

Indeed since the objective function is twice continuously differentiable (as mention in §1.4.1), $c_1 < \frac{1}{2}$ (we use 1e-4) and d_k is equal to $B_k \nabla f_k$ (which is our case as mention in 21) *Theorem 3.6* obtained equation 33 is equivalent to:

$$\lim_{k\to\infty} \frac{\|(B_k - \nabla^2 f(x_*)d_k)\|}{\|d_k\|} = 0. \tag{34}$$

This shows that a superlinear convergence rate can be obtained even if the sequence of B_k does not converge to $\nabla^2 f(x_*)$. It is sufficient that the B_k is positive definite (which is our case as mention above, section 6.1 of [1]) and its accuracy approximations increase to $\nabla^2 f(x_*)$ progressively. The condition 34 is sufficient to ensure the local superlinear convergence of L-BFGS (more in general of the quasi-Newton methods). This is formalized in *Theorem 3.7* (of [1]): suppose that $f: \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable. Consider the iteration: $x_{k+1} = x_k + p_k$ (i.e. the step length α_k is uniformly 1) and that p_k is given by $B_k \nabla f_k$ (as mention in 21). Let us assume that x_k converges to a point x_* such that $\nabla f(x_*) = 0$ and $\nabla^2 f(x_*)$ is positive definite. Then x_k converges superlinearly if and only if 34 holds. The proof can be found in [1] (Theorem 3.7).

In a non-convex case (which is our case as mentioned in §1.4.1) additional assumptions are needed to guarantee a superlinear convergence. The sequence $||x_k - x^*||$ has to converge in a such a way that:

$$\sum_{k=1}^{\infty} ||x_k - x^*|| < \infty \tag{35}$$

is obtained. Then *Assumption 6.2* of [1] is required: the Hessian matrix G is Lipschitz continuous at x^*, that is:

$$||G(x) - G(x^*)|| \leq L||x - x^*|| \tag{36}$$

for all x near x^*, where L is a positive constant. These two properties are needed by *Theorem 6.6* of [1], it states: suppose that f is twice continuously differentiable and that the iterates generated by the BFGS algorithm converge to a minimizer x^* at which equation 36 holds. Suppose also that equation 35 holds, then x_k converges to x^* with a superlinear rate. However, in our case we cannot verify these conditions, because we cannot state that the equation 35 is satisfied. Supposing that it is satisfied, we have to state that equation 36 holds. To do this we refer to the proof of *Assumption 6.2* of [1]. For these reasons we cannot surely state that a superlinear convergence rate will be obtained.

But, during the experimental phase we developed an Armijo-Wolfe line search that has a maximum number of iteration to obtain a feasible value in a finite interval of time. For this reason, we guarantee the property of the matrix B_k to be positive definite by checking in each iteration the satisfiability of the *curvature condition* and we found that the rate of convergence of the algorithm is superlinear (see fig. 10a) even in our non-convex case.

If m is very small the behaviour of L-BFGS will be similar to the Gradient method and if m is very large the behaviour is similar as BFGS. However, m is problem dependent so a good trade-off between CPU computational time and convergence speed needs to be found when choosing it.

2.3 Bundle methods

Bundle methods are used mainly with non-differentiable functions. The main idea of these methods is to use past subgradients as global information. In convex function we know that at each iteration we get a subgradient in which the epigraph of the loss function is always above the epigraph of the subgradient, so epi(L_x) $\supseteq$ epi(f) .

We define as $\{x_i\}$ the sequence of iterates that will hopefully converge at some point x_*. The bundle of first-order information is:

$$\{x_k\} \rightarrow \mathcal{B} = \{(x_k, f^k = f(x_k), g^k \in \partial f(x_k)\} \tag{37}$$

and the cutting-plane model of f - $(1+\epsilon)$-order model as:

$$f_B(x) = \max\{f^k + \langle g^k, x - x_l\rangle : (x_k, f^k, g^k) \in B\} \tag{38}$$

in which g^i is the subgradient chosen at i^{th} iteration. Before getting deep in defining the bundle methods, we need to introduce some information about subgradient.

2.3.1 Subgradient

The subgradient is a generalization of gradients appropriate for convex functions, including those which are not necessarily smooth. The subgradient s of $f(x)$ is defined as:

$$f(y) \geq f(x) + s(y - x) \quad \forall y \in \mathbb{R}^n. \tag{39}$$

The set of all subgradients at a point is called the sub-differential, and it is denoted by $\partial f(x)$. If at a point the function is differentiable, the subgradient is exactly the gradient of the function at that point ($\partial f(x) = \{\nabla f(x)\}$). In a point in which the function is not differentiable, s is an infinity set of point closed and convex. If we choose s such that it respects:

$$\frac{\partial f}{\partial d}(x) = \sup\{\langle s, d\rangle : s \in \partial f(x)\} \tag{40}$$

we have that d is a descent direction $\iff \langle s, d\rangle < 0 \quad \forall s \in \partial f(x)$.
We defined s_* as the s with the steepest descent direction:

$$s_* = -argmin\{\| s \| : s \in \partial f(x)\} \tag{41}$$

2.3.2 Master problem

We know that f_B is a polyhedral function, for this reason is composed of a finite number of linear functions. In this way, we can define the master problem as:

$$min\{f_B(x)\} = min\{v : v \geq f^k + \langle g^k, x - x_k\rangle \qquad (x_k, f^k, g^k) \in B\} \tag{42}$$

A method to use efficiently the equation 42 is the *cutting plane algorithm* in which at every iteration we add the knowledge of the g^i sub-gradient at B if the v^i is greater than $f(x_*)$ otherwise we find the optimal value of f. Unfortunately, the master problem as defined in 42 have some issue when x_i is very far from x_*. Due to this, the cutting plane algorithm does not work very well in practice. For this reason, we use a stabilized master problem:

$$d_k = argmin\{f_B(x) \ + \mu \| x - \overline{x} \|_2^2 /2\} \tag{43}$$

in which $\overline{x}$ is a particular point called *stability center* and μ is the *stability parameter*. It is a term that it is introduced to take close the new point x_{*B} from $\overline{x}$. Usually, $\overline{x}$ is chosen as the best x_i at the moment and μ is a hyperparameter. For a large μ the parable is extremely sticky and small movement in the next iteration are done. Instead, for a small μ, the parable is too flat and in this case the same result as in the un-stabilized cutting plane method can be obtained. For this reason, is important to find the right μ to use.

2.3.3 Proximal Bundle Method

The main idea behind the Proximal Bundle Method is to approximate the objective function f by the *cutting-plane* model as mention in 38. To obtain the step size to apply into the result d_k obtained solving the *Master problem*, the following line search procedure is used (as described in [19] §12.1): Assume that $m_L \in (0, \frac{1}{2}, m_R \in (m_L, 1))$ and $\bar{t} \in (0, 1])$ are fixed line search parameters. The first search for the largest number $t_L^k \in [0, 1]$ such that $t_L^k \geq \bar{t}$ and

$$f(x_k + t_L^k d_k) \leq f(x_k) + m_L t_L^k v_k, \tag{44}$$

where v_k is the predicted amount of descent

$$v_k = \hat{f}_k(x_k + d_k) - f(x_k) < 0. \tag{45}$$

If exists that parameter a *long serious step* is taken:

$$x_{k+1} = x_k + t_L^k d_k \text{ and } y_{k+1} = x_{k+1}. \tag{46}$$

Otherwise, if 44 holds but $0 < t_L^k < \hat{t}$, a *short serious step* is taken:

$$x_{k+1} = x_k + t_L^k d_k \text{ and } y_{k+1} = x_k + t_R^k d_k \tag{47}$$

and, if $t_L^k = 0$, a *null step* is taken:

