ISSN 2281-4299

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Online Block Layer Decomposition schemes for training Deep Neural Networks

Laura Palagi
Ruggiero Seccia

Technical Report n. 6, 2019

Online Block Layer Decomposition schemes for training Deep Neural Networks

Laura Palagi, Ruggiero Seccia

Department of Computer, Control and Management Engineering, Sapienza University of Rome, Rome, Italy

Abstract

Deep Feedforward Neural Networks' (DFNNs) weights estimation relies on the solution of a very large nonconvex optimization problem that may have many local (no global) minimizers, saddle points and large plateaus. Furthermore, the time needed to find good solutions to the training problem heavily depends on both the number of samples and the number of weights (variables). In this work, we show how Block Coordinate Descent (BCD) methods can be applied to improve the performance of state-of-the-art algorithms by avoiding bad stationary points and flat regions. We first describe a batch BCD method able to effectively tackle difficulties due to the network's depth; then we further extend the algorithm proposing an *online* BCD scheme able to scale with respect to both the number of variables and the number of samples. We perform extensive numerical results on standard datasets using different deep networks, and we showed how the application of (online) BCD methods to the training phase of DFNNs permits to outperform standard batch/online algorithms leading to an improvement on both the training phase and the generalization performance of the networks.

Keywords: Deep Feedforward Neural Networks, Block coordinate decomposition, Online Optimization, Large scale optimization

Email addresses: laura.palagi@uniroma1.it (Laura Palagi), ruggiero.seccia@uniroma1.it (Ruggiero Seccia)

1. Introduction

In this paper we consider the problem of training a Deep Feedforward Neural Network (DFNN) being available a set of training data $\{x_p, y_p\}$ for $p = 1, \dots, P$. According to the Empirical Risk Minimization (ERM) principle, training a DFNN can be formulated as an unconstrained non convex optimization problem

$$\min_{w \in \mathbb{R}^n} \; f(w) = \frac{1}{P} \sum_{p=1}^{P} E_p(w) + g(w) \tag{1}$$

where w is the vector representing the connection weights in the DFNN, the first term represents the average loss, being E_p the loss on the single sample p, and the second term is a regularization term added to improve generalization properties which can also help from an optimization viewpoint by inducing a convexification of the objective function.

This problem is known to be a very hard optimization problem both because is highly *non-convex*, which implies the presence of stationary points that are not global minimizers, because it is characterized by the presence of plateaus and cliffs [12, 28] and because of the vanishing/exploding gradient issue [16] which can extremely slow down the speed of convergence of gradient-based optimization algorithms. This peculiar structure together with the large dimension n, when considering wide and deep networks, and/or the large number of samples P, when considering large training set, leads to a high computational effort for finding a solution of the optimization problem, being objective function and gradient evaluations the heaviest tasks in the optimization process.

Problem (1) has been principally tackled with two different approaches: *batch* algorithms (a.k.a. *offline* methods), which at each iteration consider the whole dataset to update the model's weights; *online* algorithms (a.k.a. *online* methods), which at each iteration consider only part of the samples in the training set, a minibatch, to update the weights of the model. The former approach can be effectively used when the number of variables n and the number of samples P in the dataset are not too large. The latter approach, by considering at each iteration only a small subset of all the available data, is more efficient when the dataset is composed of a large number of samples P. Although *online* algorithms are less prone to issues when dealing with Big data, they are still affected by the dimension n of the problem, which in DFNN can grow fast.

An effective solution to solve optimization problems where the number of variables n is very large is represented by *Block Coordinate Descent* (BCD) methods [33, 25, 1, 19], which at each iteration update only a subset of the whole variables. In contrast with *online* methods, BCD methods present the opposite behaviour: they are not heavily affected by the number of variables but they still suffer when the number of samples P becomes too large.

In this paper, we study how the layered structure of DFNNs can be leveraged to define efficient BCD methods for speeding up the optimization process of these models. In particular, we introduce two different optimization frameworks:

- First, following the work done in [18] for shallow networks, we define a general *batch* BCD method for deep networks. We analyze the impact of *backpropagation* procedure on the choice of the blocks of variables and we show how a decomposition approach can help to escape from "bad" attraction regions.
- Then, exploiting both the variable decomposition, typical of BCD methods, and the sample-wise decomposition, typical of *online methods* we embed the BCD scheme of above into a *minibatch* framework, defining a general *online* BCD framework. Numerical results suggest how this double decomposition leads to a speed up in the solution of the non-convex optimization problem (1).

This work is organized as follows. In Section 2 an overall literature review on both BCD algorithms, *online* algorithms and how these two methods have already been combined is provided. In Section 3 the mathematical notatio is deined and the optimization problem is formulated. In Section 4 and 5, respectively the *batch* BCD framework and the *online* BCD framework are illustrated. In these sections, each algorithm is followed by some considerations concerning the optimization strategy implemented and numerical results on standard test sets. In Section 6, conclusions and suggestions on further direction of investigations are provided. In the Appendix, convergence results and further numerical experiments are provided .

2. Related works

Batch BCD algorithms are effective methods for large scale optimization problems. By updating a subset of the variables at each iteration, the application of such methods for large scale problems has already been proved to be successful in many applications [29, 25, 33, 19].
Concerning the application of BCD methods for training NNs, some approaches have already been carried out. Extreme Learning Machines (ELMs) [20, 21] are algorithms where the different role played by the output weights, namely weights of the last layer, and hidden weights are considered. Hidden weights are randomly fixed, while output weights are tuned by an optimization procedure. The great advantage of ELMs is that, for suitable choices of both the loss function and the activation function of the last layer, the optimization problem turns out to be easy to solve. From the optimization point of view, ELMs do not possess convergence properties to a stationary point. In [9] and in [18] globally convergent schemes for training Shallow Radial Basis Function (RBF) NNs and MultyLayer Perceptron (MLP) were introduced, where the convexity of the objective function when optimizing with respect to the output weights was leveraged to improve the optimization process.
Concerning *online* algorithms, the first proposed methods for training deep networks were Incremental Gradient (IG) [3, 2] and Stochastic Gradient (SG) [30, 4, 6], where the main difference is on how minibatches are picked at each iteration, in an incremental deterministic order in IG or randomly in SG. The rate of convergence of these methods is usually slower than standard *batch* methods and depends on the variance

in the gradient estimations [7]. To reduce the variance in the gradient estimation and speed up the optimization process, different approaches were implemented such as *Gradient Aggregation*, where the estimation of the gradient is improved by considering the estimated gradients in the previous iterations. Methods like these are: SVRG [22], SAGA [13], RMSProp [31], ADAgrad [14] and Adam [24]. As already pointed out, these methods are really efficient when the number of data is huge but are not able to scale with respect to the number of variables that can blow up when using Deep Learning networks.
The behaviour of *online* BCD methods has already been studied in the strongly convex case where a geometric rate of convergence in expectation has been established [32] and its effectiveness has been tested in strongly convex sparse problems such as LASSO [35] or Sparse Logistic Regression [10]. Concerning the application of *online* BCD methods in training Shallow Neural Networks, in [8] a two block decomposition scheme is proposed where at each iteration the output weights are exactly optimized using the full batch while the hidden weights are updated using a minibatch strategy with an IG update.
In this paper, filling the gap left out in [18], we first discuss how to practically extend the batch BCD scheme to deep networks exploiting the nested structure of the objective function. Afterwards, we propose a general *online* BCD procedure able to speed up the training process in DFNNs by leveraging both the advantages of BCD and *online* methods.

3. Problem Definition

Training a DFNN fits in the class of supervised learning, where a a set of data $\{x_p, y_p\}_{p=1}^{P}$, with $x_p \in \mathbb{R}^d$ representing the input features and $y_p \in \mathbb{R}^m$ the corresponding label are given. A DFNN with d inputs and m outputs is represented by an acyclic oriented network consisting of neurons arranged in L layers connected in a feed-forward way. Each neuron j in a layer $\ell = 1, \dots, L$ is characterized by an activation function $g(\cdot)$, that for the sake of simplicity we assume the same for all the neurons, with the only exception of the output layer which has a linear activation function. We introduce the following notation that will be useful in the following.

- each layer $\ell = 1, \dots L$ consists of N_ℓ neurons, being $N_0 = d$ and $N_L = m$ respectively the input and the output layers;
- w_ℓ^{ji} is the weight from neuron i in layer $\ell - 1$ to neuron j in layer ℓ;
- $w_\ell \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ is the weights matrix associated to the layer $\ell \in \mathcal{L}$ (i. e $\{w_\ell^{ji}\}_{i=1,\dots,\mathbb{R}^{N_{\ell-1}}, j=1,\dots,\mathbb{R}^{N_\ell}}$);
- the index set $\mathcal{L} = \{1, ..., L\}$, where the general index ℓ is used to refer to the block of variables w_ℓ;
- we may refer to the variable w as composed by L blocks of variables, $w = (w_1, \dots, w_L)$;

- z_ℓ^i is the output of neuron i in layer ℓ with activation function $g(\cdot)$, $z_\ell^i = g(a_\ell^i)$, where $a_\ell^i = \sum_{k=1}^{N_{\ell-1}} w_{\ell-1}^{ik} z_{\ell-1}^k$

Using this notation, given the sample x_p, we have that the output $\tilde{y}_p$ of the DFNN can be written as:

$$\tilde{y}_p(w) = \tilde{y}(w; x_p) = w_L g(w_{L-1} g(w_{L-2} \dots g(w_1^T x_p))). \tag{2}$$

As loss function in (1), we can use in principle any continuously differentiable function, $E \in \mathcal{C}^1$, coupled with a ℓ_2 norm regularization term so that the overall unconstrained problem (1) is continuously differentiable. Although any loss function in $\mathcal{C}^1(\mathbb{R}^n)$ fits in out framework, the use of the MSE allows gaining properties that can be exploited from the computational point of view of decomposition methods. Indeed, in this case the optimization problem with respect to the weights of the last layer reduces to a Linear Least Square LLSQ (see Section 4.1 for further details). With these choices, the optimization problem (1) can be written as follows:

$$\min_w f(w) = \frac{1}{P} \sum_{p=1}^{P} \|\tilde{y}_p(w) - y_p\|^2 + \rho \|w\|^2 \tag{3}$$

As already said, this problem is highly non-convex with many local minima, saddle points, plateau and cliffs which yield to a very hard optimization problem (see e.g. [28] for a review of recent results on local- global minima issue in DNNs). Furthermore, the dependency on the number of samples and the dimension of the network make the problem even harder by increasing the time needed for objective function and gradient evaluations.
The driving idea in this paper aims at defining optimization frameworks which may reduce the computational burden, facilitate to escape bad attraction regions and scale with respect to both the number of samples P and the number of variables n.

4. Batch Block Coordinate Decomposition algorithm

Block Decomposition methods are iterative algorithms where at each iteration k only a suitable subset of the variables, the *working set*, is selected to be updated.
In DFNNs the layered structure of the network can be exploited to select the working set at each iteration. Indeed, the weights of each layer w_ℓ enter in the objective function in a nested way and a "natural" decomposition of w appears that can be fruitfully used. In this Section, taking steps from the algorithmic scheme proposed in [18] for shallow networks, we define an efficient *Batch* Block Decomposition scheme for the training problem of deep networks and analyze the most critical issues in its characterization. In order to show the effectiveness of the proposed choices, in Section 4.2 we provide extensive numerical results on a set of benchmark datasets to point out the role of variable decomposition when dealing with deep and wide networks.