$$x_{k+1} = x_k + t_L^k d_k \text{ and } y_{k+1} = x_k + t_R^k d_k \tag{48}$$

where $t_R^h > t_L^h$ is such that

$$-\beta_{k+1}^{k+1} + g^i d_k \geq m_R v_k. \tag{49}$$

To use β_{k+1}^{k+1}, α_j that is a *linearization error* has to be introduced:

$$\alpha_j^k = f(x_k) - f(y_j) - g^k(x_k - \hat{x}). \tag{50}$$

But in our case, since f is non-convex, α_j^k can be replaced by β_j^k called *subgradient locality measure* (see chapter 12 of [19]):

$$\beta_j^k = max\{|\alpha_j^k|, \gamma||x_k - y_j||^2\}, \tag{51}$$

Algorithm 6 PBM.

```
procedure PBM
    initialization:
    $x_1 \in \mathbb{R}^n$, $J = \{1\}$, $\bar{t} \in (0,1]$, $m_L \in (0, \frac{1}{2})$, $u_1 > 0$, $\epsilon > 0$ and $v_0 = -\epsilon$;
    Set k=1;
    Evaluate f($x_1$) and $\xi_1 \in \partial f(x_1)$;
    repeat
        $d_k \leftarrow argmin_{d \in \mathbb{R}^n} \{f_B(x_k + d) + \frac{1}{2}\mu_k d^T d\}$;
        Compute $v_k \leftarrow f_B(x_k + d) - f(x_k)$;
        Find step sizes $t_L^k$ and $t_R^k$;
        if $f(x_k + t_L^k * d_k) \leq f(x_k) + m_L * t_L^k * v_k$ then
            if $t_L^k > \bar{t}$ then
                LONG SERIOUS STEP
                Set $x_{k+1} = x_k + t_L^k * d_k$;
                Set $y_{k+1} = x_{k+1}$
                Evaluate $\xi_{k+1} \in \partial f(y_{k+1})$
                END LONG SERIOUS STEP
            else
                SHORT SERIOUS STEP
                Set $x_{k+1} = x_k + t_L^k * d_k$;
                Set $y_{k+1} = x_k + t_R^k * d_k$
                Evaluate $\xi_{k+1} \in \partial f(y_{k+1})$
                END SHORT SERIOUS STEP
            end if
        else
            NULL STEP
            Set $x_{k+1} = x_k$;
            Set $y_{k+1} = x_k + t_R^k * d_k$
            Evaluate $\xi_{k+1} \in \partial f(y_{k+1})$
            END NULL STEP
        end if
        Update $J_k$ and $u_k$ according some updating rules;
        $k = k + 1$;
    until $v_k > \epsilon$.
end procedure
```

2.3.4 Algorithm convergence

The convergence of the *Bundle method* is guarantee under specific assumptions as shown in theorem 7.16 in [16]:

1. The function f is convex;
2. The set $X \in R^n$ is convex and closed.

As mentioned in [13] and demonstrated by Y. Du and A. Ruszczyński in chapter 4: "Rate of convergence" of [17] ("Assumption 6: strong convexity of the function $f(\cdot)$") under the assumption that:

- The function $f(\cdot)$ has a unique minimum point x_*;
- $\exists\alpha > 0$, such that $f(x) - f(x_*) \geq \alpha \parallel x - x_* \parallel_2, \forall x \in \mathbb{R}^n$ with $f(x) \leq f(x_1)$ (where x_1 are solutions of problems 43 at the earlier iterations of the method).

Then the convergence rate is linear. In this case, we suppose that given the number of iterations k, the precision of the solution is approximately $O(1/k)$. Without any assumption, the convergence rate is sublinear.

The convergence theorem does not apply to our case, because our loss function is not convex. We think that this would be relevant in practice, but due to the *Lipschitz continuity* of our loss function we suppose that the method can converge.

In this direction, section 4: "Convergence Analysis" of [14] helped us to understand that under specific assumption and constraint, the proximal bundle method for non convex function surely converge. In [14] are defined the conditions for the convergence but it takes in consideration constrained problems. For this reason, we report the simplified assumption for unconstrained problems:

1. The function f is f°-pseudoconvex;
2. The function f is weakly semismooth;

The definition of *f°-pseudoconvex* and *weakly semismooth* are taken by chapter 2 of [14].
A function $f : \mathbb{R}^n \to \mathbb{R}$ is weakly semismooth if the classical directional derivative

$$f'(x, d) = \lim_{t \downarrow 0} \frac{f(x + t * d) - f(x)}{t} \tag{52}$$

exists for all x and d, and

$$f'(x, d) = \lim_{t \downarrow 0} \xi(x + t * d)^T * d \tag{53}$$

where $\xi(x + t * d) \in \partial f(x + t * d)$.
A function $f : \mathbb{R}^n \to \mathbb{R}$ is f°-pseudoconvex, if it is locally Lipschitz continuous and for all x, y $\in \mathbb{R}^n$

$$f(y) < f(x) \text{ implies } f^\circ(x; y - x) < 0 \tag{54}$$

Where $f^\circ(x; y)$ is the *Clarke's* generalized directional derivative of f at x in the direction y and it is defined as

$$f^\circ(x, y) = \lim_{z \to x, t \downarrow 0} \frac{f(z + t * d) - f(z)}{t} \tag{55}$$

We know that our function is globally and locally Lipschitz continuous and this is surely useful to prove that our loss function is f°-pseudoconvex and weakly semismooth. At the moment, our knowledge does not allow us to face an analysis to prove it. But, we think that the properties defined in section 1.4.1 can help to make it so.

2.3.5 Quadratic solver

For the quadratic problem, we decided to use Gurobi solver [20]. It uses the *Barrier Method* to solve the quadratic problem. We decided to use it because it seemed well developed, fast and properly explained, also it exposes a well written C++ API which was appropriate for our project. In the aftermath, we think that it was the right choice because we have tried some other optimizers and we believe that, after learning its library, it was quite simple to model our quadratic problem.

3 Experiments

3.1 Input data

To test the correctness of the algorithms we used the MONK's datasets. To use these datasets correctly, we did the following steps:

- We preprocessed MONK's datasets with *1-of-k* encoding to convert categorical data to numerical data and we obtained 17 binary input features vectors. This preprocessing is divided between two classes, *Preprocessing* and *LoadDataset*. The former reads, shuffle, and splits the dataset whereas the latter performs the *1-to-k* encoding.
- To view all the networks in vector formulation terms and exploit the *Armadillo* numerical library, we performed batch computation by loading and transposing the entire dataset in a single matrix. The labels were split and saved in another matrix to compute the *MSE* (sez 1.4) after the forward phase. To reduce the cost of moving matrices we took advantage of the *move* operator available since C++11.
- Our library can deal with classification and regression tasks exploiting the composition of the *Layer* class. So we implemented *sigmoid* and *linear* activation functions for the output layer and *hyperbolic tangent* activation function for the hidden layers.

To obtain a deterministic behaviour of the algorithms, we used the entire MONKS datasets as the input of the network. We obtained three matrices that had dimensions: 124x18 (Monk 1), 169x17 (Monk 2), and 122x17 (Monk 3). To compare the behaviour of the algorithms, we collected three parameters for every iteration: the error of the network (that is MSE for not regularized network and MSE plus the regularization term for regularized network), the norm of the gradient and the computational time spent on completing the iteration. These parameters were used to make the rate of convergence and the errors with respect to the minima plots shown below. For better visualization of the plots the y-axis is plotted on a logarithmic scale (except for §3.3 "Rate of convergence") and an enlargement version of each plots is put on the side or down below. Before showing the plots a table with all the configurations used and the values obtained are shown.

3.2 Configuration tested

Our goal was to compare the optimization performance of the algorithms described in §2 on the following model:

- M1 is a neural network with topology and activation function of your choice, provided it is differentiable, but mandatory L1 regularization;
- M2 is a neural network with topology and activation function of your choice, but mandatory L2 regularization.

As mentioned in §3.4, we decided to use as stop condition the norm of the gradient, the error plus the regularization term and the number of iteration. This was done to obtain a solution in a feasible amount of time after an analysis of each algorithm. The table 1 summarizes the threshold used for the stop condition of each algorithm.

Optimizer	**Iteration**	$\|\nabla f_k\|$	f^*
MDA	15000	1-e3	1e-2
NMDA	15000	1-e3	1e-2
L-BFGS	500	5e-1	1e-2

PBM	2000	5e-5	9e-2

Table 1: Threshold used for the stop condition of each algorithm.

The notation f^* specifies the optimal value reached by a specific configuration of the problem (that includes the regularization term if it is used). The notation LS specifies the usage of the Line Search method to find an appropriate step length for each iteration of the algorithms that use it.

Task	**Optimizer**	**Model**	**Iteration**	**Step length**	**Lambda**	**Mom**	f^*	$\|\nabla f_k\|$	**Time(ms)**
Monk1	MDA	M1	15000	0.9	3e-4	0.9	1.301e-2	7.855e-3	9161
Monk1	MDA	M2	15000	0.9	3e-4	0.6	2.101e-2	6.740e-2	2321
Monk1	NMDA	M1	15000	0.9	3e-4	0.9	1.143e-2	8.343e-3	14695
Monk1	NMDA	M2	15000	0.9	3e-4	0.6	1.105e-2	3.462e-2	5231
Monk1	L-BFGS	M1	32	LS	3e-4	0	7.939e-2	6.324e-6	19427
Monk1	L-BFGS	M2	500	LS	3e-4	0	4.327e-2	5.456e-1	19362
Monk1	PBM	M1	2000	LS	3e-4	0	8.796e-2	3.869e-5	1556359
Monk1	PBM	M2	54	LS	3e-4	0	8.491e-2	1.349e-0	100055
Monk2	MDA	M1	15000	0.9	3e-4	0.9	1.374e-2	1.236e-2	20422
Monk2	MDA	M2	15000	0.9	3e-4	0.6	2.264e-2	9.747e-2	3421
Monk2	NMDA	M1	15000	0.9	3e-4	0.9	1.374e-2	1.228e-2	18934
Monk2	NMDA	M2	15000	0.9	3e-4	0.6	2.251e-2	1.264e-1	6098
Monk2	L-BFGS	M1	381	LS	3e-4	0	6.845e-1	9.43e-2	27206
Monk2	L-BFGS	M2	481	LS	3e-4	0	4.542e-2	4.294e-1	26744
Monk2	PBM	M1	270	LS	3e-4	0	1.176e-1	9.973e-6	24036
Monk2	PBM	M2	2000	LS	3e-4	0	1.96e-1	6.971e-2	3654
Monk3	MDA	M1	15000	0.9	3e-4	0.9	3.458e-2	7.675e-2	22280
Monk3	MDA	M2	15000	0.9	3e-4	0.6	2.794e-2	3.502e-2	3458
Monk3	NMDA	M1	15000	0.9	3e-4	0.9	1.587e-2	7.145e-3	20003
Monk3	NMDA	M2	15000	0.9	3e-4	0.6	2.15e-2	8.247e-2	3405
Monk3	L-BFGS	M1	251	LS	3e-4	0	5.608e-2	9.905e-2	10352
Monk3	L-BFGS	M2	201	LS	3e-4	0	4.354e-2	4.038e-1	8314
Monk3	PBM	M1	1433	LS	3e-4	0	7.423e-2	4.994e-5	130621
Monk3	PBM	M2	1434	LS	3e-4	0	7.971e-2	4.992e-5	129678

Table 2: Network configurations with f^*.

3.2.1 Methods minima analysis

As mentioned in §1.4.1, the objective function is not convex. For this reason, we try to understand if the methods used were approaching the same optimal value or different ones. First of all, we analysed the f^* value of each configuration to understand at which optimal values the algorithms converge.

Task	**Optimizer**	**Model**	**Iteration**	**Step length**	**Lambda**	**Mom**	f^*
Monk1	NMDA	M1	15000	0.9	3e-4	0.9	1.143e-2
Monk1	MDA	M1	15000	0.9	3e-4	0.9	1.301e-2
Monk1	L-BFGS	M1	32	LS	3e-4	0	7.939e-2
Monk1	PBM	M1	2000	LS	3e-4	0	8.796e-2
Monk1	NMDA	M2	15000	0.9	3e-4	0.6	1.105e-2
Monk1	MDA	M2	15000	0.9	3e-4	0.6	2.101e-2

Monk1	L-BFGS	M2	500	LS	3e-4	0	4.327e-2
Monk1	PBM	M2	54	LS	3e-4	0	8.491e-2

Table 3: Monk 1 optimizer configuration divided by model and displayed in increasing order of f^*.

As can be seen from table 3, the different M1 configuration of the optimizers MDA and NMDA with 1500 iterations converge to the same optimal values. This can be stated because their optimal values differ of 1.5e-3. The other configuration have not value enough similar to state that the algorithms were converging to the same value.

Task	**Optimizer**	**Model**	**Iteration**	**Step length**	**Lambda**	**Mom**	f^*
Monk2	MDA	M1	15000	0.9	3e-4	0.9	1.374e-2
Monk2	NMDA	M1	15000	0.9	3e-4	0.9	1.374e-2
Monk2	PBM	M1	270	LS	3e-4	0	1.176e-1
Monk2	L-BFGS	M1	381	LS	3e-4	0	6.845e-1
Monk2	NMDA	M2	15000	0.9	3e-4	0.6	2.251e-2
Monk2	MDA	M2	15000	0.9	3e-4	0.6	2.264e-2
Monk2	L-BFGS	M2	481	LS	3e-4	0	4.542e-2
Monk2	PBM	M2	2000	LS	3e-4	0	1.96e-1

Table 4: Monk 2 optimizer configuration divided by model and displayed in increasing order of f^*.

As can be seen from table 4, the different configurations of M2 model of the optimizers MDA and NMDA with 15000 iterations differ of value smaller than 1e-4. Also the MDA and NMDA of model M1 obtained the same optimal values.

Task	**Optimizer**	**Model**	**Iteration**	**Step length**	**Lambda**	**Mom**	f^*
Monk3	NMDA	M1	15000	0.9	3e-4	0.9	1.587e-2
Monk3	MDA	M1	15000	0.9	3e-4	0.9	3.458e-2
Monk3	L-BFGS	M1	251	LS	3e-4	0	5.608e-2
Monk3	PBM	M1	1433	LS	3e-4	0	7.423e-2
Monk3	NMDA	M2	15000	0.9	3e-4	0.6	2.15e-2
Monk3	MDA	M2	15000	0.9	3e-4	0.6	2.794e-2
Monk3	L-BFGS	M2	201	LS	3e-4	0	4.354e-2
Monk3	PBM	M2	1434	LS	3e-4	0	7.971e-2

Table 5: Monk 3 optimizer configuration divided by model and displayed in increasing order of f^*.

As can be seen from table 5, the different configuration of model M2 of the optimizers MDA and NMDA with 15000 iterations differ of a maximum of 7e-3.

After further analysis, we observed that the weights initialization influences the optimal values reached by all the models after the optimization process. Moreover, as said previously, this value are strong dependant on the function shape and for this reason the algorithms could converge to different minimum values.