4.1. Block Layer Decomposition (BLD) scheme

We present a decomposition scheme, called Block Layer Decomposition (BLD), where blocks of variables are directly defined by the layers of the network w_ℓ, $\ell = 1, \dots, L$. At the basis of this choice is the fact that fixing the weights of some layer may highlight nice structures of the optimization problem with respect to the other variables, the working set, which makes it easier to solve (cfr [20, 18]). Furthermore, it allows alleviating the heaviest computational task which is the gradient computation as explained in the next subsection.

In order to define the BLD algorithm, we need to introduce the Armijo Linesearch procedure, which is reported in Algorithm 1.

Algorithm 1 Armijo Linesearch

1: Given $a > 0, \gamma \in (0,1), \delta \in (0,1)$
2: Fix $\alpha = a$
3: **while** $f(w^k + \alpha d^k) > f(w^k) + \gamma\alpha\nabla f(w^k)^T d^k$ **do**
4: $\quad \alpha = \delta\alpha$
5: **end while**
6: Return α

At each iteration k the proposed *batch* BLD method selects an index $\ell^k \in \mathcal{L}$ and updates only the block of weights associated with layer ℓ^k, the subvector w_{ℓ^k}, by setting $w_{\ell^k}^{k+1} = \tilde{w}_{\ell^k}$, where $\tilde{w}_{\ell^k}$ must satisfy some conditions. The algorithm iterates until a stopping criterion, such as a maximum number of iterations or the norm of the gradient below a certain threshold, is met.

In particular, given the index $\ell^k = \ell$, in order to guarantee convergence toward stationary points, the new point w^{k+1} obtained by updating the ℓ block as $w^{k+1} = (w_1^k, ..., \tilde{w}_\ell, ..., w_L^k)$ must satisfy the following conditions [18]:

1. The objective function evaluated at the new point w^{k+1} satisfies:

$$f(w^{k+1}) \leq f(w_1^k, \dots, w_\ell - \alpha^k \nabla_{w_\ell} f(w^k), \dots, w_L^k) \tag{4}$$

   where the stepsize α^k is chosen by an Armijo Linesearch.

2. The difference between the values of the objective function in the points w^k and in the new one w^{k+1} satisfies

$$f(w^{k+1}) - f(w^k) \leq -\sigma(\|\tilde{w}_\ell - w_\ell^k\|) \tag{5}$$

   where σ is a forcing function satisfying

$$\lim_{k\to\infty} \sigma(t^k) = 0 \implies \lim_{k\to\infty} t^k = 0$$

The general Batch BLD scheme is reported in Algorithm 2.

Algorithm 2 *Batch* BLD scheme

Given $\{x_p, y_p\}_{p=1}^P, \mathcal{L} = \{1, ..., L\}$
Choose $w^0 \in \mathbb{R}^n$ and set $k = 0$;
while (stopping criterion not met) **do**
    Select an index $\ell^k \subseteq \mathcal{L}$
    Set $d_{\ell^k} = -\nabla_{w_{\ell^k}} f(w^k)$
    if $d_{\ell^k} \neq 0$ **then**
        Compute α^k along d_{ℓ^k} with an Armijo Linesearch
        Find a point $\tilde{w}_{\ell^k}$ such that (4) and (5) are satisfied
    else
        $\tilde{w}_{\ell^k} = w^k_{\ell^k}$
    end if
    Update $w^{k+1} = (w^k_1, \ldots, \tilde{w}_{\ell^k}, \ldots, w^k_L)$
    Set $k = k + 1$
end while

Algorithm 2 must be further specified by characterizing both how the index ℓ^k is chosen and how the new block $\tilde{w}_\ell$ is computed. Indeed, these choices may affect the convergence properties of the algorithm and its computational efficiency as we discuss later.
Concerning convergence to stationary non-maxima points of the Batch BLD framework, we have that it fits within the general decomposition scheme proposed in Section 7 of [19]. To prove convergence properties of the algorithm, we will consider the following assumption on the selection of the index set ℓ^k:

Assumption 4.1 (Cyclic updating rule)**.** *Blocks of variables w_ℓ must be updated in a cyclical order, namely:*

$$\forall k \qquad \cup_{i=k}^{L} \{\ell^i\} = \mathcal{L} \tag{6}$$

Note that convergence of the framework may be proved under milder assumptions such as an *essentially cyclic updating condition* [18, 33] which requires that each block is updated within a finite number of steps. Being it out of purpose with the aim of the paper, we limit to consider the cyclic rule stated above which is obviously encompassed by this milder assumption .
Under this assumption on the selection rule, the following convergence result holds whose proof can be easily derived from the convergence results in [19].

Theorem 4.2. *Algorithm BLD with a cyclical selection of the blocks generates a sequence of points $\{w^k\}$ such that*

$$\lim_{k\to\infty} \nabla f(w^k) = 0 \tag{7}$$

In the following, we discuss the possible choices that satisfy the assumptions on ℓ^k and on $\tilde{w}_{\ell^k}$ needed to guarantee convergence of BLD with a look at savings in time computations, mainly due to the reduced time needed for objective function and gradient evaluations.

Selection of the working set

Before providing general rules on how to choose the index ℓ^k, we need to recall how the *backpropagation* procedure works in order to comprehend the crucial role played by the working set definition in the BLD scheme.
In DFNNs, the main effort in computing $\nabla f(w)$ stays in evaluating $\sum_{p=1}^{P} \nabla_w E_p$. Thanks to the chain rule, indeed, each element $\nabla_w E_p$ is computed as

$$\frac{\partial E_p(w)}{\partial w_\ell^{ji}} = z_{\ell-1}^i \delta_\ell^j \tag{8}$$

where the z_ℓ are obtained by forward propagation of the input until the last layer and the δ_ℓ by backward propagation of the error $e = \tilde{y}_p - y_p$ throughout the layers

$$\delta_L^j = e g'(a_L) \qquad \delta_\ell^j = \sum_{k=1}^{N_{\ell+1}} \delta_{\ell+1}^k w_{\ell+1}^{kj} g'(a_\ell^j) \qquad \ell \leq L-1 \tag{9}$$

where g' is the derivative of the activation function g.
To get derivatives with respect to the block w_ℓ, the *backpropagation* procedure requires to forward propagate the input and evaluate the δs by backpropagating the errors from δ_L to δ_ℓ, being δ_ℓ depending on $\delta_{\ell+1}$. Then, if the full gradient $\nabla f(w^k)$ must be evaluated, as in standard gradient based methods, forward and backward propagation involves all the layers of the network. On the other hand, when only the gradient with respect to a block ℓ is needed, as in BLD scheme, the propagation step goes only over *part* of the network, reducing the computational effort.
As regard the potential strategies for choosing the index ℓ^k, any *fixed* order in which layers are selected according to a priori cyclical rule may be implemented which ensure Assumption (6), such as:

- ***Forward* selection**. Layers are selected according to the forward rule $\{1, 2, ..., L\}$.
- ***Backward* selection**. Layers are selected according to the backward rule $\{L, L-1, ..., 1\}$.

Instead of optimizing at each iteration only the variables of a layer ℓ, as done in a BLD scheme, other strategies could be implemented in order to define the set of variables which are updated at each iteration. However, in order to save computations in the evaluation of the gradient, the proposed BLD strategy turns out to be the most effective. Indeed, due to the nested structure of the output (2) and looking at (8) and (9), when a block update of w_ℓ is performed, all the z_is in the preceding layers $i = 1, \ldots, \ell-1$ remains unchanged and we need to forward propagate the input only from layer ℓ to L and backpropagate the errors from δ_L up to only δ_ℓ. Hence, a selection rule that considers blocks with weights belonging to the same layers allows saving some of the computations needed, while strategies which consider at each iteration weights of different layers would imply a loss in efficiency during both the forward and the backward steps.

We highlight also that a selection of blocks of variables using weights entering each single neurons of a layer, as done in [18] for shallow networks, turns out to be inefficient, since to update the general neuron j at layer ℓ we need to compute δ_ℓ which depends on all the δs of the next layers which would be computed but not further used, leading to a waste of calculi.
A different block selection rule that may be used, consists in updating at iteration k the weights of all the layers from ℓ^k to L, being these δs already computed to evaluate δ_ℓ. This blocks selection rule would be more the values of all the δs already computed to evaluate δ_ℓ. Even if this blocks selection rule would be more efficient since it would imply the utilization of all the δs computed at each iteration, however, this procedure leads to an unbalanced block update because the last layers are optimized more often than the first ones. Extensive numerical results (not reported here for the lack of space) shows a worse performance as the final value of the objective function.

Update of the block $\tilde{w}_\ell$
Once layer ℓ^k is selected, a point $\tilde{w}_{\ell^k}$ satisfying conditions (4) (5) must be determined. First we note that, similarly to Extreme Learning Machine [20, 21], when the last block $\ell^k = L$ is selected, the optimization problem (3) reduces to a strictly convex linear least square problem (LLSQ). As a consequence, the update $\tilde{w}_L$ satisfying conditions 4, 5 can be found (see [19]) by solving the subproblem to global solution $\tilde{w}_L^{k+1} = \arg\min_{w_L} f(w)$ either in closed form or by applying an iterative method such as the Conjugate Gradient algorithm [27]. When a block different from the last one is selected, $\ell \neq L$, the optimization problem becomes non-convex so that global solution cannot be pursued and finding good updates $\tilde{w}_\ell$ may becomes harder.
We highlight that the point returned by the Armijo Linesearch along the steepest descent direction

$$\tilde{w}_{\ell^k} = w_{\ell^k}^k - \alpha^k \nabla_{w_{\ell^k}} f(w^k)$$

satisfies the two conditions (4) (5), but a single iteration of the gradient method for each ℓ^k can slow down the overall procedure. However, it is well known [5] that a finite number of steps of a gradient method with Armijo Linesearch guarantees conditions (4) (5), so that $\tilde{w}_{\ell^k}$ can also be obtained by iterating a gradient method with an Armijo Linesearch. Moreover, since the number of variables in DNNs may be very high, methods which approximate second-order information and do not suffer too much of the *curse of dimensionality* would perform remarkably better than gradient methods. So, at each iteration an optimization method such as a limited memory Quasi-Newton method [27] like LBFGS can be applied to solve the subproblem in the variables w_{ℓ^k} to get a trial point $\tilde{w}_{\ell^k}$. Conditions (4) and (5) are checked and if satisfied the point is accepted; otherwise $\tilde{w}_{\ell^k}$ is obtained by an Armijo Linesearch along the steepest descent direction.

4.2. Implementation and performance of a Backward Block Layer Decomposition (B^2LD) method

For the experimental results, we implemented a BLD algorithm with the following settings:

- a *backward cyclic order* selection rule of the layer;

- updated point $\tilde{w}_\ell^k$ obtained by means of a LBFGS method with increasing accuracy ε^k and maximum number of iterations.