3.3 Rate of convergence

r To compute the rate of convergence for each algorithm, we consider f^* as the optimal value obtained by each of the optimization algorithm plus the regularization term and $f(x_k)$ as the optimal value obtained at the iteration k plus the regularization term. If exists a real positive constant γ such that

$$lim_{k\to\infty}\frac{|f(x_{k+1})-f^*|}{|f(x_k)-f^*|^p}=\gamma>0, \tag{56}$$

we can state that the sequence of values $f(x_k)$ produced by the algorithm has an order of convergence p. In our case, we can derive an alternative faster and rapid procedure to compute it. Considering equation 56 we can obtain the following formula:

$$p=lim_{k\to\infty}\frac{log|f(x_{k+1})-f^*|-log\gamma}{log|f(x_k)-f^*|}\approx lim_{k\to\infty}\frac{log|f(x_{k+1})-f^*|}{log|f(x_k)-f^*|}. \tag{57}$$

If $p=1$ the convergence is linear and if $p=2$ the convergence is quadratic. We can observe in figure 3 that L-BFGS reached a superlinear rate of convergence. As we expect from the analysis we made, L-BFGS reached a superlinear convergence rate in some epoch. Also for the Momentum Descent Approach, we see that the convergence rate obtained is linear as we expected until the last few iterates. Instead, for the Proximal Bundle method we had some convergences rate peaks that we did not expect. In our opinion, this can be caused by the particular loss shape of Monks dataset. It is interesting to view that all of these methods, even with a noisy dataset as Monk 3, converge with the same convergence rate as mentioned in the theory. Also, as the theory says, the PBM algorithm in the M1 model should have better convergence rate than the other algorithms that use gradient instead of subgradient. This is due to L1 regularization that introduces non-differentiability, and in most of the cases we saw this advantage (see figure 2a and figure 6a). It is interesting to see that L-BFGS continues to have a superlinear rate of convergence even in the M1 model, see figure 4a. The rate of convergence curves obtained, also with an enlargement of the right and left side for each of them, are shown below.

3.3.1 M1 comparison

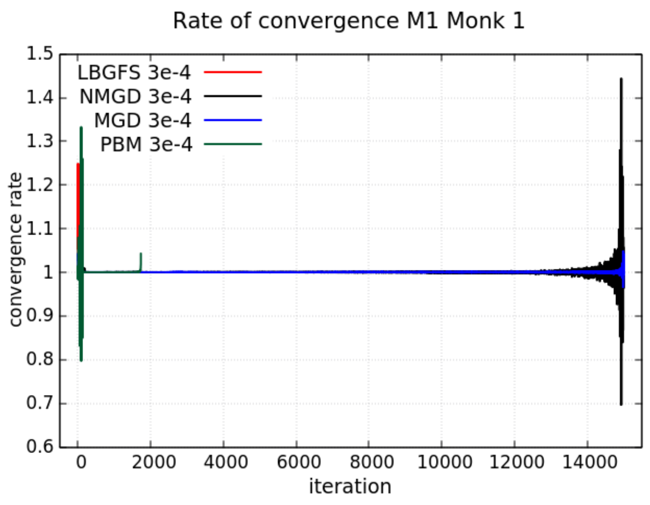
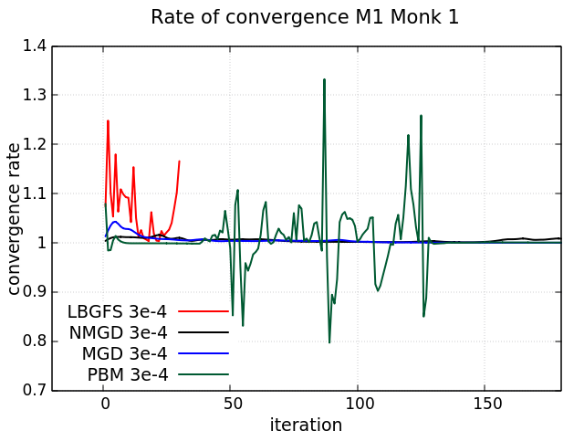
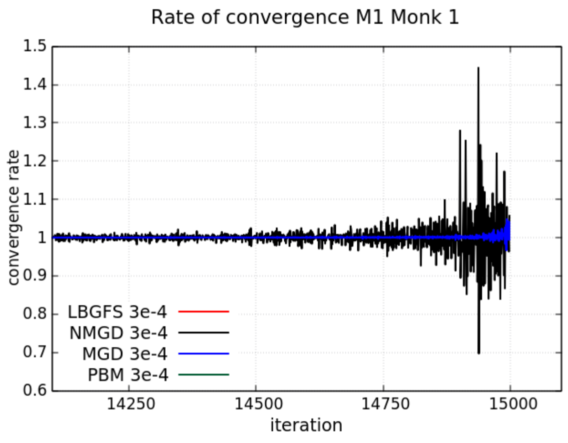


Figure 1: M1 converge rate comparison Monk 1 of configurations defined in table 2.



(a) Left zoom



(b) Right zoom

Figure 2: M1 converge rate comparison Monk 1 of configurations defined in table 2.

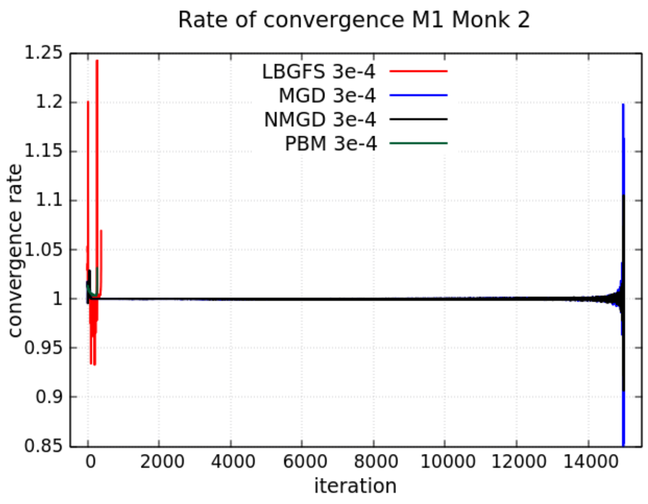
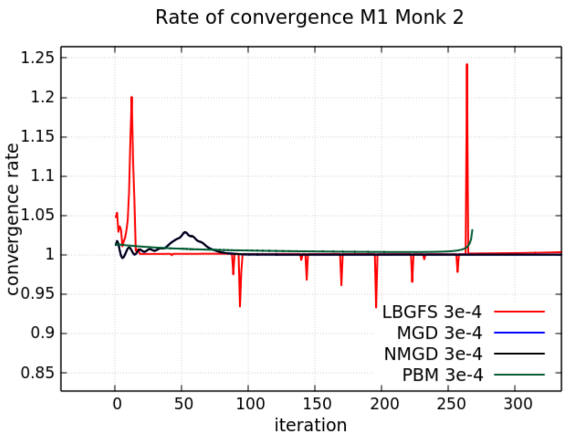
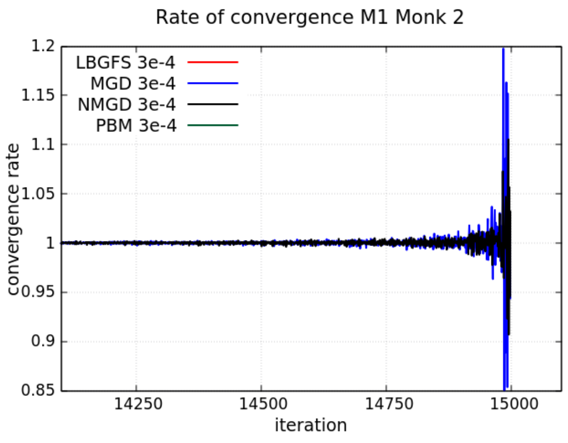


Figure 3: M1 converge rate comparison Monk 2 of configurations defined in table 2.



(a) Left zoom



(b) Right zoom

Figure 4: M1 converge rate comparison Monk 2 of configurations defined in table 2.

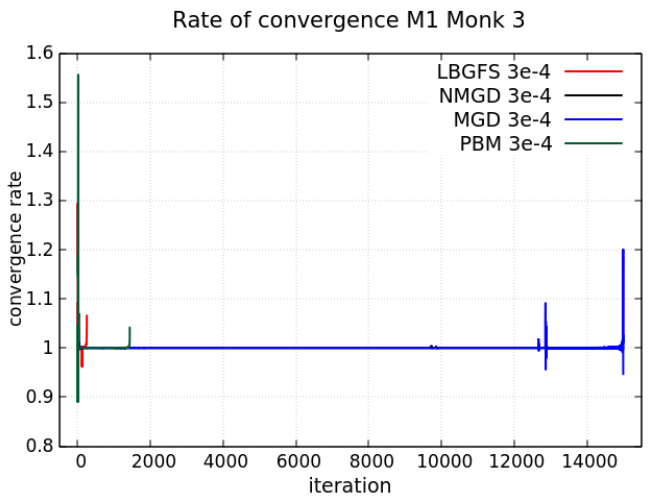
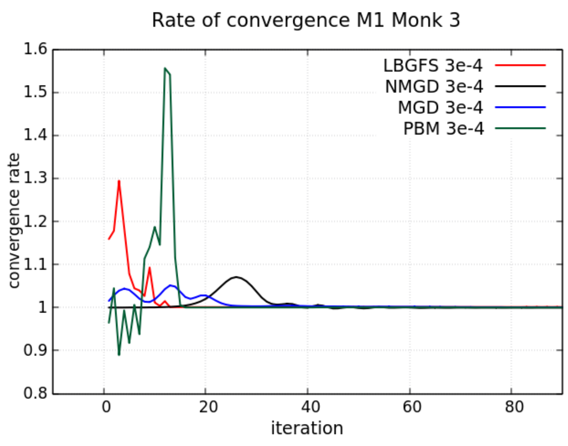
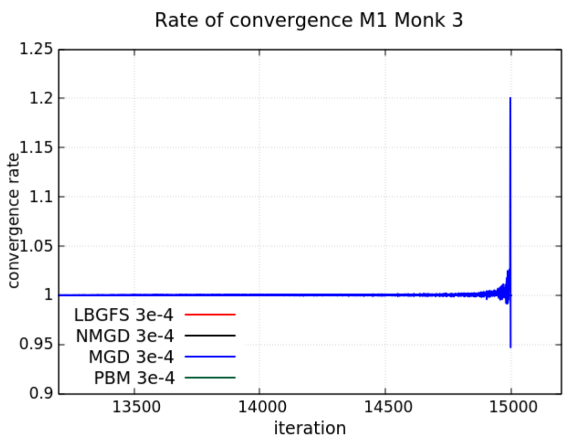


Figure 5: M1 converge rate comparison Monk 3 of configurations defined in table 2.



(a) Left zoom



(b) Right zoom

Figure 6: M1 converge rate comparison Monk 3 of configurations defined in table 2.

3.3.2 M2 comparison

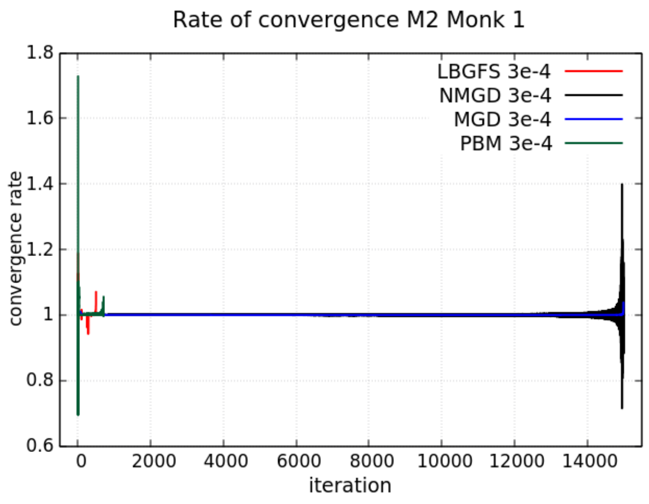
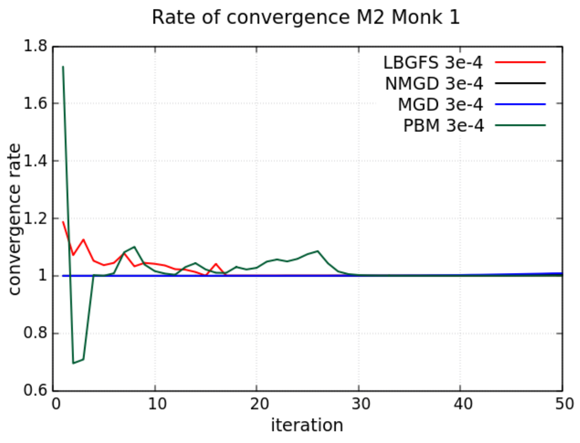
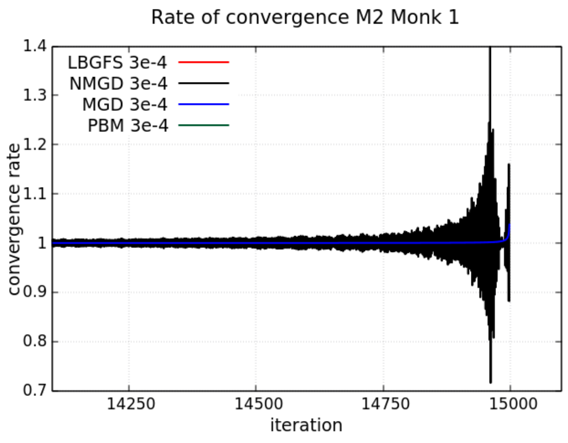


Figure 7: M2 converge rate comparison Monk 1 of configurations defined in table 2.



(a) Left zoom



(b) Right zoom

Figure 8: M2 converge rate comparison Monk 1 of configurations defined in table 2.

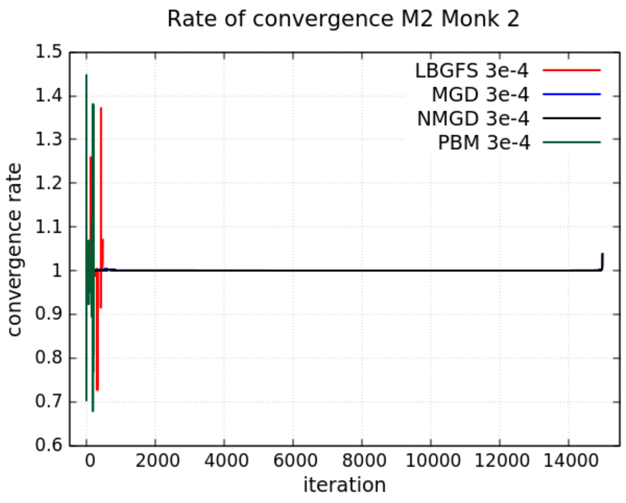
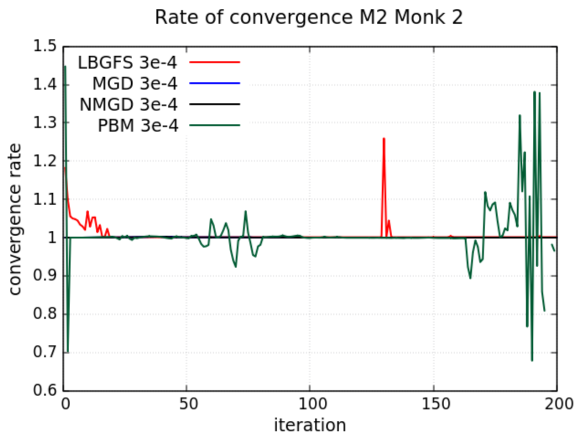
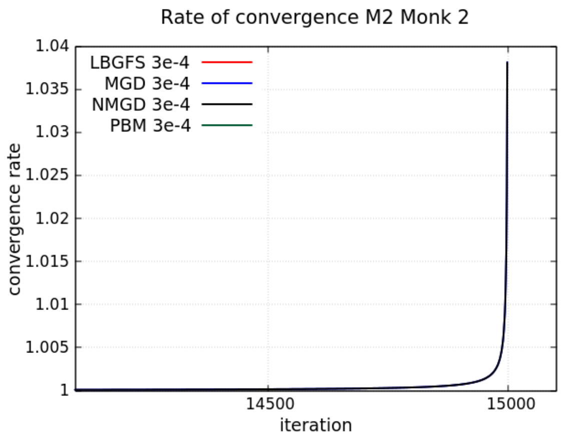