Algorithm 3 Backward Block Layer Decomposition scheme (B^2LD)

Given $\{x_p, y_p\}_{p=1}^P$
Choose $w^0 \in \mathbb{R}^n$ and set $k = 0$; set $\epsilon^0 > 0, \delta \in (0, 1)$.
while (stopping criterion not met) **do**
    for $l = L, ..., 1$ **do**
        Find $\tilde{w}_\ell$ s.t. $\|\nabla_{w_\ell} f(\tilde{w}_\ell)\| \leq \epsilon^k$
        if $\tilde{w}_\ell$ satisfies (4) (5) **then**
            $w_\ell^{k+1} = \tilde{w}_\ell$
        else
            $w_\ell^{k+1} = w_\ell^k - \alpha^k \nabla_{w_\ell} f(w^k)$ with α^k obtained by an Armijo Linesearch
        end if
    end for
    $\epsilon^{k+1} = \epsilon^k \delta$
    $k = k + 1$
end while

We remark that we used the LBFGS method also to solve the LLSQ problem in the variables of the last block w_L. This choice is due to the observation in the numerical results that forcing a high accuracy in the optimization of the last layer from the early stages seemed to get worse results in terms of quality of the solution found.
We use B^2LD for the solution of problem (3) with the regularization parameter $\rho = 10^{-3}/n$, a sigmoid function as activation function for the hidden neurons and a linear function for the output neurons. We compared B^2LD performance against an LBFGS method applied to the whole optimization problem (3).
The initial point w^0 has been set to the same random value for both the two methods. The following stopping criteria were set: i) $\|\nabla f(w^k)\| \leq 10^{-3}$; ii) $f(w^k) - f(w^{k+1}) \leq 10^{-4}(\max\{f(w^k), 1\})$; iii) computational time limit fixed at 150 seconds.
The two algorithms have been compared over five different network architectures in order to analyze how the algorithms perform when dimensions of the problems increase with respect to two different parameters: width N^ℓ and depth L of the network. For the sake of simplicity in most cases we assume equal numbers of neurons per layer $N_\ell = N$, so that each network is identified by the numbers of hidden layers L and neurons per layer N using the notation $[L \times N]$, so that we consider $[1 \times 50], [3 \times 20], [5 \times 50], [10 \times 50]$ and the three layer network $[200, 50, 200]$, where the values represent the number of neurons per layer.
Comparison were carried over seven different datasets publicly available at `https://sci2s.ugr.es/keel/index.php` and `https://archive.ics.uci.edu/ml/index.php`. The number of samples in the training and test sets and the number of

features per dataset are reported in Table 1 along with the number of variables for each optimization problem.

Table 1: Description of the dataset and corresponding number of variables for each network configuration

				# Variables in the network				
Dataset	# Train	# Test	# Features	$[1 \times 50]$	$[3 \times 20]$	$[200, 50, 200]$	$[5 \times 50]$	$[10 \times 50]$
Ailerons	11000	2750	41	2100	1640	28400	12100	24600
Bank Marketing	36168	9042	40	2050	1620	28200	12050	24550
Bejing PM 2.5	33406	8351	48	2450	1780	29800	12450	24950
Bikes Sharing	13903	3476	59	3000	2000	32000	13000	25500
California	16512	4128	9	500	1000	22000	10500	23000
CCPP [3]	7654	1913	5	300	920	21200	10300	22800
Mv	32614	8154	13	700	1080	22800	10700	23200

Both the two algorithms B^{2}LD and LBFGS were implemented in Python version 3.6, using the package *Scipy* [23] for the optimization algorithms; numerical tests are carried on a Intel(R) Core(TM) i7-870 CPU 2.93 GHz.

Performances are compared on the results obtained on 10 multistart runs. We consider both the best and the averaged performance over the 10 runs (see Tables 2, 3 and the behaviour of the test error during iterations when applying the two different algorithms (see Figure (1)).
In Table 2 the detailed results corresponding to the best run of each of the two algorithms are reported. For each network architecture and for each problem we report: a) the best solution found, b) the corresponding norm of the gradient and c) the CPU time needed to find it. We highlighted in boldface the best results. Note that for technical reasons due to the Scipy interface, the computational times were checked only every 30 iterations of LBFGS method that is why in Table 2 the values in the CPU Times column can be higher than the limit of 150 seconds.
Looking at Table 2, the advantages of using B^{2}LD does not seem so evident for small networks, e.g. $[1 \times 50]$ and $[3 \times 20]$, where the performance of the algorithms are quite similar; however, its ability in exploiting the layer decomposition of the network becomes evident when considering deeper and wider problems, e.g. $[200, 50, 200]$, $[5 \times 50]$ and $[10 \times 50]$, where B^{2}LD beats LBFGS on the values of the objective function.
Furthermore, we observe on the deepest network $[10 \times 50]$ that in most of the problems, LBFGS seems to be trapped in a local solution or to suffer of small gradient values. Indeed the value of the error returned by LBFGS is significantly larger than the one of B^{2}LD (10^{-1} versus 10^{-3}) but the corresponding norm of the gradient is much smaller than in B^{2}LD so that LBFGS stops earlier. There are several possible explanations for this behaviour. Firstly, standard descent optimization methods, by following descent direction for the whole set of variables, may be naturally attracted toward close stationary points, while decomposition methods, by splitting variables and thus moving along different routes, may be able to escape from such poor regions [28, 18]. Similar ideas are also at the basis of the use of greedy pre-training algorithms for Deep networks (see

Table 2: Best result returned over 10 runs by the B^{2}LD and LBFGS methods

Network	Data Set	Objective Function		Gradient Norm		Cpu Time	
		B^{2}LD	LBFGS	B^{2}LD	LBFGS	B^{2}LD	LBFGS
[1 x 50]	Ailerons	2.37×10^{-3}	$\mathbf{2.32}\times10^{-3}$	9.98×10^{-4}	$\mathbf{4.03}\times10^{-4}$	**0.18**	3.04
	Bank Marketing	7.92×10^{-5}	$\mathbf{3.67}\times10^{-5}$	$\mathbf{4.96}\times10^{-4}$	6.06×10^{-4}	**13.91**	29.77
	Bejing Pm25	$\mathbf{4.79}\times10^{-3}$	5.32×10^{-3}	$\mathbf{8.57}\times10^{-4}$	2.25×10^{-3}	9.16	**8.43**
	Bikes Sharing	$\mathbf{2.04}\times10^{-3}$	3.01×10^{-3}	$\mathbf{7.12}\times10^{-4}$	4.10×10^{-3}	16.44	**11.58**
	California	2.00×10^{-2}	2.00×10^{-2}	$\mathbf{2.98}\times10^{-4}$	3.34×10^{-3}	**1.50**	2.99
	CCPP	$\mathbf{3.35}\times10^{-3}$	3.53×10^{-3}	$\mathbf{5.88}\times10^{-4}$	6.87×10^{-4}	**0.08**	0.91
	Mv	2.93×10^{-4}	$\mathbf{2.68}\times10^{-4}$	7.18×10^{-4}	$\mathbf{4.90}\times10^{-4}$	**20.90**	23.78
[3 x 20]	Ailerons	2.33×10^{-3}	$\mathbf{2.24}\times10^{-3}$	9.29×10^{-4}	$\mathbf{4.88}\times10^{-4}$	**3.50**	3.62
	Bank Marketing	1.09×10^{-4}	$\mathbf{3.91}\times10^{-5}$	8.09×10^{-4}	$\mathbf{7.66}\times10^{-4}$	59.19	**17.45**
	Bejing Pm25	$\mathbf{4.45}\times10^{-3}$	7.60×10^{-3}	$\mathbf{9.48}\times10^{-4}$	1.36×10^{-3}	6.24	**6.04**
	Bikes Sharing	$\mathbf{1.88}\times10^{-3}$	2.81×10^{-3}	$\mathbf{8.37}\times10^{-4}$	2.36×10^{-3}	**10.76**	16.54
	California	$\mathbf{2.02}\times10^{-2}$	2.25×10^{-2}	$\mathbf{4.10}\times10^{-4}$	2.28×10^{-3}	2.44	**1.56**
	CCPP	3.47×10^{-3}	$\mathbf{3.35}\times10^{-3}$	3.89×10^{-4}	$\mathbf{3.38}\times10^{-4}$	1.51	**1.17**
	Mv	2.92×10^{-4}	$\mathbf{2.17}\times10^{-4}$	7.71×10^{-4}	$\mathbf{3.70}\times10^{-4}$	**23.86**	37.44
[200,50,200]	Ailerons	$\mathbf{2.33}\times10^{-3}$	2.56×10^{-3}	$\mathbf{9.09}\times10^{-4}$	1.58×10^{-3}	**17.93**	36.77
	Bank Marketing	$\mathbf{1.93}\times10^{-4}$	2.31×10^{-4}	$\mathbf{1.61}\times10^{-3}$	4.98×10^{-3}	184.34	**151.09**
	Bejing Pm25	$\mathbf{4.59}\times10^{-3}$	5.76×10^{-3}	$\mathbf{8.44}\times10^{-4}$	2.59×10^{-3}	137.09	**113.77**
	Bikes Sharing	$\mathbf{2.01}\times10^{-3}$	4.00×10^{-3}	$\mathbf{5.81}\times10^{-4}$	3.12×10^{-3}	146.89	**132.06**
	California	$\mathbf{1.98}\times10^{-2}$	2.31×10^{-2}	1.14×10^{-2}	$\mathbf{5.46}\times10^{-3}$	17.37	**15.41**
	CCPP	$\mathbf{3.50}\times10^{-3}$	3.70×10^{-3}	$\mathbf{7.04}\times10^{-4}$	8.17×10^{-4}	**6.67**	16.39
	Mv	$\mathbf{2.49}\times10^{-4}$	9.73×10^{-4}	$\mathbf{7.84}\times10^{-4}$	5.59×10^{-3}	**145.32**	185.38
[5 x 50]	Ailerons	$\mathbf{2.37}\times10^{-3}$	4.89×10^{-3}	$\mathbf{8.15}\times10^{-4}$	5.49×10^{-3}	12.57	**10.52**
	Bank Marketing	2.02×10^{-4}	$\mathbf{9.19}\times10^{-5}$	$\mathbf{9.41}\times10^{-4}$	1.43×10^{-3}	**110.51**	152.88
	Bejing Pm25	$\mathbf{5.03}\times10^{-3}$	7.26×10^{-3}	$\mathbf{5.38}\times10^{-4}$	1.34×10^{-3}	42.72	**27.26**
	Bikes Sharing	$\mathbf{2.07}\times10^{-3}$	3.46×10^{-2}	$\mathbf{5.70}\times10^{-4}$	5.06×10^{-3}	74.30	**1.81**
	California	$\mathbf{2.05}\times10^{-2}$	5.47×10^{-2}	2.92×10^{-3}	$\mathbf{1.57}\times10^{-3}$	12.56	**1.46**
	CCPP	$\mathbf{3.79}\times10^{-3}$	4.45×10^{-3}	$\mathbf{4.54}\times10^{-4}$	1.54×10^{-3}	**7.33**	8.61
	Mv	$\mathbf{3.59}\times10^{-4}$	7.76×10^{-4}	$\mathbf{6.12}\times10^{-4}$	8.38×10^{-3}	**131.44**	150.55
[10 x 50]	Ailerons	$\mathbf{2.65}\times10^{-3}$	1.31×10^{-2}	2.92×10^{-2}	$\mathbf{4.11}\times10^{-5}$	68.38	**1.95**
	Bank Marketing	$\mathbf{2.10}\times10^{-3}$	6.28×10^{-2}	1.94×10^{-1}	$\mathbf{7.93}\times10^{-5}$	188.25	**6.77**
	Bejing Pm25	$\mathbf{8.59}\times10^{-3}$	8.74×10^{-3}	8.18×10^{-4}	$\mathbf{1.77}\times10^{-5}$	**0.21**	5.08
	Bikes Sharing	$\mathbf{2.68}\times10^{-3}$	3.51×10^{-2}	3.34×10^{-2}	$\mathbf{6.89}\times10^{-5}$	158.46	**2.65**
	California	$\mathbf{1.88}\times10^{-2}$	5.50×10^{-2}	1.31×10^{-1}	$\mathbf{1.35}\times10^{-4}$	75.67	**2.65**
	CCPP	$\mathbf{4.89}\times10^{-3}$	5.16×10^{-2}	1.61×10^{-2}	$\mathbf{1.12}\times10^{-4}$	17.05	**1.16**
	Mv	$\mathbf{3.20}\times10^{-3}$	5.53×10^{-2}	1.76×10^{-1}	$\mathbf{1.16}\times10^{-4}$	167.36	**4.85**

[17] and reference therein), where the original complex optimization problem is split into simpler tasks.

Another analysis regards the dynamic behaviour of the objective function during the iterations. Since plotting the results obtained for all the experiments was unfeasible, in Figure 1, we plot the objective function values over the running time up to 30 seconds for a selection of dataset and architectures which are representative of the most common situation encountered. Similarly to the Extreme Learning machines [21], in most of the cases plots highlight a sharper decrease of the values obtained by B^{2}LD with respect to LBFGS. This aspect may be useful when implementing early stopping rules which is a common strategy used for preventing overfitting, because it may allow stopping earlier the optimization procedure. Furthermore, in some cases (e.g. in the figure Bank Marketing-[5 x 50] and Bikes Sharing-[10 x 50]), it shows how LBFGS may end up in some bad regions where the error decreases much slower than the regions explored by B^{2}LD.

Since from the machine learning perspective, finding a global solution of problem (3) may not lead to better performance on generalization error on unseen samples, we also analyze the performance of the two algorithms on the test set. In particular, we carried out an average performance analysis in order to assess whether the means of the values of the test error obtained by the two algorithms are statistically different or not. The average performance of the two algorithms LBFGS and B^{2}LD over the 10 multistart runs for each dataset and network structure is evaluated by means of a standard two-tailed T-test [15]. The results of the statistical analysis are reported in Table 3 where we report a) the mean of the test error found by the two algorithms and b) the corresponding p-value. Whenever the p-value is below a value $p_{\min}$, this ensures that with a probability of $(1 - p_{\min})$ the two means are statistically different and so we can conclude which is the best algorithm. As threshold value, we considered $p_{\min} = 5 \times 10^{-2}$. In 28 out of the 35 experiments, B^{2}LD turns out to find solutions that, with a probability of 95%, are statistically better than the solutions found by LBFGS (these are highlighted in boldface in Table 3).

Finally, we also report how many times the weights of each layer are updated when using B^{2}LD. Indeed, while LBFGS method updates all the weights of the network at the same time, B^{2}LD, by considering layers one by one, focuses only on the more worthwhile layers. Indeed, whenever the gradient with respect to w_ℓ and/or the relative reduction of the objective function in two subsequent iterations are below a given tolerance, B^{2}LD skips the optimization of the block ℓ avoiding wastes of time in optimizing parts of the network that will not lead significant improvement. To this aim, we consider the deepest network [10 x 50], and for each dataset we report in Table 4 the number of updates performed by B^{2}LD for each weight layer $\ell = 0, \ldots, 10$ being the zero layer the input one. Overall, B^{2}LD is not affected by the increased depth of the network but is able to focus only on some layers which interestingly turn out to be the first and last layers. This behaviour highlights the fact that the network might be too deep for the problems into account and a smaller number of hidden layers may be enough. We also note that on the dataset Bejing PM 2.5, B^{2}LD performs only the optimization of the last layer and it finds a better solution than LBFGS (see Table 2).

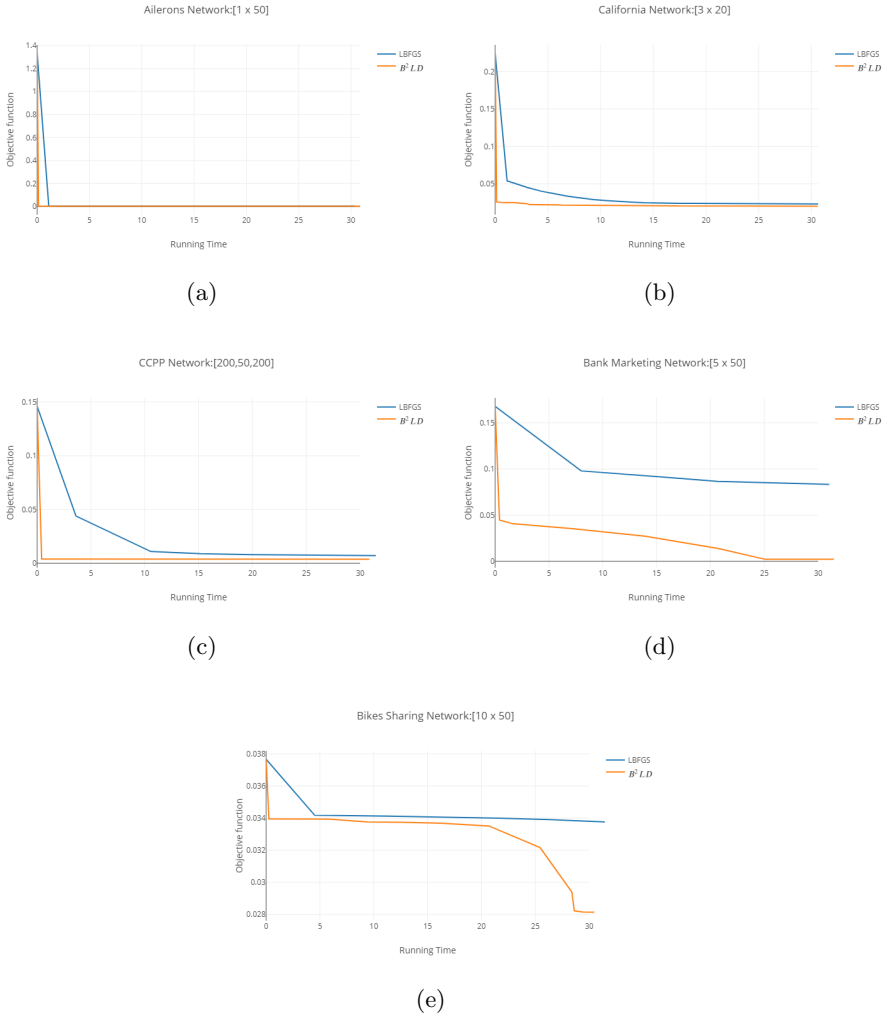

Figure 1: Comparison of the objective function behaviour during the first 30 seconds of optimization of both LBFGS and B^{2}LD.

Table 3: Means of the test error returned by the two algorithms over the 10 multistart runs and the p-value of a two-tailed T-test.

Network	Dataset	Mean B^{2}LD	Mean LBFGS	p-value
[1 x 50]	Ailerons	$2{,}40\times10^{-3}$	$2{,}43\times10^{-3}$	$2{,}22\times10^{-1}$
	Bank Marketing	$4{,}45\times10^{-4}$	$\mathbf{2{,}60}\times10^{-4}$	$2{,}50\times10^{-3}$
	Bejing Pm25	$\mathbf{4{,}77}\times10^{-3}$	$5{,}05\times10^{-3}$	$5{,}46\times10^{-5}$
	Bikes Sharing	$\mathbf{2{,}71}\times10^{-3}$	$8{,}51\times10^{-3}$	$3{,}80\times10^{-5}$
	California	$\mathbf{2{,}04}\times10^{-2}$	$2{,}22\times10^{-2}$	$8{,}58\times10^{-5}$
	CCPP	$3{,}54\times10^{-3}$	$3{,}55\times10^{-3}$	$8{,}41\times10^{-1}$
	Mv	$3{,}58\times10^{-4}$	$3{,}59\times10^{-4}$	$9{,}69\times10^{-1}$
[3 x 20]	Ailerons	$\mathbf{2{,}88}\times10^{-3}$	$7{,}52\times10^{-3}$	$1{,}08\times10^{-2}$
	Bank Marketing	$\mathbf{6{,}56}\times10^{-4}$	$1{,}39\times10^{-1}$	$7{,}84\times10^{-3}$
	Bejing Pm25	$\mathbf{4{,}37}\times10^{-3}$	$8{,}12\times10^{-3}$	$1{,}31\times10^{-16}$
	Bikes Sharing	$2{,}70\times10^{-3}$	$1{,}06\times10^{-2}$	$5{,}34\times10^{-2}$
	California	$\mathbf{2{,}28}\times10^{-2}$	$4{,}11\times10^{-2}$	$4{,}70\times10^{-3}$
	CCPP	$4{,}72\times10^{-3}$	$8{,}36\times10^{-3}$	$4{,}51\times10^{-1}$
	Mv	$8{,}41\times10^{-4}$	$6{,}62\times10^{-4}$	$3{,}38\times10^{-1}$
[200,50,200]	Ailerons	$\mathbf{3{,}08}\times10^{-3}$	$3{,}41\times10^{-3}$	$4{,}24\times10^{-4}$
	Bank Marketing	$1{,}05\times10^{-3}$	$9{,}85\times10^{-4}$	$7{,}50\times10^{-1}$
	Bejing Pm25	$\mathbf{4{,}75}\times10^{-3}$	$6{,}94\times10^{-3}$	$1{,}33\times10^{-7}$
	Bikes Sharing	$\mathbf{3{,}03}\times10^{-3}$	$1{,}04\times10^{-2}$	$3{,}93\times10^{-9}$
	California	$\mathbf{2{,}09}\times10^{-2}$	$3{,}14\times10^{-2}$	$1{,}81\times10^{-2}$
	CCPP	$\mathbf{4{,}23}\times10^{-3}$	$4{,}78\times10^{-3}$	$1{,}42\times10^{-4}$
	Mv	$\mathbf{5{,}71}\times10^{-4}$	$2{,}59\times10^{-3}$	$9{,}30\times10^{-6}$
[5 x 50]	Ailerons	$\mathbf{3{,}53}\times10^{-3}$	$1{,}06\times10^{-2}$	$3{,}99\times10^{-6}$
	Bank Marketing	$\mathbf{2{,}30}\times10^{-3}$	$2{,}23\times10^{-1}$	$1{,}24\times10^{-5}$
	Bejing Pm25	$\mathbf{4{,}88}\times10^{-3}$	$8{,}09\times10^{-3}$	$7{,}01\times10^{-15}$
	Bikes Sharing	$\mathbf{3{,}19}\times10^{-3}$	$3{,}33\times10^{-2}$	$3{,}45\times10^{-34}$
	California	$\mathbf{2{,}59}\times10^{-2}$	$5{,}84\times10^{-2}$	$8{,}42\times10^{-22}$
	CCPP	$\mathbf{6{,}24}\times10^{-3}$	$4{,}60\times10^{-2}$	$7{,}79\times10^{-8}$
	Mv	$\mathbf{1{,}28}\times10^{-3}$	$3{,}95\times10^{-2}$	$2{,}16\times10^{-4}$
[10 x 50]	Ailerons	$\mathbf{5{,}78}\times10^{-3}$	$1{,}31\times10^{-2}$	$7{,}18\times10^{-8}$
	Bank Marketing	$\mathbf{1{,}07}\times10^{-2}$	$2{,}78\times10^{-1}$	$5{,}51\times10^{-35}$
	Bejing Pm25	$8{,}47\times10^{-3}$	$8{,}40\times10^{-3}$	$5{,}87\times10^{-7}$
	Bikes Sharing	$\mathbf{5{,}12}\times10^{-3}$	$3{,}36\times10^{-2}$	$1{,}03\times10^{-26}$
	California	$\mathbf{2{,}54}\times10^{-2}$	$5{,}86\times10^{-2}$	$1{,}01\times10^{-19}$
	CCPP	$\mathbf{1{,}36}\times10^{-2}$	$5{,}11\times10^{-2}$	$5{,}20\times10^{-19}$
	Mv	$\mathbf{1{,}60}\times10^{-2}$	$5{,}58\times10^{-2}$	$5{,}96\times10^{-15}$

Table 4: Number of updates per layer $\ell = 0, \ldots, 10$ performed by B^{2}LD during the best run for the network [10 x 50].