Figure 9: M2 converge rate comparison Monk 2 of configurations defined in table 2.



(a) Left zoom



(b) Right zoom

Figure 10: M2 converge rate comparison Monk 2 of configurations defined in table 2.

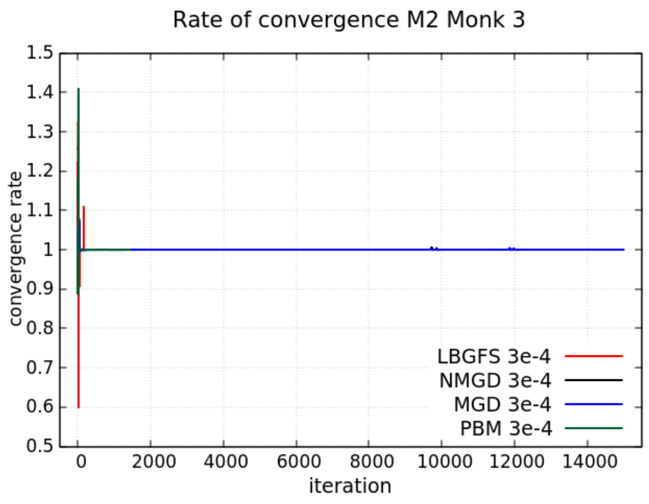
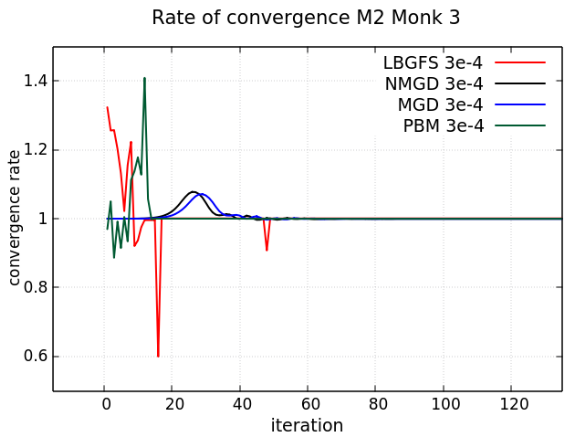
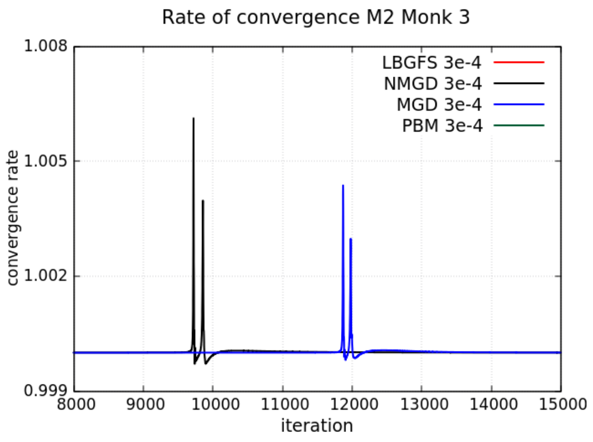


Figure 11: M2 converge rate comparison Monk 3 of configurations defined in table 2.



(a) Left zoom



(b) Right zoom

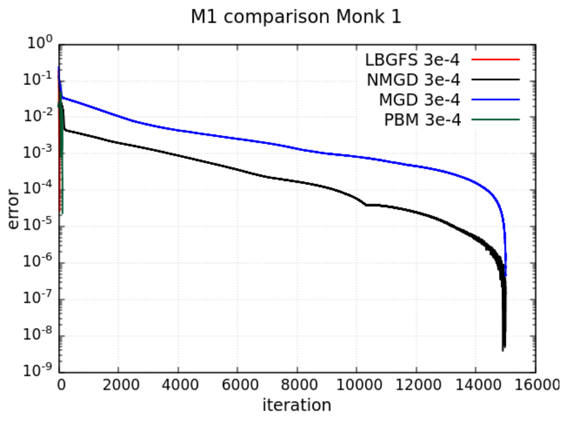
Figure 12: M2 converge rate comparison Monk 3 of configurations defined in table 2.

3.4 Errors with respect to the minima

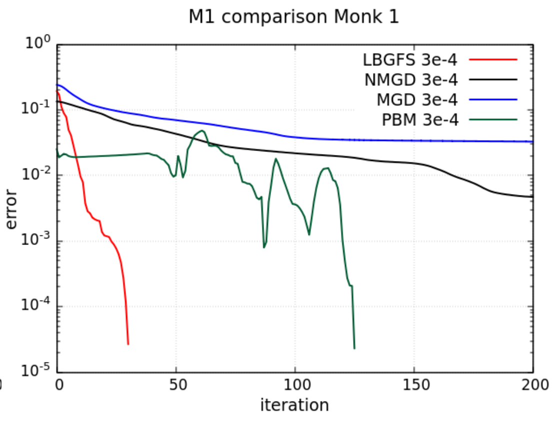
The error with respect to the minimum for each iteration is obtained by the following formula: $f(x_k) - f^*$ where $f(x_k)$ is the MSE obtained at the iteration k plus the regularization term and f^* is the optimal value reached by the specific configuration of the problem plus the regularization term. The plots of the error with respect to the minimum value obtained from the configurations in table 2 are shown below, also an enlargement for each of them is put to the side.

An important thing to notice is that f^* might not be 0 because it depends on the function that we minimize (that is the MSE plus the regularization term). This function in our case is not convex and due to this fact, it has multiple local minima in which the algorithm could stop. For this reason, to understand if an algorithm has converged to a local minimum, that cannot reach an error equal to zero, but it is still a minimum, the norm of the gradient has to be checked to understand how far from the stationary point (and how far from the end of convergence) we are. This can be seen in model M2, dataset Monk 3 with PBM optimizer with 3e-4 of regularization in which the f^* obtained is 7.971e-2 but the norm of the gradient is 4.992e-5, so the algorithm has converged near a local minimum.