	Layer of the network										
Dataset	0	1	2	3	4	5	6	7	8	9	10
Ailerons	33	16	24	63	4	7	4	4	4	4	21
Bank Marketing	30	30	30	30	1	6	2	30	1	1	5
Bejing PM 2.5	0	0	0	0	0	0	0	0	0	0	5
Bikes Sharing	120	62	35	6	62	5	4	4	4	4	18
California	28	28	39	11	2	2	2	2	2	2	13
CCPP	11	2	2	31	2	2	2	2	2	2	4
Mv	30	30	48	8	2	2	2	2	2	2	14

5. Online Block Layer Decomposition algorithm

So far, we have seen that the block layer decomposition method can improve the performance with respect to standard optimization algorithms. In this section, we exploit the addictive structure of the objective function and we embed the B^{2}LD scheme proposed above into a *online* strategy with the aim of enhancing the performance of both pure online methods and pure batch decomposition methods. Numerical results are reported in Section 5.2.

5.1. Online Block Layer Decomposition Method

Let $\mathcal{B}_h \subset \{1, \ldots, P\}$ be the index set identifying a minibatch, i.e. a subset of the samples. Given a partition $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_H\}$ of the set $\{1, \ldots, P\}$, the objective function in problem (3) can be expressed as

$$f(w) = \frac{1}{P} \sum_{h=1}^{H} \left(\sum_{p \in \mathcal{B}_h} E_p(w) + \rho |\mathcal{B}_h| \|w\|^2 \right) = \sum_{h=1}^{H} f_h$$

where

$$f_h = \frac{1}{P} \left(\sum_{p \in \mathcal{B}_h} E_p(w) + \rho |\mathcal{B}_h| \|w\|^2 \right).$$

The *online* BCD algorithm applies a double decomposition to the problem, meaning that at each iteration only information on the function involving a minibatch $\mathcal{B}_h$ is used to update a subset of variables w_ℓ with $\ell \in \mathcal{J}^k \subseteq 1, ..., L$ corresponding to the weights of layers.

In particular at each iteration k, a minibatch $\mathcal{B}_h^k$ is used to update the selected blocks $\ell \in \mathcal{J}^k$ and then both a new *minibatch* $\mathcal{B}_h^{k+1}$ and a new working set $\mathcal{J}^{k+1}$ are chosen. The general framework of an online block layer decomposition scheme is reported in Algorithm 4.

Algorithm 4 Minibatch Block Layer Decomposition

Given $\{x_p, y_p\}_{p=1}^{P}$
Choose $w^0 \in \mathbb{R}^n$ and set $k = 0$;
Define a partition $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_H\}$ of $\{1, \ldots, P\}$
while (stopping criterion not met) **do**
    Select a minibatch $\mathcal{B}_h, h \in \{1, ..., H\}$
    Choose $\mathcal{J} \subseteq \{1, \ldots, L\}$
    Set $\tilde{w} = w^k$;
    for $j = 1, ..., |\mathcal{J}|$ **do**
        Select $\ell^j \in \mathcal{J}$
        Compute Δw_{ℓ^j} according to some rule
        Update $\tilde{w}_{\ell^j} = \tilde{w}_{\ell^j} + \Delta w_{\ell^j}$
    end for
    $w^{k+1} = \tilde{w}$
    $k = k + 1$
end while

Algorithm 4 can be differently characterized depending on the definition of the selection rule of the minibatch $\mathcal{B}_h$, the choice of the subset of weights $\mathcal{J}^k$ and the definition of the update Δw_{ℓ^j}.
We briefly address possible interesting choices in the following paragraphs that lead to the final version of the online decomposition algorithm.

Selection of the minibatch $\mathcal{B}_h$

Selection of the minibatch over the P samples can be implemented following classical rules used in *online* methods for Machine Learning. In particular, given a partition $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_H\}$ of $\{1, \ldots, P\}$ some of the potential upadting rules we can consider are:

- **Incremental Rule**. The order in which minibatches $\mathcal{B}_h$ are selected is defined *a priori* and kept unchanged over the iterations.
- **Stochastic Rule**. Minibatch $\mathcal{B}_h$ is chosen randomly from the available list $\mathcal{B}$.
- **Random without replacement rule**. Incremental with reshuffling after all the minibatches have been considered

Selection of the working set $\mathcal{J}^k$

The index set $\mathcal{J}^k$ defines which blocks will be sequentially updated at iteration k using the minibatch $\mathcal{B}_h$. Different strategies can be chosen for updating the weights. In particular, all the blocks can be updated sequentially using the same minibatch, which corresponds to select $\mathcal{J}^k = \{1, ..., L\}$, or only a subset of the layers are updated with a given minibatch which means $\mathcal{J}^k \subset \{1, ..., L\}$ Furthermore, $\mathcal{J}^k$ can be defined at each iteration, or it can be kept fixed once for all the iterations, i.e. $\mathcal{J}^k = \mathcal{J}\ \forall k$.

In the definition of the set $\mathcal{J}^k$, a computationally efficient choice, that exploits the backpropagation rule for updating the gradient, consists in setting $\mathcal{J}^k = \{1, \dots, L\}$. Indeed, when updating a block of variables $w_{\ell j}$ with a minibatch $\mathcal{B}_h$ in each layer ℓ we can store the output z_ℓ^i of each neuron i so that if the same minibatch is used to update another block of variables $w_{\ell j+1}$ we do not need to evaluate the output z_ℓ of the previous layers $\ell < \ell^{j+1}$, being these unchanged. Similarly to the batch case, in this online framework we propose to consider a scheme where all the layers are updated subsequently with the same minibatch following a backward rule from the last layer L to the input one.

How to compute the update $\Delta w_{\ell j}$

Concerning the updating step of the blocks of variables, different techniques may be implemented. The easiest choice consists in applying an *online* gradient step at the updated point, namely:

$$\Delta w_{\ell j} = -\alpha^k \nabla_{w_{\ell j}} f_h(w_1^k, \dots, w_{\ell j}^k, \tilde{w}_{\ell j+1}, \dots, \tilde{w}_L) \tag{10}$$

where we are assuming a backward rule for the selection of blocks.
The stepsize must be chosen, as standard in online gradient methods, according to a diminishing stepsize rule which guarantees:

$$\sum_{k=0}^{\infty} (\alpha^k)^2 < \infty \qquad \sum_{k=0}^{\infty} (\alpha^k) = \infty \tag{11}$$

In order to accelerate the optimization process, we could introduce a *momentum* term [26], with a different velocity parameter for each block of variables.
Furthermore, other optimization methods (see [7] for an updated review), such as variance reduction methods as SVRG [22] or RMSProp [31] could be embedded too for the update of each block of variables.

5.2. Implementation and performance of an online BLD method

For the experimental results, we implemented an *online* BCD algorithm with the following settings:

- Incremental Rule for selection of minibatches $\mathcal{B}_h$: a partition $\{\mathcal{B}_1, \dots \mathcal{B}_H\}$ is defined e.g. randomly and the order in which minibatches are used is fixed *a priori*;
- for each minibatch all the layers are updated sequentially following a *backward* order, namely $\mathcal{J}^k = \{L, \dots, 1\}$;
- the update step Δw_ℓ^j is computed with a simple gradient direction as in (10), with the aim of checking improvement over classic incremental gradient algorithms.
- α^k is updated according to the following rule:

$$\alpha^k = \alpha^k(1 - \epsilon\alpha^k) \qquad \alpha^0 > 0;\ \epsilon \leq 1$$

  which satisfies (11).

Briefly, at each iteration, the algorithm picks a minibatch $\mathcal{B}_h$ and updates the weights of the layers sequentially in a backward order by performing a steepest descent iteration at the current point. After having updated all the layers, a new minibatch is considered and the same procedure is applied until a certain stopping criterion is met. We call this scheme Block Layer Incremental Gradient (BLInG) which is reported in Algorithm 5.

Algorithm 5 Block Layer Incremental Gradient (BLInG)

Given $\{x_p, y_p\}_{p=1}^{P}, \mathcal{L} = \{1, ..., L\}$
Choose $w^0 \in \mathbb{R}^n$ and set $k = 0, \alpha^0 > 0, \epsilon \in (0; 1)$;
Define a partition $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_H\}$ of $\{1, \ldots, P\}$
while (stopping criterion not met) **do**
    for h=1,...,H **do**
        Set $\tilde{w} = w^k$;
        for $\ell = L, ..., 1$ **do**
            Update $\tilde{w}_\ell = \tilde{w}_\ell - \alpha^k \nabla_{w_\ell} f_h(\tilde{w})$
        end for
        $w^{k+1} = \tilde{w}$
        $\alpha^{k+1} = \alpha^k(1 - \epsilon\alpha^k)$
        $k = k + 1$
    end for
end while

Convergence of BLInG can be proved under suitable assumptions following [4] by looking at the iteration generated by BLInG as a gradient method with error. A similar approach has been followed in [8] for a two-block decomposition where the last block is optimized with a full batch strategy.
We compared BLInG with the standard Incremental Gradient (IG) [3, 2], which represents its non-decomposed counterpart.
As already mentioned, more advanced online algorithms can be embedded within the Block Layer Decomposition framework. In Appendix, we report a BLD version of RMSProp [31] and show its effectiveness through some numerical experiments.
We use for both the two algorithms the same setting and normalization for the update step, implementing a *clipping* strategy on the update step. In particular, following [34], in the updating formula of w_ℓ^k we can normalize the update with a normalization term, properly bounded to avoid overflow,

$$\frac{1}{\max\{\beta, min\{\gamma, \|\nabla_{w_{\ell j}} f_h(\tilde{w})\|\}\}}$$

This choice makes the updating process more robust by avoiding vanishing and exploding gradient. In particular, the thresholds γ and β were fixed to 10^{-3} and 10^6 respectively.
The initial values of the learning rate α^0 and the fraction reduction ϵ were chosen through a grid-search procedure which led to different values for the two methods

which depend on the network architecture:

$$\alpha^0_{IG} = 0.5 \qquad \alpha^0_{BLInG} = \frac{0.5}{\max\{1, L-2\}} \qquad \epsilon = 5 \times 10^{-3}$$

IG performance was invariant with respect to the initial value of the learning rate which best value was the same for all the architectures of the network; BLInG instead performs better with a smaller initial learning rate for deeper networks.
Numerical results are carried out over a larger set of dataset than the one reported in Table 1 by adding three larger datasets which are described in Table 5. These datasets were not considered in the batch case since full batch algorithms suffer when P becomes too large and slow drastically down the convergence.

Table 5: Description of additional dataset considered and corresponding number of variables for each network configuration

Dataset	# Train	# Test	#features	[1 x 50]	[3 x 20]	[200,50,200]	[5 x 50]	[10 x 50]
YearPrediction MSD	412275	103069	92	4650	2660	38600	14650	27150
Covertype	464810	116202	55	2800	1920	31200	12800	25300
Skin_NonSkin	196046	49011	4	250	900	21000	10250	22750

As in the batch case, for each dataset and network configuration, we performed 10 multistart runs starting from randomly chosen initial points. Since *online* methods doesn't evaluate either the objective function or the gradient and are usually faster in finding good solutions for large scale optimization problems, the stopping criterion for these algorithms is a limit on the computational time that was fixed to 60 seconds. For each algorithm, the best run is selected looking at the best value of the objective function over the 10 runs. The results are reported in Table 6 where for each of the two algorithms, we report the best values of the objective function found over the 10 runs and the corresponding value of the test error within the time limit.
Overall, the introduction of a block layer decomposition improves the algorithm performance especially when the dimension of the network blows up. Indeed, if BLInG is generally able to return better solutions with better generalization properties on the test set than those returned by IG, this gap increases when it comes to solving problems where the network becomes very large and wide.
In order to assess the average behaviour over the 10 multistart runs of the two algorithms BLInG and IG, as done for the batch case, we performed a two-tailed T-test in order to assess whether the means of the values of the test error obtained by the two algorithms are statistically different or not. For each dataset and network results are reported in Table 7. In those experiments where the two means are statistically different, the smaller mean is highlighted in bold case. On average, for the deepest network [10 x 50], BLInG returns values which are statistically better than those returned by IG, while, for smaller networks, performance is more similar and not always statistically different with the exception of the shallow network.
Furthermore, in order to highlight the behaviour of the algorithms among the iterations, we plot in Figure 2 the values of the objective function in the first 10 seconds for some

Table 6: Function values returned by the best run over 10 multistart runs by the BLInG and IG algorithms

Network	Dataset	Objective Function		Test Error	
		BLInG	IG	BLInG	IG
[1 x 50]	Ailerons	$2{,}01\times10^{-3}$	$\mathbf{1{,}99}\times10^{-3}$	$1{,}98\times10^{-3}$	$\mathbf{1{,}93}\times10^{-3}$
	Bank Marketing	$2{,}45\times10^{-6}$	$\mathbf{2{,}01}\times10^{-6}$	$2{,}83\times10^{-5}$	$\mathbf{1{,}29}\times10^{-5}$
	Bejing PM 2.5	$\mathbf{4{,}26}\times10^{-3}$	$4{,}63\times10^{-3}$	$\mathbf{3{,}98}\times10^{-3}$	$4{,}28\times10^{-3}$
	Bikes Sharing	$\mathbf{1{,}79}\times10^{-3}$	$2{,}21\times10^{-3}$	$\mathbf{1{,}95}\times10^{-3}$	$2{,}38\times10^{-3}$
	California	$\mathbf{1{,}69}\times10^{-2}$	$1{,}89\times10^{-2}$	$\mathbf{1{,}74}\times10^{-2}$	$1{,}93\times10^{-2}$
	CCPP	$\mathbf{3{,}17}\times10^{-3}$	$3{,}21\times10^{-3}$	$\mathbf{3{,}14}\times10^{-3}$	$3{,}18\times10^{-3}$
	Covtype	$\mathbf{3{,}08}\times10^{-2}$	$3{,}20\times10^{-2}$	$\mathbf{3{,}08}\times10^{-2}$	$3{,}20\times10^{-2}$
	Mv	$\mathbf{7{,}40}\times10^{-5}$	$8{,}03\times10^{-5}$	$\mathbf{7{,}72}\times10^{-5}$	$8{,}22\times10^{-5}$
	Skin_NonSkin	$\mathbf{1{,}15}\times10^{-2}$	$1{,}93\times10^{-2}$	$\mathbf{1{,}13}\times10^{-2}$	$1{,}91\times10^{-2}$
	YearPredictionMSD	$1{,}14\times10^{-2}$	$1{,}14\times10^{-2}$	$1{,}14\times10^{-2}$	$1{,}14\times10^{-2}$
[3 x 20]	Ailerons	$\mathbf{2{,}02}\times10^{-3}$	$2{,}05\times10^{-3}$	$\mathbf{1{,}94}\times10^{-3}$	$2{,}00\times10^{-3}$
	Bank Marketing	$\mathbf{2{,}40}\times10^{-7}$	$1{,}26\times10^{-6}$	$\mathbf{1{,}07}\times10^{-6}$	$4{,}24\times10^{-6}$
	Bejing PM 2.5	$\mathbf{4{,}12}\times10^{-3}$	$4{,}19\times10^{-3}$	$\mathbf{3{,}83}\times10^{-3}$	$3{,}94\times10^{-3}$
	Bikes Sharing	$\mathbf{1{,}78}\times10^{-3}$	$1{,}80\times10^{-3}$	$\mathbf{1{,}87}\times10^{-3}$	$1{,}93\times10^{-3}$
	California	$\mathbf{1{,}61}\times10^{-2}$	$1{,}65\times10^{-2}$	$\mathbf{1{,}68}\times10^{-2}$	$1{,}74\times10^{-2}$
	CCPP	$3{,}14\times10^{-3}$	$\mathbf{3{,}13}\times10^{-3}$	$\mathbf{3{,}07}\times10^{-3}$	$3{,}07\times10^{-3}$
	Covtype	$\mathbf{3{,}25}\times10^{-2}$	$3{,}39\times10^{-2}$	$\mathbf{3{,}26}\times10^{-2}$	$3{,}40\times10^{-2}$
	Mv	$\mathbf{1{,}46}\times10^{-5}$	$2{,}10\times10^{-5}$	$\mathbf{1{,}51}\times10^{-5}$	$2{,}15\times10^{-5}$
	Skin_NonSkin	$\mathbf{4{,}29}\times10^{-3}$	$1{,}81\times10^{-2}$	$\mathbf{4{,}24}\times10^{-3}$	$1{,}78\times10^{-2}$
	YearPredictionMSD	$\mathbf{1{,}13}\times10^{-2}$	$1{,}15\times10^{-2}$	$\mathbf{1{,}12}\times10^{-2}$	$1{,}15\times10^{-2}$
[200,50,200]	Ailerons	$\mathbf{2{,}09}\times10^{-3}$	$2{,}14\times10^{-3}$	$\mathbf{2{,}03}\times10^{-3}$	$2{,}07\times10^{-3}$
	Bank Marketing	$\mathbf{7{,}14}\times10^{-6}$	$2{,}16\times10^{-4}$	$\mathbf{1{,}59}\times10^{-5}$	$3{,}22\times10^{-4}$
	Bejing PM 2.5	$\mathbf{4{,}60}\times10^{-3}$	$5{,}02\times10^{-3}$	$\mathbf{4{,}27}\times10^{-3}$	$4{,}68\times10^{-3}$
	Bikes Sharing	$\mathbf{2{,}52}\times10^{-3}$	$3{,}10\times10^{-3}$	$\mathbf{2{,}53}\times10^{-3}$	$3{,}11\times10^{-3}$
	California	$\mathbf{1{,}81}\times10^{-2}$	$2{,}15\times10^{-2}$	$\mathbf{1{,}90}\times10^{-2}$	$2{,}22\times10^{-2}$
	CCPP	$\mathbf{3{,}16}\times10^{-3}$	$3{,}22\times10^{-3}$	$\mathbf{3{,}10}\times10^{-3}$	$3{,}17\times10^{-3}$
	Covtype	$\mathbf{3{,}36}\times10^{-2}$	$3{,}40\times10^{-2}$	$\mathbf{3{,}37}\times10^{-2}$	$3{,}41\times10^{-2}$
	Mv	$\mathbf{4{,}87}\times10^{-5}$	$5{,}40\times10^{-4}$	$\mathbf{4{,}90}\times10^{-5}$	$5{,}33\times10^{-4}$
	Skin_NonSkin	$\mathbf{1{,}60}\times10^{-2}$	$2{,}55\times10^{-2}$	$\mathbf{1{,}59}\times10^{-2}$	$2{,}53\times10^{-2}$
	YearPredictionMSD	$1{,}18\times10^{-2}$	$1{,}18\times10^{-2}$	$1{,}18\times10^{-2}$	$1{,}18\times10^{-2}$
[5 x 50]	Ailerons	$\mathbf{2{,}04}\times10^{-3}$	$2{,}10\times10^{-3}$	$\mathbf{1{,}99}\times10^{-3}$	$2{,}04\times10^{-3}$
	Bank Marketing	$\mathbf{6{,}99}\times10^{-7}$	$1{,}18\times10^{-5}$	$\mathbf{2{,}46}\times10^{-6}$	$2{,}13\times10^{-5}$
	Bejing PM 2.5	$\mathbf{4{,}22}\times10^{-3}$	$4{,}35\times10^{-3}$	$\mathbf{3{,}92}\times10^{-3}$	$4{,}03\times10^{-3}$
	Bikes Sharing	$\mathbf{2{,}02}\times10^{-3}$	$2{,}19\times10^{-3}$	$\mathbf{2{,}15}\times10^{-3}$	$2{,}32\times10^{-3}$
	California	$\mathbf{1{,}68}\times10^{-2}$	$1{,}77\times10^{-2}$	$\mathbf{1{,}73}\times10^{-2}$	$1{,}83\times10^{-2}$
	CCPP	$\mathbf{3{,}10}\times10^{-3}$	$3{,}15\times10^{-3}$	$\mathbf{3{,}04}\times10^{-3}$	$3{,}09\times10^{-3}$
	Covtype	$\mathbf{3{,}35}\times10^{-2}$	$3{,}47\times10^{-2}$	$\mathbf{3{,}36}\times10^{-2}$	$3{,}48\times10^{-2}$
	Mv	$\mathbf{1{,}14}\times10^{-5}$	$5{,}01\times10^{-5}$	$\mathbf{1{,}14}\times10^{-5}$	$5{,}07\times10^{-5}$
	Skin_NonSkin	$\mathbf{1{,}09}\times10^{-2}$	$1{,}78\times10^{-2}$	$\mathbf{1{,}08}\times10^{-2}$	$1{,}75\times10^{-2}$
	YearPredictionMSD	$\mathbf{1{,}13}\times10^{-2}$	$1{,}17\times10^{-2}$	$\mathbf{1{,}12}\times10^{-2}$	$1{,}17\times10^{-2}$
[10 x 50]	Ailerons	$\mathbf{2{,}13}\times10^{-3}$	$2{,}17\times10^{-3}$	$\mathbf{2{,}12}\times10^{-3}$	$2{,}12\times10^{-3}$
	Bank Marketing	$\mathbf{8{,}32}\times10^{-7}$	$5{,}01\times10^{-5}$	$\mathbf{7{,}45}\times10^{-7}$	$4{,}19\times10^{-5}$
	Bejing PM 2.5	$\mathbf{4{,}75}\times10^{-3}$	$8{,}68\times10^{-3}$	$\mathbf{4{,}45}\times10^{-3}$	$8{,}15\times10^{-3}$
	Bikes Sharing	$\mathbf{2{,}88}\times10^{-3}$	$4{,}16\times10^{-3}$	$\mathbf{2{,}95}\times10^{-3}$	$4{,}31\times10^{-3}$
	California	$\mathbf{1{,}80}\times10^{-2}$	$5{,}49\times10^{-2}$	$\mathbf{1{,}88}\times10^{-2}$	$5{,}81\times10^{-2}$
	CCPP	$\mathbf{3{,}30}\times10^{-3}$	$3{,}49\times10^{-3}$	$\mathbf{3{,}21}\times10^{-3}$	$3{,}40\times10^{-3}$
	Covtype	$\mathbf{3{,}45}\times10^{-2}$	$3{,}55\times10^{-2}$	$\mathbf{3{,}45}\times10^{-2}$	$3{,}56\times10^{-2}$
	Mv	$\mathbf{3{,}58}\times10^{-4}$	$8{,}31\times10^{-4}$	$\mathbf{3{,}51}\times10^{-4}$	$7{,}96\times10^{-4}$
	Skin_NonSkin	$\mathbf{1{,}89}\times10^{-2}$	$4{,}39\times10^{-2}$	$\mathbf{1{,}86}\times10^{-2}$	$4{,}41\times10^{-2}$
	YearPredictionMSD	$\mathbf{1{,}24}\times10^{-2}$	$1{,}51\times10^{-2}$	$\mathbf{1{,}23}\times10^{-2}$	$1{,}50\times10^{-2}$