3.4.1 M1 comparison

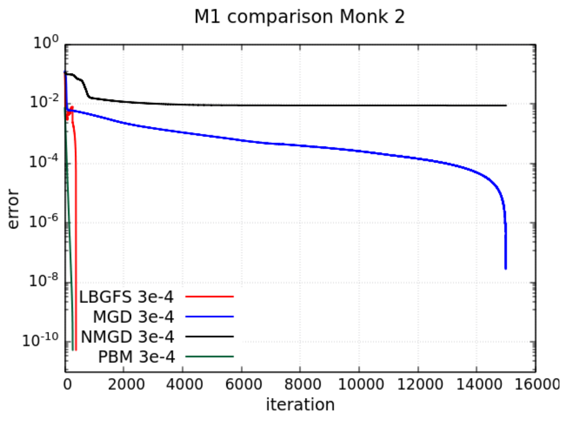


(a) Original

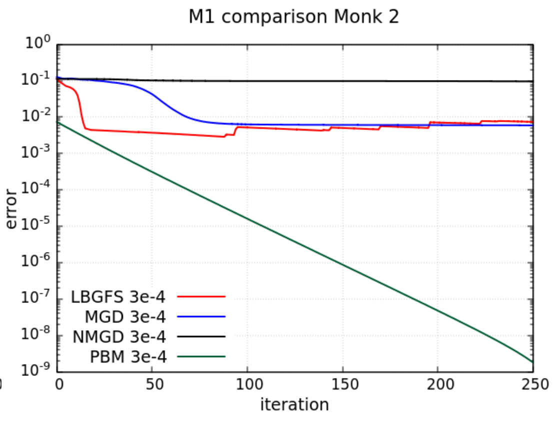


(b) Zoom

Figure 13: Errors with respect to the minimum comparison Monk 1 of Model M1 of configurations defined in table 2

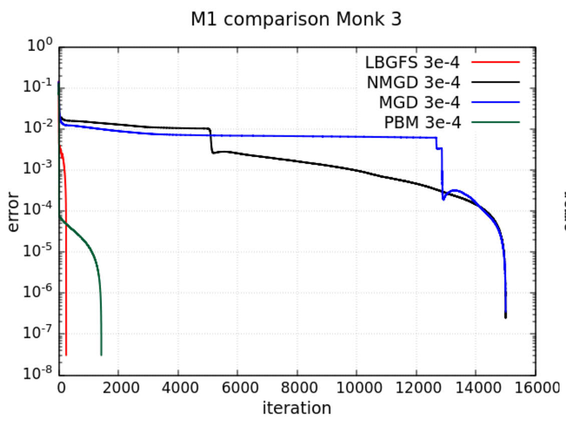


(a) Original

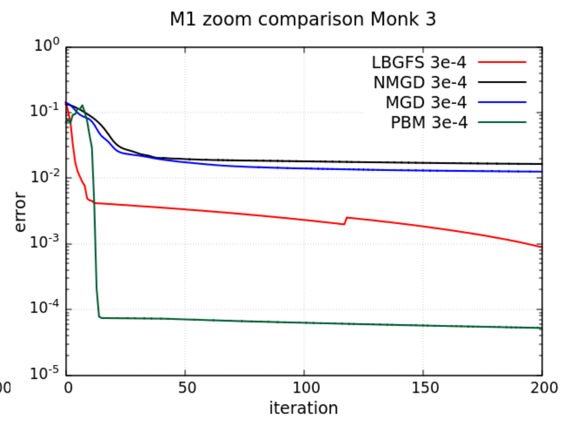


(b) Zoom

Figure 14: Errors with respect to the minimum comparison Monk 2 of Model M1 of configurations defined in table 2



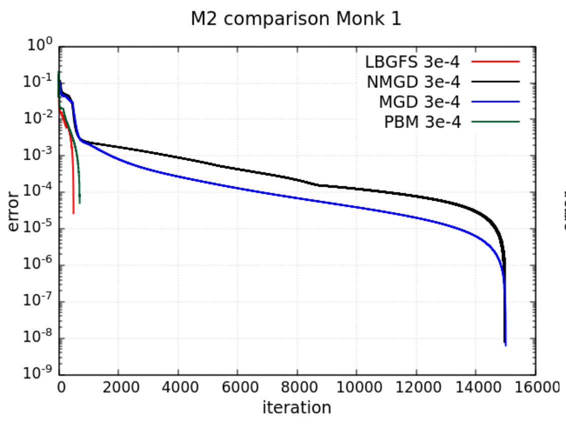
(a) Original



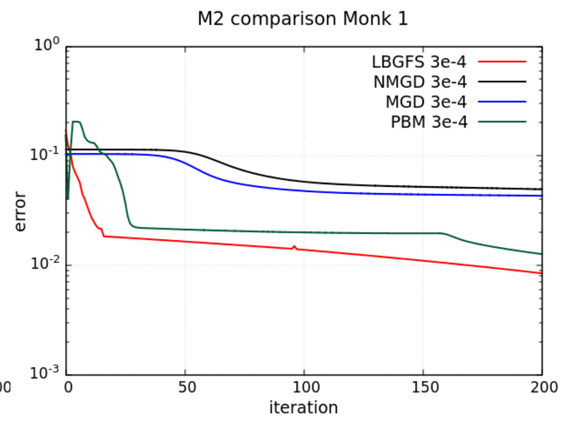
(b) Zoom

Figure 15: Errors with respect to the minimum comparison Monk 3 of Model M1 of configurations defined in table 2

3.4.2 M2 comparison

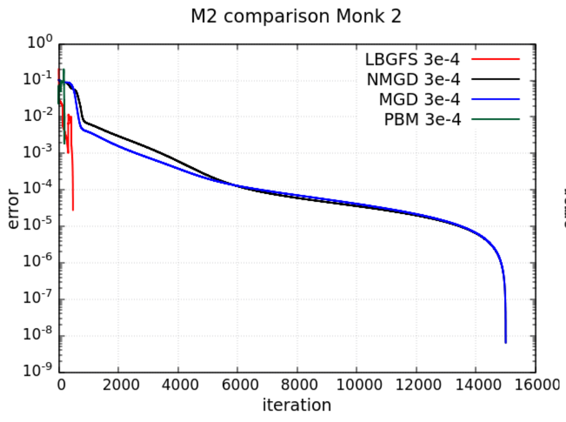


(a) Original

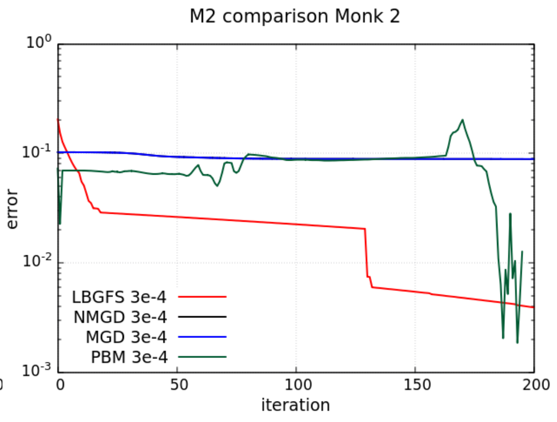


(b) Zoom

Figure 16: Errors with respect to the minimum comparison Monk 1 of Model M1 of configurations defined in table 2

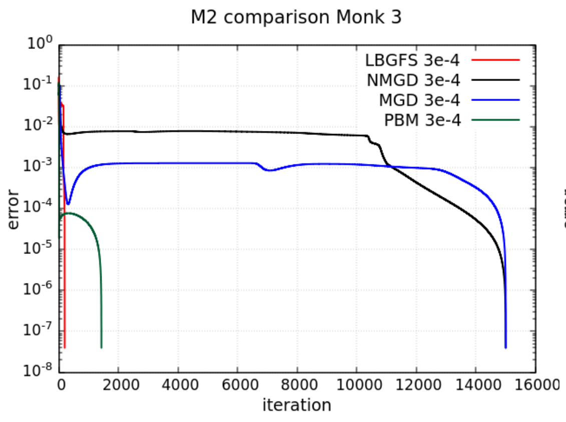


(a) Original

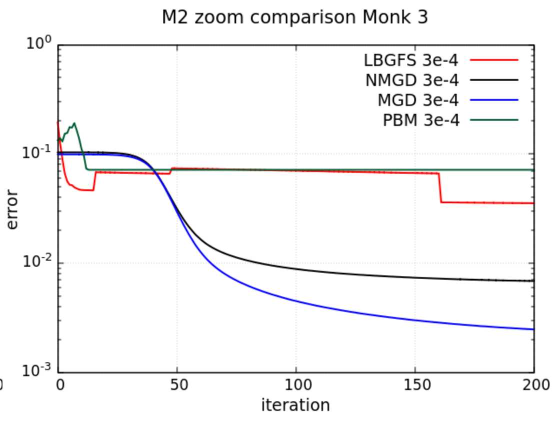


(b) Zoom

Figure 17: Errors with respect to the minimum comparison Monk 2 of Model M1 of configurations defined in table 2



(a) Original



(b) Zoom

Figure 18: Errors with respect to the minimum comparison Monk 3 of Model M1 of configurations defined in table 2

Appendices

A Repeatability

Each variant of the methods was tested with specific seed and initial interval of weight in order to make the execution repeatable. All network tested with L2 regularization were composed by 1 hidden layer of 15 units.