Table 7: Comparison, for each dataset and network structure, of the means of the test error values returned by BLInG and IG over the 10 runs and the corresponding p-value of a two-tailed T-Test

Network	Dataset	Mean BLInG	Mean IG	p-value
[1 x 50]	Ailerons	$2{,}02\times10^{-3}$	$\mathbf{1{,}95}\times10^{-3}$	$3{,}36\times10^{-3}$
	Bank Marketing	$2{,}51\times10^{-5}$	$\mathbf{1{,}72}\times10^{-5}$	$3{,}68\times10^{-2}$
	Bejing Pm25	$\mathbf{4{,}02}\times10^{-3}$	$4{,}41\times10^{-3}$	$1{,}68\times10^{-10}$
	Bikes Sharing	$\mathbf{2{,}12}\times10^{-3}$	$2{,}43\times10^{-3}$	$2{,}76\times10^{-9}$
	California	$\mathbf{1{,}83}\times10^{-2}$	$1{,}96\times10^{-2}$	$9{,}39\times10^{-6}$
	CCPP	$\mathbf{3{,}15}\times10^{-3}$	$3{,}19\times10^{-3}$	$6{,}53\times10^{-4}$
	Covtype	$\mathbf{3{,}15}\times10^{-2}$	$3{,}23\times10^{-2}$	$6{,}32\times10^{-5}$
	Mv	$\mathbf{8{,}82}\times10^{-5}$	$1{,}02\times10^{-4}$	$2{,}88\times10^{-2}$
	Skin_NonSkin	$\mathbf{1{,}40}\times10^{-2}$	$2{,}05\times10^{-2}$	$7{,}89\times10^{-10}$
	YearPredictionMSD	$1{,}16\times10^{-2}$	$\mathbf{1{,}15}\times10^{-2}$	$7{,}32\times10^{-3}$
[3 x 20]	Ailerons	$2{,}03\times10^{-3}$	$2{,}02\times10^{-3}$	$5{,}16\times10^{-1}$
	Bank Marketing	$9{,}00\times10^{-6}$	$5{,}64\times10^{-6}$	$2{,}09\times10^{-1}$
	Bejing Pm25	$3{,}96\times10^{-3}$	$3{,}97\times10^{-3}$	$7{,}68\times10^{-1}$
	Bikes Sharing	$2{,}02\times10^{-3}$	$1{,}96\times10^{-3}$	$2{,}13\times10^{-1}$
	California	$1{,}72\times10^{-2}$	$1{,}74\times10^{-2}$	$7{,}00\times10^{-2}$
	CCPP	$3{,}19\times10^{-3}$	$3{,}08\times10^{-3}$	$6{,}60\times10^{-2}$
	Covtype	$\mathbf{3{,}31}\times10^{-2}$	$3{,}44\times10^{-2}$	$8{,}23\times10^{-7}$
	Mv	$3{,}38\times10^{-5}$	$4{,}72\times10^{-5}$	$3{,}07\times10^{-1}$
	Skin_NonSkin	$\mathbf{5{,}78}\times10^{-3}$	$1{,}83\times10^{-2}$	$6{,}98\times10^{-17}$
	YearPredictionMSD	$1{,}16\times10^{-2}$	$1{,}15\times10^{-2}$	$8{,}49\times10^{-1}$
[200,50,200]	Ailerons	$3{,}28\times10^{-3}$	$2{,}30\times10^{-3}$	$3{,}17\times10^{-1}$
	Bank Marketing	$8{,}71\times10^{-4}$	$6{,}52\times10^{-4}$	$6{,}65\times10^{-1}$
	Bejing Pm25	$7{,}05\times10^{-3}$	$5{,}65\times10^{-3}$	$1{,}72\times10^{-1}$
	Bikes Sharing	$\mathbf{4{,}15}\times10^{-3}$	$6{,}52\times10^{-3}$	$4{,}17\times10^{-2}$
	California	$2{,}28\times10^{-2}$	$2{,}33\times10^{-2}$	$7{,}33\times10^{-1}$
	CCPP	$4{,}47\times10^{-3}$	$3{,}84\times10^{-3}$	$4{,}66\times10^{-1}$
	Covtype	$3{,}57\times10^{-2}$	$3{,}49\times10^{-2}$	$3{,}84\times10^{-1}$
	Mv	$\mathbf{6{,}49}\times10^{-4}$	$1{,}44\times10^{-3}$	$2{,}29\times10^{-2}$
	Skin_NonSkin	$\mathbf{2{,}14}\times10^{-2}$	$2{,}73\times10^{-2}$	$1{,}62\times10^{-4}$
	YearPredictionMSD	$1{,}28\times10^{-2}$	$1{,}22\times10^{-2}$	$2{,}27\times10^{-1}$
[5 x 50]	Ailerons	$2{,}22\times10^{-3}$	$2{,}07\times10^{-3}$	$1{,}30\times10^{-1}$
	Bank Marketing	$\mathbf{8{,}01}\times10^{-6}$	$2{,}71\times10^{-5}$	$2{,}05\times10^{-6}$
	Bejing Pm25	$4{,}15\times10^{-3}$	$4{,}11\times10^{-3}$	$6{,}34\times10^{-1}$
	Bikes Sharing	$2{,}36\times10^{-3}$	$2{,}40\times10^{-3}$	$4{,}60\times10^{-1}$
	California	$1{,}84\times10^{-2}$	$1{,}85\times10^{-2}$	$8{,}15\times10^{-1}$
	CCPP	$3{,}26\times10^{-3}$	$3{,}15\times10^{-3}$	$2{,}57\times10^{-1}$
	Covtype	$\mathbf{3{,}42}\times10^{-2}$	$3{,}51\times10^{-2}$	$3{,}71\times10^{-5}$
	Mv	$\mathbf{2{,}39}\times10^{-5}$	$7{,}54\times10^{-5}$	$1{,}80\times10^{-9}$
	Skin_NonSkin	$2{,}03\times10^{-2}$	$1{,}79\times10^{-2}$	$5{,}81\times10^{-2}$
	YearPredictionMSD	$1{,}16\times10^{-2}$	$1{,}19\times10^{-2}$	$5{,}79\times10^{-2}$
[10 x 50]	Ailerons	$\mathbf{2{,}19}\times10^{-3}$	$7{,}43\times10^{-3}$	$5{,}52\times10^{-3}$
	Bank Marketing	$\mathbf{1{,}75}\times10^{-5}$	$1{,}41\times10^{-1}$	$7{,}50\times10^{-3}$
	Bejing Pm25	$\mathbf{6{,}13}\times10^{-3}$	$8{,}16\times10^{-3}$	$2{,}36\times10^{-3}$
	Bikes Sharing	$\mathbf{3{,}53}\times10^{-3}$	$1{,}20\times10^{-2}$	$2{,}84\times10^{-2}$
	California	$\mathbf{1{,}94}\times10^{-2}$	$5{,}84\times10^{-2}$	$1{,}83\times10^{-30}$
	CCPP	$3{,}36\times10^{-3}$	$3{,}55\times10^{-3}$	$6{,}12\times10^{-2}$
	Covtype	$\mathbf{3{,}61}\times10^{-2}$	$4{,}78\times10^{-2}$	$3{,}46\times10^{-4}$
	Mv	$4{,}08\times10^{-4}$	$6{,}45\times10^{-3}$	$2{,}83\times10^{-1}$

meaningful runs. Values are plotted in a logarithmic scale (adding +1 to each term in order to get always positive values). Overall, the objective function values decrease faster when optimizing with BLInG than IG, and then the two algorithms stabilize to similar values. However, it seems that IG is more hindered than BLInG by the depth of the network, having troubles in reducing the error in the deepest network. This might explain the better results obtained by BLInG on the deepest network in both the analysis reported in Tables 6 and 7, e.g. on the Bank Marketing dataset.
In order to assess how much the depth of the network influences algorithms' performance, in Table 8 the ratio between best value found in the network [10 × 50] and [1 × 50] are provided. Higher values mean that the algorithm has been harmed by the increased depth of the network. Comparing these solutions, BLInG turns out to be less affected by the increased structure of the network and is able to find always similar values regardless of the structure of the network while IG performs worse.

Table 8: Ratio between best value found in [10 × 50] and [1 × 50]. The smaller the value, the better the performance of the algorithm when dealing with deep networks.

Network	Objective value		Test Error	
[10x50]/[1x50]	BLInG	IG	BLInG	IG
Ailerons	1,06	1,09	1,07	1,10
Bank Marketing	0,34	24,91	0,03	3,25
Bejing PM 2.5	1,12	1,88	1,12	1,91
Bikes Sharing	1,61	1,89	1,51	1,81
California	1,06	2,91	1,08	3,01
CCPP	1,04	1,09	1,02	1,07
Covertype	1,12	1,11	1,12	1,11
Mv	4,84	10,35	4,54	9,68
Skin_NonSkin	1,64	2,28	1,65	2,31
YearPredictionMSD	1,08	1,32	1,08	1,32

6. Conclusions

In this work, we focused on the application of batch and online Block Coordinate Decomposition methods for training Deep Feedforward Neural Networks. We studied how the layered structure of a DFNN can be effectively leveraged for training these models and we defined general *batch* and *online* block layer decomposition schemes. Extensive numerical experiments over different network architectures were performed to assess how the performance of state-of-the-art algorithms can be improved. Overall, the application of Block Coordinate Decomposition methods turned out to be effective in avoiding bad attraction regions and speeding up the training process of DFNNs. Both the two proposed methods outperformed no-variable-decomposed counterparts leading to better solutions with better generalization properties as well.

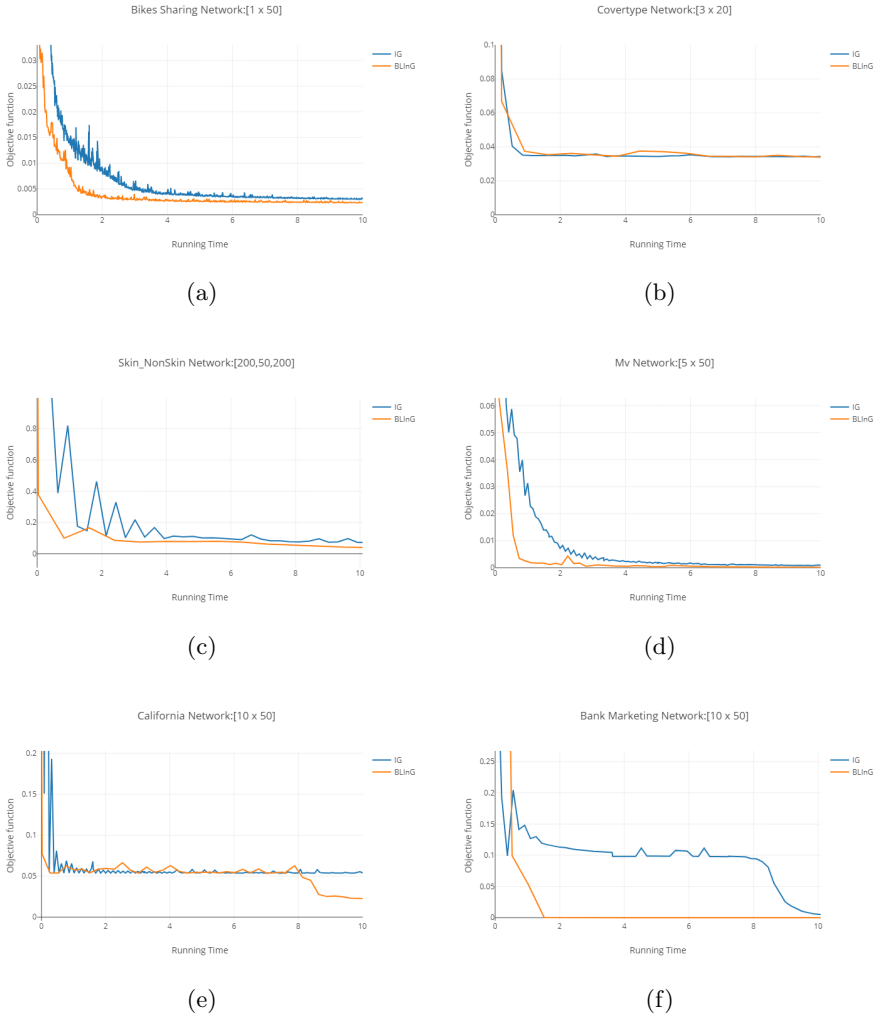

Figure 2: Comparison of the objective function behaviour during the first 10 seconds of optimization of both BLInG and IG

Appendix A. Block Layer RMSProp implementation

In order to show how more sophisticated online algorithms can be embedded in the block decomposition framework, we consider the RMSProp algorithm instead of the incremental gradient as optimization method and we embed it in our backward decomposition algorithm. RMSProp is one of the most widely used training algorithms for DFNN [17] and it is easier to tune rather than other online algorithms, such as e.g. Adam or variance reduction methods, having only two hyperparameters to be tuned.
In the Block Layer decomposition scheme with RMSProp we need to change the updating rule Δw_ℓin Step 10 of Algorithm 4 as

$$\Delta w_\ell = \alpha^k(\delta + r_\ell^{k+1})^{-1/2} \odot d_\ell$$

where:

$$d_\ell = -\nabla_{w_\ell} f_h(\tilde{w})$$
$$r_\ell^{k+1} = \rho r_\ell^k + (1-\rho) d_\ell \odot d_\ell \qquad r_\ell^0 = 0$$

with $\rho, \delta > 0$, $\odot$ is the classical componentwise product and the stepsize must be driven to zero as e.g. $\alpha^{k+1} = \alpha^k(1 - \epsilon\alpha^k)$.
We implemented the BLD version of RMSProp (BL-RMSProp) and the classical (RMSProp) with the hyperparameters fixed to the default values used in the RMSProp implementation of Keras [11], i.e. $\rho = 0.9$, $\delta = 10^{-6}$, $\alpha^0 = 0.001$ and $\epsilon = 0.005$ as in the previous numerical experiments.
We perform 10 multistart runs of both the algorithms only on those datasets of Tables 1 and 5 with more than 30.000 samples and on a subset of networks, being interested in assessing the performance over the largest problems and deepest networks. In Table A.9 we report the best value of the objective function returned by each algorithm among the 10 runs.
The results in Table A.9, although not extensive, show that the use of a block decomposition strategy coupled with the online RMSprop algorithm helps to improve the performance of the algorithm in most of the cases when the network is large. Thus, coupling online methods and block decomposition seems to be a promising approach whatever is the online method used.

[1] A. Beck and L. Tetruashvili. On the convergence of block coordinate descent type methods. *SIAM Journal on Optimization*, 23(4):2037–2060, 2013.

[2] D. P. Bertsekas. Incremental least squares methods and the extended kalman filter. *SIAM J. on Optimization*, 6(3):807–822, Mar. 1996.

[3] D. P. Bertsekas. Incremental gradient, subgradient, and proximal methods for convex optimization: A survey. *CoRR*, abs/1507.01030, 2015.

[4] D. P. Bertsekas and J. N. Tsitsiklis. Gradient Convergence in Gradient methods with Errors. *SIAM Journal on Optimization*, 10(3):627–642, 2000.

Table A.9: Function values returned by the best run over 10 multistart runs by the BL-RMSProp and RMSProp.

	Objective Function					
	[200.50.200]		[5 x 50]		[10 x 50]	
	BL-RMSProp	RMSProp	BL-RMSProp	RMSProp	BL-RMSProp	RMSProp
Bank Marketing	$\mathbf{2.77}\times10^{-5}$	2.98×10^{-3}	$\mathbf{1.86}\times10^{-5}$	2.11×10^{-5}	$\mathbf{1.38}\times10^{-6}$	6.92×10^{-5}
Bejing PM 2.5	$\mathbf{4.31}\times10^{-3}$	4.39×10^{-3}	4.14×10^{-3}	$\mathbf{4.07}\times10^{-3}$	$\mathbf{4.58}\times10^{-3}$	8.56×10^{-3}
Covertype	$\mathbf{3.11}\times10^{-2}$	3.22×10^{-2}	3.29×10^{-2}	$\mathbf{3.13}\times10^{-2}$	$\mathbf{3.46}\times10^{-2}$	5.42×10^{-2}
Mv	$\mathbf{1.93}\times10^{-4}$	2.42×10^{-3}	8.76×10^{-5}	$\mathbf{1.65}\times10^{-5}$	4.87×10^{-4}	$\mathbf{4.15}\times10^{-4}$
Skin_NonSkin	1.53×10^{-2}	$\mathbf{7.10}\times10^{-3}$	$\mathbf{1.49}\times10^{-2}$	1.65×10^{-2}	$\mathbf{1.88}\times10^{-2}$	1.64×10^{-1}
YearPredictionMSD	$\mathbf{1.21}\times10^{-2}$	1.44×10^{-2}	1.16×10^{-2}	$\mathbf{1.11}\times10^{-2}$	$\mathbf{1.19}\times10^{-2}$	1.51×10^{-2}
	Test Error					
	[200.50.200]		[5 x 50]		[10 x 50]	
	BL-RMSProp	RMSProp	BL-RMSProp	RMSProp	BL-RMSProp	RMSProp
Bank Marketing	$\mathbf{2.76}\times10^{-5}$	2.98×10^{-3}	$\mathbf{1.84}\times10^{-5}$	2.11×10^{-5}	$\mathbf{1.42}\times10^{-6}$	6.92×10^{-5}
Bejing PM 2.5	$\mathbf{4.39}\times10^{-3}$	4.46×10^{-3}	4.24×10^{-3}	$\mathbf{4.19}\times10^{-3}$	$\mathbf{4.65}\times10^{-3}$	8.63×10^{-3}
Covertype	$\mathbf{3.10}\times10^{-2}$	3.21×10^{-2}	3.29×10^{-2}	$\mathbf{3.14}\times10^{-2}$	$\mathbf{3.47}\times10^{-2}$	5.42×10^{-2}
Mv	$\mathbf{1.91}\times10^{-4}$	2.42×10^{-3}	8.71×10^{-5}	$\mathbf{1.65}\times10^{-5}$	4.84×10^{-4}	$\mathbf{4.11}\times10^{-4}$
Skin_NonSkin	1.51×10^{-2}	$\mathbf{7.00}\times10^{-3}$	$\mathbf{1.47}\times10^{-2}$	1.62×10^{-2}	1.86×10^{-2}	$\mathbf{1.65}\times10^{-1}$
YearPredictionMSD	$\mathbf{1.21}\times10^{-2}$	1.43×10^{-2}	1.16×10^{-2}	$\mathbf{1.11}\times10^{-2}$	$\mathbf{1.19}\times10^{-2}$	1.50×10^{-2}

[5] u. y. Bertsekas, Dimitri P. , isbn=9787302482345. *Nonlinear Programming (third Edition)*.

[6] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *in COMPSTAT*, 2010.

[7] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[8] L. Bravi and M. Sciandrone. An incremental decomposition method for unconstrained optimization. *Applied Mathematics and Computation*, 235:80–86, 2014.

[9] C. Buzzi, L. Grippo, and M. Sciandrone. Convergent decomposition techniques for training rbf neural networks. *Neural Computation*, 13(8):1891–1920, 2001.

[10] V. K. Chauhan, K. Dahiya, and A. Sharma. Mini-batch block-coordinate based stochastic average adjusted gradient methods to solve big data problems. In *Proceedings of the Ninth Asian Conference on Machine Learning*, volume 77 of *Proceedings of Machine Learning Research*, pages 49–64. PMLR, 15–17 Nov 2017.

[11] F. Chollet et al. Keras, 2015.

[12] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems 27*, pages 2933–2941. 2014.

[13] A. Defazio, F. Bach, and S. Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems 27*, pages 1646–1654. 2014.

[14] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[15] R. A. Fisher. Statistical methods for research workers. In *Breakthroughs in statistics*, pages 66–70. Springer, 1992.

[16] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. urlhttp://www.deeplearningbook.org.

[18] L. Grippo, A. Manno, and M. Sciandrone. Decomposition Techniques for Multilayer Perceptron Training. *IEEE Transactions on Neural Networks and Learning Systems*, 27(11):2146–2159, 2016.

[19] L. Grippo and M. Sciandrone. Globally convergent block-coordinate techniques for unconstrained optimization. *Optimization Methods and Software*, 10(4):587–637, 1999.

[20] H. Guang-bin, Z. Qin-yu, and S. Chee-kheong. Extreme learning machine: Theory and applications, 2006.

[21] G.-B. Huang, D. H. Wang, and Y. Lan. Extreme learning machines: a survey. *International Journal of Machine Learning and Cybernetics*, 2(2):107–122, Jun 2011.

[22] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems 26*, pages 315–323. 2013.

[23] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed ¡today¿].

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[25] Y. Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[26] Y. E. Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[27] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[28] L. Palagi. Global optimization issues in deep network regression: an overview. *Journal of Global Optimization*, pages 1–39, 2018.

[29] T. Qin, K. Scheinberg, and D. Goldfarb. Efficient block-coordinate descent algorithms for the group lasso. 5, 06 2013.

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

[31] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[32] H. Wang and A. Banerjee. Randomized block coordinate descent for online and stochastic optimization. *arXiv preprint arXiv:1407.0107*, 2014.

[33] S. J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[34] A. W. Yu, L. Huang, Q. Lin, R. Salakhutdinov, and J. Carbonell. Normalized gradient with adaptive stepsize method for deep neural network training. *CoRR*, abs/1707.04822, 2017.

[35] T. Zhao, M. Yu, Y. Wang, R. Arora, and H. Liu. Accelerated mini-batch randomized block coordinate descent method. In *Advances in neural information processing systems*, pages 3329–3337, 2014.