Task	**Method**	**Variant**	**Initialization**	**Seed**
MONK 1	MDA	M 0.6 L2 3e-4	1e-2, -1e-2	107
MONK 2	MDA	M 0.6 L2 3e-4	1e-2, -1e-2	196
MONK 3	MDA	M 0.6/0.9 L2 3e-4	1e-2, -1e-2	89
MONK 1	NMDA	M 0.6 L2 3e-4	1e-2, -1e-2	70
MONK 2	NMDA	M 0.6 L2 3e-4	1e-2, -1e-2	24
MONK 3	NMDA	M 0.6/0.9 L2 3e-4	1e-2, -1e-2	31
MONK 1	PBM	L2 3e-4	1, -1	87
MONK 2	PBM	L2 3e-4	1, -1	222 *
MONK 3	PBM	L2 3e-4	1e-2, -1e-2	33 *
MONK 1	L-BFGS	L2 3e-4	1, -1	507
MONK 2	L-BFGS	L2 3e-4	1, -1	295
MONK 3	L-BFGS	L2 3e-4	1, -1	4
MONK 1	MDA	M 0.6 L1 3e-4	1e-2, -1e-2	82
MONK 2	MDA	M 0.6 L1 3e-4	1e-2, -1e-2	7
MONK 3	MDA	M 0.6/0.9 L1 3e-4	1e-2, -1e-2	61
MONK 1	NMDA	M 0.6 L1 3e-4	1e-2, -1e-2	82
MONK 2	NMDA	M 0.6 L1 3e-4	1e-2, -1e-2	7
MONK 3	NMDA	M 0.6/0.9 L1 3e-4	1e-2, -1e-2	61
MONK 1	PBM	L1 3e-4	1e-2, -1e-2	4
MONK 2	PBM	L1 3e-4	0, 0	30
MONK 3	PBM	L1 3e-4	1e-2, -1e-2	33 *
MONK 1	L-BFGS	L1 3e-4	1, -1	86
MONK 2	L-BFGS	L1 3e-4	1, -1	295
MONK 3	L-BFGS	L1 3e-4	1, -1	4

Table 6: MONK's problems parameter.

* With "percentage_constraints_skipped" parameter set to 0.5 of percentage in the algorithm.

B Code

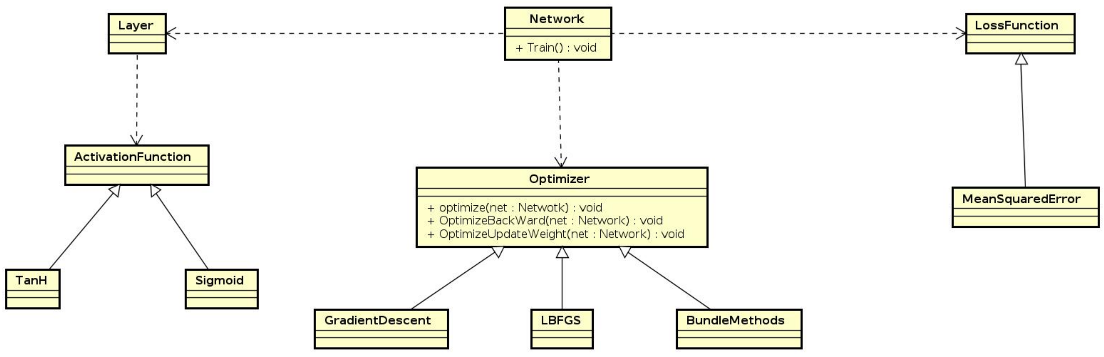


Figure 19: UML of the project.

References

[1] Jorge Nocedal Stephen J. Wright. *Numerical Optimization.* Springer, (2nd Edition, 2006).

[2] Dong C. Liu and Jorge Nocedal. *On the limited memory BFGS method for large scale optimization.* Mathematical Programming 45 (1989), pp. 503-528.
`https://people.sc.fsu.edu/~inavon/5420a/liu89limited.pdf`

[3] B.Robitaille, B.Marcos, M.Veillette and G.Payre *Modified quasi-Newton methods for training neural networks*
`https://www.sciencedirect.com/science/article/abs/pii/0098135495002286`

[4] David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams . *Learning representation by back-propagating errors.*
`https://www.nature.com/articles/323533a0`

[5] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. *Optimization methods for largescale machine learning.* [*Analyses of Stochastic Gradient Methods*]. Chapter 4. 2016.

[6] Conrad Sanderson and Ryan Curtin. *Armadillo: a template-based C++ library for linear algebra.*
`http://arma.sourceforge.net/armadillo_joss_2016.pdf`

[7] K. Eriksson, D. Estep, C. Johnson. *Applied Mathematics: Body and Soul.* Springer.

[8] Bazaraa, Sherali, Shetty. *Nonlinear programming, theory and algorithms.* Wiley-Interscience, (3rd Edition).

[9] Simon Haykin. *Neural Networks and Learning Machines.* Prentice-Hall, (3rd Edition, 2008).

[10] T. M. Mitchell. *Machine learning.* McGraw-Hill, 1997.

[11] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning.* MIT Press, 2016.

[12] D.P. Bertsekas. *Nonlinear Programming.* Second edition. Athena Scientific, 2003.

[13] A. Frangioni. *Standard Bundle Methods: Untrusted Models and Duality.*
`http://pages.di.unipi.it/frangio/abstracts.html#NDOB18`

[14] Marko M. Mäkelä, Napsu Karmitsa and Outi Wilppu. *Proximal Bundle Method for Nonsmooth and Nonconvex Multiobjective Optimization.*
`http://napsu.karmitsa.fi/publications/pbm.pdf`

[15] Brendan O'Donoghue, Emmanuel Candès . *Adaptive Restart for Accelerated Gradient Schemes.*
`https://arxiv.org/pdf/1204.3982.pdf`

[16] Andrzej Ruszczyński. *Nonlinear Optimization.* Princeton University Press.

[17] Y. Du and A. Ruszczyński. *Rate of Convergence of the Bundle Method.* Journal of Optimization Theory and Applications.
`https://arxiv.org/pdf/1609.00842.pdf`

[18] S.B. Thrun, J. Bala, E. Boloederon and I. Bratko. *The MONK's Problems.* [*A performance comparison of different learning algoritms*]. Chapter 9. Carnegie Mellon University CMU-CS-91-197, 1991.
`https://pdfs.semanticscholar.org/94c0/418c4bd9d719e1203acfd42741ebbd343073.pdf`

[19] Bagirov, Adil M., Karmitsa, Napsu, Mäkelä, Marko M. *Introduction to Nonsmooth optimization.* Springer link.
`https://www.springer.com/gp/book/9783319081137`

[20] Gurobi Optimization LLC *Gurobi Optimizer Reference Manual.*
https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf