Constraints (2016) 21:435–462
DOI 10.1007/s10601-015-9234-6

A lagrangian propagator for artificial neural networks in constraint programming

Michele Lombardi[1] · **Stefano Gualandi**[2]

Published online: 12 November 2015

Abstract This paper discusses a new method to perform propagation over a (two-layer, feed-forward) Neural Network embedded in a Constraint Programming model. The method is meant to be employed in Empirical Model Learning, a technique designed to enable optimal decision making over systems that cannot be modeled via conventional declarative means. The key step in Empirical Model Learning is to embed a Machine Learning model into a combinatorial model. It has been showed that Neural Networks can be embedded in a Constraint Programming model by simply encoding each neuron as a global constraint, which is then propagated individually. Unfortunately, this decomposition approach may lead to weak bounds. To overcome such limitation, we propose a new network-level propagator based on a non-linear Lagrangian relaxation that is solved with a subgradient algorithm. The method proved capable of dramatically reducing the search tree size on a thermal-aware dispatching problem on multicore CPUs. The overhead for optimizing the Lagrangian multipliers is kept within a reasonable level via a few simple techniques. This paper is an extended version of [27], featuring an improved structure, a new filtering technique for the network inputs, a set of overhead reduction techniques, and a thorough experimentation.

Keywords Constraint programming · Lagrangian relaxation · Neural networks

✉ Michele Lombardi
michele.lombardi2@unibo.it

Stefano Gualandi
stefano.gualandi@antoptima.ch

[1] University of Bologna, Viale del Risorgimento 2, 40136, Bologna, Italy

[2] AntOptima SA, Via Aprica 26, 6900, Lugano, Switzerland

1 Introduction

Combinatorial Optimization techniques have been subject to dramatic improvements in the last decades, which enabled their successful application to a large number of industrial problems. Yet, many real-world domains are still out-of-reach for such approaches. In many cases, this is due to difficulties in formulating an accurate declarative model for the system to be optimized.

Recently, the Empirical Model Learning technique (EML, see [2]) was introduced to address this kind of issues. The main idea in EML is to embed a Machine Learning model into a more complex and comprehensive combinatorial optimization model. The Machine Learning model approximates the behavior of a system that is impervious to traditional modeling efforts. In an EML based approach, the optimization model contains a predicate in the form:

$$y = H(x) \tag{1}$$

where H is the encoding of a Machine Learning model that captures the effect of some decision variables (i.e. x) on some observables of interest (i.e. y). The observables may appear in the cost function or in some of the constraints. Unlike in derivative-free optimization (see [1, 12]), in EML the H function is not treated as a black-box. On the contrary, there is an emphasis on *exploiting the structure of the Machine Learning model to boost the search process*.

So far, the EML approach has been instantiated using Artificial Neural Networks (ANNs), Decision Trees, and Random Forests on the Machine Learning side. The employed optimization techniques are instead Constraint Programming (CP) and Local Search. For more details, the interested reader can refer to [2, 3, 9]. EML approaches based on Mixed-Integer Non-Linear Programming (MINLP) and SAT Modulo Theories (SMT) are currently under development.

Possible application examples for EML include more complex version of traditional engineering problems (e.g. design of chemical plants, prosthetic limbs, land and air vehicles), operating data centers and their cooling systems, traffic light placement, or classical optimization problems that could benefit from a more advanced predictive model (e.g. retail store location, shift scheduling).

This paper is related to works [2, 3] that show how to embed an ANN into a CP model by encoding each neuron as a global *Neuron Constraint*. The neuron inputs and the output are encoded as real-valued decision variables (say x_i for the i-th input and z for the output). Each Neuron Constraint enforces bound consistency on the equation that defines the neuron behavior. In other words, a dedicated algorithm (i.e. a *propagator*) can prune provably infeasible values from the domain of z based on the domain of the x_i, and vice versa.

In turn, such domain updates can trigger the propagators associated to other constraints. This cascade process is known as *constraint propagation* and it is one of the cornerstones of CP. Propagation is typically employed during optimization in order to reduce (often dramatically) the size of the search tree. The technique should not be confused with the error propagation employed during the training of an ANN: indeed, *in this paper we will always assume to deal with pre-trained networks*.

Using a separate constraint to encode each neuron allows for great flexibility, but such fine-grain decomposition may decrease the propagation effectiveness. It makes therefore sense to devise specialized propagators for the most common ANN types, or for more coarse-grain sub-structures. In this spirit, in [27] we have introduced a new propagator for two-layer, feed forward, ANNs. The approach does not subsume the propagation performed by individual Neuron Constraints, but it can provide tighter bounds on the network output. The new propagator can be instantiated multiple times to model networks with more than two layers, or it can be combined with Neuron Constraints to encode almost any type of ANN, including recurrent networks.[1]

The bounds for the new propagator are based on a non-linear Lagrangian relaxation, with the multipliers being optimized via a subgradient method. Lagrangian relaxation has proved to be a powerful technique for performing reduced-cost filtering in CP, mainly in the context of linear relaxations of integer programs (e.g., [32–34]). The non-linear Lagrangian relaxation used in this paper is inspired by the work presented in [19], for solving resource constrained shortest path problems with a super additive objective function.

This paper extends our previous work [27] by: 1) introducing a simple an efficient method to obtain *bounds on the input variables*; 2) presenting *techniques to reduce the computation overhead*; 3) adding an *extensive experimentation*. We compare our new propagator to a baseline approach using only separate Neuron Constraints to encode ANNs. The two approaches are compared in a standard fashion, i.e. by solving to completeness a number of instances of an optimization problem. In particular, we consider small-size instances of a thermal-aware job mapping problem for a multicore CPU. We show how the new propagator can lead to a substantial (sometimes massive) reduction of the search tree size. Our overhead reduction techniques allow us to keep the propagation time within manageable limits, so that in a significant number of cases the new approach is considerably more efficient than the baseline.

The paper is structured as follows: Section 2 provides background information. Section 3 introduces the new propagator. Then, Section 3.1 presents the Lagrangian relaxation, Section 3.2 explains how the multipliers are optimized, Section 3.3 shows how to perform filtering on the network input, and Section 3.4 discusses the overhead reduction techniques. Section 4 provides our experimental results and Section 5 the concluding remarks.

2 Background

The Lagrangian propagator presented in this paper is designed for EML. The key idea of EML is using Machine Learning to obtain an approximate *Empirical Model* for a part of the system that is difficult to tackle via conventional modeling techniques. For example, in [3] the authors address a workload dispatching problem: a set of jobs must be mapped on a 48-core system with thermal controllers; bad mapping decisions may lead to overheating, which may cause a loss of efficiency when the controllers slow down the cores to decrease their temperature. The problem is in principle well suited for Combinatorial Optimization, except that modeling the mapping-dependent efficiency is nearly-impossible via conventional means: in fact, it requires to capture the combined effect of the thermal physics and the action of the on-line controllers, with their dynamic policies. EML allows dealing with the issue by using Machine Learning to obtain an approximate model for the mapping-to-efficiency relation. In [3], this is done by means of ANNs that are then embedded in a CP model via Neuron Constraints.

[1] Of course the propagation is likely to be less effective for more complex networks.

This example is particularly important in the paper because we target the same problem, and the same Machine Learning and optimization techniques. In order to provide the necessary background, in this section we discuss the basics of ANNs and recall how they have been employed for modeling purpose in optimization approaches. We then (briefly) point out what sets EML apart from such methods, and finally we describe how to embed an ANN into a CP model by means of Neuron Constraints.

Artificial neural networks An ANN is a system emulating the behavior of a biological network of neurons. Each ANN unit (artificial neuron) corresponds to a function in the form:

$$z = f\left(b + \sum_{i=0}^{n-1} w_i\, x_i\right) \tag{2}$$

where x_i are the neuron inputs, w_i are their weights, b is called the bias and z is the neuron output. All the terms are $\in \mathbb{R}$. Besides, $f : \mathbb{R} \rightarrow \mathbb{R}$ is called *activation function* and it is always monotone non-decreasing. Some examples of activation functions follow:

$$(\mathbf{a})\ f(x) = x \quad (\mathbf{b})\ f(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ -1 \text{ otherwise} \end{cases} \quad (\mathbf{c})\ f(x) = \tanh(x) \tag{3}$$

Case (a), (b) and (c) correspond to a linear, step and (bipolar) sigmoid neuron.

The neurons are connected in a network structure, a very common case being a feed-forward (i.e. acyclic), two-layer, graph [4, 36]. The first layer is referred to as *hidden layer*, the second is called the *output layer*, because it provides the network output. Figure 1 shows an example of such a network. Each node represents a neuron, the weights are reported as labels on the arcs, the x_i are the network inputs, and o is the network output. The activation function for each neuron (a letter code referring to (3)) is written inside the node.

An ANN can be trained to approximate a complex mathematical relation for which a set of input-output pairs (i.e. a *training set*) is known. This is done by solving an optimization problem where the goal is to choose the network weights so as to minimize the average square error between the predicted and real output values on the training set.

There are many specifically designed, readily available, training algorithms [5, 24, 28]. Most of them are heuristic and require to have *differentiable and invertible* activation functions. Additionally, *non-linear activation functions are always employed for the hidden neurons*: if this is not the case, then it is possible (by algebraic manipulation) to transform the two-layer ANN into an equivalent single-layer ANN. Finally, all the activation functions classically employed in the hidden layer have *a single flex point at 0*, i.e. they are convex when $x \leq 0$ and concave afterwards. Sigmoid neurons satisfy all those properties and are a very popular choice for the hidden layer.

ANNs in optimization methods The idea of learning a system model from data is well established in Control Theory, where it is referred to as System Identification [26]. In such

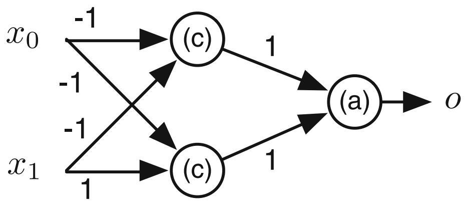

Fig. 1 A two-layer, feed forward ANN

field, the goal is to learn a dynamic model to be used by an on-line controller. Most of the systems modeled via this approach are linear, but there are important examples where ANNs are employed as on-line predictors, with their parameters being continuously adjusted based on the prediction error [15]. ANNs have been used as cheap-to-compute cost functions in metaheuristics: in [11] a Genetic Algorithm exploits an ANN to estimate the performance of an absorption chiller. Work [29] proposes a custom heuristic for workload dispatching in a data center, using an ANN for temperature estimation. A few works, such as [23], have employed ANNs for solution checking. In the OptQuest metaheuristic system [17], an Artificial Neural Network is trained during search with the aim to avoid trivially bad solutions. Other approaches have used ANNs as *surrogate system models*, to avoid repeated calls to a simulator: in [18], this is done to estimate the condition of road pavement. An excellent overview about surrogate models (including a few examples based on ANNs) can be found in [31].

As a common trait, in all the mentioned approaches the ANN is exploited in a rather limited fashion, namely as a black-box function evaluator. Conversely, EML *stresses the importance of reasoning on the network structure and weights to improve the performance of the optimization process*. Additionally, to the best of our knowledge, all the mentioned optimization techniques are designed for problems lacking a complex combinatorial structure and without complex constraints: this marks a profound difference with Empirical Model Learning, which focuses exactly on problems with such characteristics.

Embedding ANNs in CP via neuron constraints For a CP solution approach, performing filtering and propagation is the most natural way to reason on a network and boost the search process. In [2] this is achieved by means of Neuron Constraints. A Neuron Constraint is a global constraint that encodes a single network neuron and enforces consistency on (2). It is equivalent to the following pair of constraints:

$$(\mathbf{c_0}) \quad z = f(y) \qquad (\mathbf{c_1}) \quad y = b + \sum_{i=0}^{n-1} w_i x_i \tag{4}$$

where z, y and x_i are real-valued decision variables[2] with interval domain. Hence, we have $z \in [\underline{z}, \overline{z}]$, $y \in [\underline{y}, \overline{y}]$ and $x_i \in [\underline{x}_i, \overline{x}_i]$, where the underlined and overlined notation refers respectively to the domain minimum and maximum. The term y is called the neuron *activity*.

Bound Consistency algorithms for the linear constraint (c_1) are readily available. Defining a propagator for (c_0) is actually easy, since the f function is always monotone and monovariate. Bound Consistency on the domain maxima can therefore be enforced by the following rules:

$$\overline{y} \text{ changes} \xrightarrow{triggers} \overline{z} = \min(\overline{z}, f(\overline{y})) \tag{5}$$

$$\overline{z} \text{ changes} \xrightarrow{triggers} \overline{y} = \min(\overline{y}, f^{-1}(\overline{z})) \tag{6}$$

the rules for filtering the minima are analogous. The computation of f^{-1} requires some extra care to handle numerical inaccuracies due to the finite precision employed by all physical computers (more details can be found in [2]). For a single neuron, this approach leads

[2]Real-valued variables with fixed precision can be modeled via integer variables: e.g. a number in [0, 1] with precision 0.01 corresponds to a number $\in \{0..100\}$. This representation requires some care to ensure consistent rounding. More details can be found in [2].

to the tightest possible bounds on all the variables, i.e. it is sufficient to enforce Bound Consistency.

It is possible to embed an ANN in a CP model by building a Neuron Constraint for each network neuron and by introducing decision variables to represent the input/output of each neuron. Each Neuron Constraint can be implemented either as a single entity or as an actual pair of constraints.

3 A global constraint for two-layer feed-forward ANNs

Using individual constraints to encode each neuron has two advantages. First, the approach allows embedding basically any type of ANN into a CP model. Second, propagating individual neurons is quite easy, as shown in Section 2. As a main drawback, the decomposition may lead to poor overall propagation.

As a *motivating example*, consider the network from Fig. 1. Let us assume that $x_0, x_1 \in]-1, +1[$ and that we are interested in computing an upper bound for the network output o. Since the output neuron is linear, the bound can be computed easily by summing the lower/upper bounds on the inputs, depending on the sign of their weights. In this case we have:

$$ub(o) = \tanh(ub(y_0)) + \tanh(ub(y_1)) \tag{7}$$

where y_0 and y_1 are the activities of the two hidden neurons, and we have exploited the fact that tanh is a monotone function. The activities are linear expressions, hence the upper bounds $ub(y_0)$ and $ub(y_1)$ are given by:

$$ub(y_0) = -\min(x_0) - \min(x_1) = 2 \tag{8}$$
$$ub(y_1) = -\min(x_0) + \max(x_1) = 2 \tag{9}$$

and therefore $ub(o) \simeq 1.928$. Unfortunately, the actual network maximum with the described input ranges is $\simeq 1.515$, meaning that our bound is quite loose. The cause of the lack of tightness is having allowed conflicting values for x_1 in (8) and (9). *Unfortunately, avoiding this miscalculation is impossible as long as we reason on each neuron individually*.

It makes therefore sense to use global constraints to capture whole networks, or frequently occurring network sub-structures. In this paper we consider the case of two-layer, feed-forward, ANNs, by introducing a family of global constraints in the form:

$$\mathtt{ann}_{\langle f,g\rangle}(x, o, w, b, \hat{w}, \hat{b}) \tag{10}$$

where f is the activation function for the hidden layer (assumed to be differentiable, invertible, and with a single flex point – see Section 2) and g is the activation function for the output layer: together they define a specific constraint within the family. The role of all the other constraint parameters is described visually in Fig. 2, which is based on the example network from Fig. 1. In detail, x is the vector of variables x_i representing the network input, and o is the variable for the network output. The vector w contains all weights $w_{j,i}$ of each input i for the hidden neuron j. The vector b contains the bias value b_j for all the hidden neurons. The vector $\hat{w}$ contains the weights $\hat{w}_j$ of the outputs of the j-th hidden neurons, when they are used as input for the output neuron. The scalar $\hat{b}$ is the bias for the output neuron. For sake of simplicity we assume to deal with single output networks, but our methods can be generalized to multiple outputs.

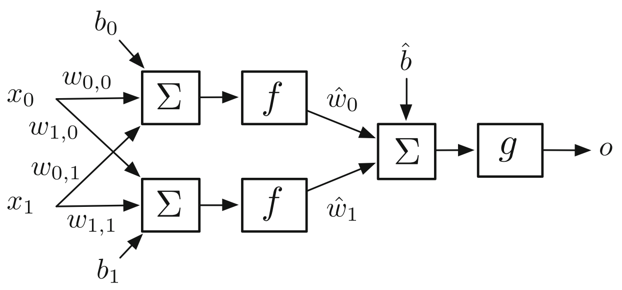

Fig. 2 Our notation for two-layer Artificial Neural Networks

Two-layer, feed-forward ANNs are very frequently employed in practice. Networks with multiple layers can be encoded via multiple instances of Constraint (10). Finally, by combining Constraint (10) with the Neuron Constraints from [2] it is possible to encode virtually any type of ANN.

An instantiation of Constraint (10) should enforce consistency on the set of all neuron equations in the network. Unfortunately, since non-linear neurons are employed in the hidden layer in all practical cases, even bound consistency cannot be enforced in polynomial time. Therefore, any practically relevant propagator should rely on some kind of relaxation. In particular, we employ two relaxations: (i) we reason on each neuron individually, and (ii) we employ *a non-linear Lagrangian relaxation to compute possibly tighter bounds on the network input and output*. In other words, there are two propagators *simultaneously* associated to Constraint (10): the first is the Neuron Constraint filtering algorithm from [2], the second is the new Lagrangian-based approach that we presented in [27] and is extended in this paper.

3.1 Computing bounds for the network output

We now consider the problem of finding an upper bound on the network output. Since all activation functions are monotone, we can focus on computing bounds for the activity of the output neuron (let it be z) rather than on the output itself (i.e. o). This also implies that different g functions in Constraint (10) can be handled with minor changes to our method. Formally, computing an upper bound requires to solve:

$$\textbf{P0}: \quad \max\ z = \hat{b} + \sum_{j=0}^{m-1} \hat{w}_j f(y_j) \tag{11}$$

$$\text{subject to:} \quad y_j = b_j + \sum_{i=0}^{n-1} w_{j,i} x_i \qquad \forall j = 0..m-1 \tag{12}$$

$$x_i \in \left[\underline{x}_i, \overline{x}_i\right] \qquad \forall i = 0..n-1 \tag{13}$$

where y_j is the activity of the j-th hidden neuron, and all other variables and parameters are as in Fig. 2. A lower bound can be obtained by changing the optimization direction. As already mentioned, problem PO is NP-hard.

Problem relaxation We employ a Lagrangian relaxation to obtain a scalable bounding procedure. Specifically, we relax Constraints (12), obtaining:

$$\mathbf{LP0}(\lambda): \quad \max_{x,y} z(\lambda) = \hat{b} + \sum_{j=0}^{m-1} \hat{w}_j f(y_j) + \tag{14}$$

$$+ \sum_{j=0}^{m-1} \lambda_j \left(b_j + \sum_{i=0}^{n-1} w_{j,i} x_i - y_j \right)$$

$$\text{subject to:} \quad x_i \in \left[\underline{x}_i, \overline{x}_i\right] \qquad \forall i = 0..n-1 \tag{15}$$

$$y_j \in \left[\underline{y}_j, \overline{y}_j\right] \qquad \forall j = 0..m-1 \tag{16}$$

where λ is a vector of Lagrangian multipliers λ_j (real numbers), acting as parameters for the relaxation. The notations x and y refer to the vectors of the x_i and y_j variables. Constraints (16) have been added to prevent LP0(λ) from becoming unbounded, and the values $\underline{y}_j$ and $\overline{y}_j$ are chosen so that the constraints are redundant in the original problem. In particular we have:

$$\underline{y}_j = b_j + \sum_i \begin{cases} w_{j,i}\, \underline{x}_i & \text{if } w_{j,i} \geq 0 \\ w_{j,i}\, \overline{x}_i & \text{otherwise} \end{cases} \tag{17}$$

and the value $\overline{y}_j$ is computed similarly.

Now, since problem LP0(λ) is a relaxation, its feasible space includes that of P0. Additionally, for every possible λ, on all points where Constraints (12) are satisfied we have $z = z(\lambda)$. Therefore, the set of solutions of LP0(λ) contains all the solutions of P0, *with the same objective value*. As a consequence, the optimal solution $z^*(\lambda)$ of LP0(λ) is always at least as good as that of P0, i.e. $z^*(\lambda)$ is a valid upper bound for z^*.

If all the λ multipliers are zero, then (14) becomes a sum of monotone functions and LP0(λ) can be solved by either minimizing or maximizing y_j based on the sign of $\hat{w}_j$. *This is the same bound computation performed by individual Neuron Constraints*. If some λ_j is non-zero, then the Lagrangian bound may be stronger or weaker, depending on the multiplier values. Due to some of our overhead reduction techniques, we cannot guarantee that the case $\lambda = 0$ is considered at each search node: this is one of the reasons for employing the original propagation from [2] along with the Lagrangian relaxation.

Solving the relaxation Problem LP0(λ) can be decomposed into two independent subproblems LP1(λ) and LP2(λ), defined respectively on the x and y variables and such that:

$$z^*(\lambda) = \hat{b} + \sum_{j=0}^{m-1} \lambda_j b_j + z^*_{LP1}(\lambda) + z^*_{LP2}(\lambda) \tag{18}$$

with:

$$\mathbf{LP1}(\lambda): \quad z^*_{LP1}(\lambda) = \max_x \; z_{LP1}(\lambda) = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{m-1} \lambda_j w_{j,i} \right) x_i \tag{19}$$

$$\text{s.t.} \quad x_i \in \left[\underline{x}_i, \overline{x}_i\right] \qquad \forall i = 0..n-1 \tag{20}$$

and:

$$\textbf{LP2}(\lambda): \quad z^*_{LP2}(\lambda) = \max_y \; z_{LP2}(\lambda) = \sum_{j=0}^{m-1} \left(\hat{w}_j f(y_j) - \lambda_j y_j\right) \tag{21}$$

$$\text{s.t.} \quad y_j \in \left[\underline{y}_j, \overline{y}_j\right] \qquad \forall j = 0..m-1 \tag{22}$$

Since the two subproblems are independent, they can be addressed separately.

Solving $LP1(\lambda)$ Each x_i appears in the objective of LP1(λ) with weight:

$$\tilde{w}_i(\lambda) = \sum_{j=0}^{m-1} \lambda_j w_{j,i} \tag{23}$$

Therefore, the problem can be solved by assigning each x_i either to $\underline{x}_i$ or to $\overline{x}_i$, depending on the sign of $\tilde{w}_i(\lambda)$. In detail:

$$z^*_{LP1}(\lambda) = \sum_{i=0}^{n-1} \begin{cases} \tilde{w}_i(\lambda)\,\overline{x}_i \text{ if } \tilde{w}_i(\lambda) \geq 0 \\ \tilde{w}_i(\lambda)\,\underline{x}_i \text{ otherwise} \end{cases} \tag{24}$$

The process has time complexity $\Theta(nm)$, due to the weight computation step. If a single multiplier changes, it is possible to update all weights in $\Theta(n)$ and solve LP1(λ) with the same complexity level. If all the multipliers change at the same time (as it is the case in this paper), no simple incremental technique can be applied to reduce the complexity.

Solving $LP2(\lambda)$: Problem LP2(λ) can be further decomposed into a sum of single-variable maximization problems over a finite feasible interval:

$$\max_{y_j} \; h_j(y_j, \lambda) = \hat{w}_j f(y_j) - \lambda_j y_j \tag{25}$$

$$\text{s.t. } y_j \in \left[\underline{y}_j, \overline{y}_j\right] \tag{26}$$

Each single-variable subproblem can be tackled via classical analytical methods. In particular the maximum feasible value of $h_j(y_j, \lambda)$ will be reached at either (i) a local maximum point, provided it falls within the feasible interval, or (ii) at the boundaries of the feasible interval, i.e. $\underline{y}_j$ and $\overline{y}_j$.

The value of $h_j\left(y_j, \lambda\right)$ at $\underline{y}_j$ and $\overline{y}_j$ is immediate to compute. The position of all local maxima can be found by checking all the stationary points, i.e.the points where the derivative of $h_j(y_j, \lambda)$ is null:

$$\frac{d\,h_j(y_j, \lambda)}{d\,y_j} = \hat{w}_j \frac{d\,f}{d\,y_j}(y_j) - \lambda_j = 0 \tag{27}$$

where we recall from Section 2 that f can be assumed to be differentiable. If $\hat{w}_j = 0$ and $\lambda_j \neq 0$, then no stationary point exists: $h_j(y_j, \lambda)$ reaches a maximum on one of the interval boundaries. If $\hat{w}_j = 0$ and $\lambda_j = 0$, then all points are stationary: $h_j(y_j, \lambda)$ is constant and its maximum/minimum value can be computed at the interval boundaries. If $\hat{w}_j \neq 0$, we have:

$$\frac{d\,f}{d\,y_j}(y_j) = \frac{\lambda_j}{\hat{w}_j} \tag{28}$$

Since f is non-linear in all practical cases, (28) may be in principle hard to solve. Luckily, we recall from Section 2 that in all classical cases f has a single flex point at 0. Hence, the derivative of f will be increasing for $y_j \leq 0$, and decreasing afterwards.

This behavior can be observed in Fig. 3, showing (as an example) the value of the derivative for a tanh activation function. From a graphical perspective, solving (28) amounts to finding the intersections between a function with shape similar to that in Fig. 3 and a horizontal line. It can be seen that there can be either no intersection at all, or exactly two intersections.

Therefore, if f has a single flex point, then (28) has either no solution or exactly two solutions, one before 0 and the other afterwards. It is always possible to find such solutions with arbitrary precision via bisection over the half intervals $]\underline{y_j}, 0]$ and $]0, \overline{y_j}[$, although the process may be inefficient.

Solving (28) with sigmoid neurons In many practical cases it is possible to obtain a closed-form solution for (28). In this paper we focus on the case where f is a bipolar sigmoid (i.e. tanh), but most of the activation functions typically employed in ANNs can be tackled with minor adaptations. If $f = \tanh$, we have that:

$$\tanh(x) = \frac{2}{1 - e^{-2x}} - 1 \qquad\qquad \frac{d\ \tanh}{dx}(x) = \frac{4e^{-2x}}{\left(1 + e^{-2x}\right)^2} \tag{29}$$

where the derivative is precisely the function from Fig. 3, which takes values in the range $]0, 1]$. Therefore, (28) has no solution if λ_j is zero, or ${}^{\lambda_j}/_{\hat{w}_j}$ is negative, or ${}^{\lambda_j}/_{\hat{w}_j}$ is greater than one. If none of such conditions holds, then we can combine (29) and (28) to obtain:

$$\frac{4e^{-2y_j}}{\left(1 + e^{-2y_j}\right)^2} = \frac{\lambda_j}{\hat{w}_j} \tag{30}$$

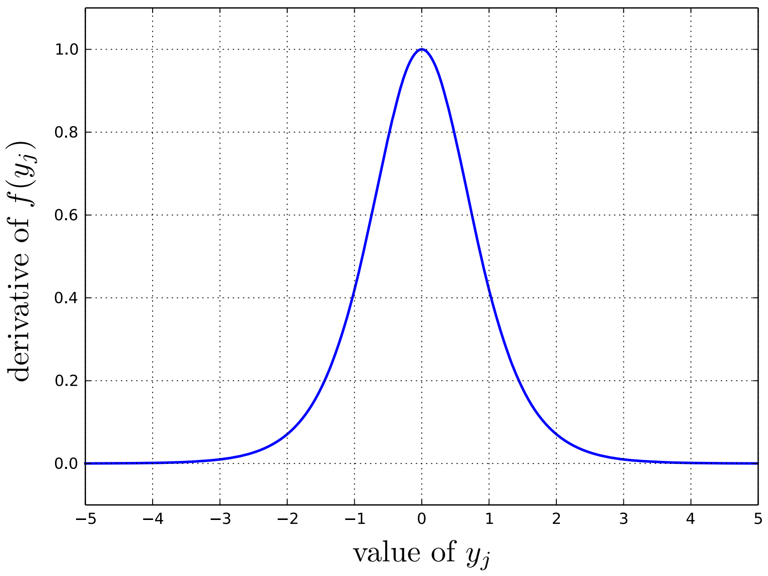

Fig. 3 Value of $\frac{d\,f(y_j)}{d\,y_j}$ for a tanh activation function

where $\left(1+e^{-2y_j}\right)^2$ is always greater than zero. Then, by substituting $u = e^{-2y_j}$ and performing algebraic transformations, we get:

$$u^2 + \left(2 - 4\frac{\hat{w}_j}{\lambda_j}\right)u + 1 = 0 \tag{31}$$

Which can be solved via the classic quadratic formula for second degree equations, yielding two solutions u' and u''. The solutions are non-complex iff:

$$\left(2 - 4\frac{\hat{w}_j}{\lambda_j}\right)^2 - 4 \geq 0 \quad \Leftrightarrow \quad \frac{\hat{w}_j}{\lambda_j}\left(\frac{\hat{w}_j}{\lambda_j} - 1\right) \geq 0 \tag{32}$$

Inequality (32) always holds if the conditions to have a solution are true, because in that case ${}^{\hat{w}_j}/_{\lambda_j}$ is non-negative and greater than or equal to 1. The same conditions ensure also that u' and u'' are both positive. Once u' and u'' are known, we can obtain the y_j values corresponding to the stationary points:

$$y_j' = -\frac{1}{2}\log u', \quad y_j'' = -\frac{1}{2}\log u'' \tag{33}$$

In detail, one point will correspond to a local minimum of $h(y_j, \lambda)$ and the other to a local maximum.

In summary For a certain assignment of the multipliers λ, the Lagrangian relaxation LP0(λ) can be solved by:

1. Finding the optimal values $x^*(\lambda)$ for all x variables (complexity $\Theta(nm)$)
2. Using the $x^*(\lambda)$ values to compute $z^*_{LP1}(\lambda)$ (complexity $\Theta(n)$)
3. Finding the optimal values $y^*(\lambda)$ for all the y variables, which is done by:
    (a) Computing $\underline{y}_j$ and $\overline{y}_j$ (complexity $\Theta(n)$)
    (b) Evaluating $h_j(y_j, \lambda)$ at $\underline{y}_j$ and $\overline{y}_j$ (complexity $O(1)$)
    (c) If the conditions for the existence of the two stationary points apply, evaluating $h_j(y_j, \lambda)$ at y_j' and y_j'' (complexity depending on f)
4. Using the $y^*(\lambda)$ values to compute $Z_{LP2}(\lambda)$
5. Computing the bound using (18) (complexity $\Theta(m)$)

Algorithm 1 shows the detailed process when the hidden neurons use a bipolar sigmoid activation function, i.e. f is tanh. In such case and in any case where (28) has a closed form solution, step 4(c) has complexity $O(1)$, hence the bound computation is done in $\Theta(nm)$, which is the same complexity of propagating each neuron equation individually.

Algorithm 1 Computing an upper bound on z by solving LP0(λ)

```
1: initialize $z^*(\lambda) = \hat{b} + \sum_{j=0}^{m-1} \lambda_j b_j$
2: compute all $\tilde{w}_i(\lambda)$
3: for $i = 0..n-1$ do
4:     set $x_i^*(\lambda) = \begin{cases} \overline{x}_i \text{ if } \tilde{w}_i(\lambda) \geq 0 \\ \underline{x}_i \text{ otherwise} \end{cases}$
5:     update $z^*(\lambda) = z^*(\lambda) + \tilde{w}_i(\lambda) x_i^*(\lambda)$
6: for $j = 0..m-1$ do
7:     compute $\underline{y}_j$ and $\overline{y}_j$
8:     set $y_j^*(\lambda) = \text{argmax}\{h_j(\underline{y}_j, \lambda), h_j(\overline{y}_j, \lambda)\}$
9:     if $\hat{w}_j \neq 0$ and $\lambda_j \neq 0$ and ${}^{\lambda_j}/_{\hat{w}_j} \in ]0,1[$ then
10:        set $y'_j, y''_j = -\frac{1}{2} \log\left(\frac{-\beta \pm \sqrt{\beta^2 - 4}}{2}\right)$, with $\beta = 2 - 4\frac{\hat{w}_j}{\lambda_j}$
11:        if $y'_j \in ]\underline{y}_j, \overline{y}_j[$ and $h_j(y'_j, \lambda_j) > h_j(y_j^*(\lambda), \lambda_j)$ then $y_j^*(\lambda) = y'_j$
12:        if $y''_j \in ]\underline{y}_j, \overline{y}_j[$ and $h_j(y''_j, \lambda_j) > h_j(y_j^*(\lambda), \lambda_j)$ then $y_j^*(\lambda) = y''_j$
13:     set $z^*(\lambda) = z^*(\lambda) + h_j(y_j^*(\lambda), \lambda)$
14: return $z^*(\lambda)$
```

3.2 Optimizing the lagrangian multipliers

Any assignment of the multipliers λ yields a different bound on the z variable. Hence it is possible to improve the bound quality by optimizing the multiplier values, i.e. by solving the following unconstrained minimization problem:

$$\textbf{L0}: \quad \min_{\lambda} z^*(\lambda) \tag{34}$$

Where $z^*(\lambda)$ is here a function that denotes the optimal solution of LP0(λ). Problem L0 is convex in λ and hence has a unique minimum. This is true even if LP0(λ) is non-convex: in fact, the two problems are defined on different variables (i.e. λ versus x and y). The minimum point can therefore be found via a descent method.

Now, let λ' be an assignment of λ and let $x^*(\lambda')$ and $y^*(\lambda')$ be the values of x and y in the corresponding solution of LP0(λ). If such solution stays constant for small changes of λ', then $z^*(\lambda)$ is differentiable at λ' and the j-th component of the gradient is given by:

$$s_j \equiv b_j + \sum_{i=0}^{n-1} w_{j,i} x_i^*(\lambda') - y_j^*(\lambda') \tag{35}$$

The expression s_j is obtained by differentiating the objective of LP0(λ) under the above mentioned assumptions.

If the solution of LP0(λ) is not stable for small changes of λ', then s (i.e. the vector of all s_j) is still a valid *subgradient*. The optimum of L0 can therefore be found via a subgradient method [25], by starting from an assignment $\lambda^{(0)}$ and iteratively applying the update rule:

$$\lambda^{(k+1)} = \lambda^{(k)} - \sigma^{(k)} s^{(k)} \tag{36}$$

where $\lambda^{(k)}$ denotes the multipliers for the k-th step, $s^{(k)}$ is the subgradient, and $\sigma^{(k)}$ is a scalar representing a step length.

Step update policy In subgradient methods, the step length must be dynamically updated for the process to converge. In this paper, we perform the updates according to the corrected

Polyak policy with non-vanishing threshold from [13]. This guarantees the convergence *with bounded error* to the optimal multipliers. Other policies from the literature are more accurate, but have a slower convergence rate, which is in our case *the* critical parameter (since we run the subgradient method within a propagator). In detail, we choose $\sigma^{(k)}$ as follows:

$$\sigma^{(k)} = \beta \frac{z^*\left(\lambda^{(k)}\right) - \left(z^{best} - \delta^{(k)}\right)}{\|s^{(k)}\|^2} \tag{37}$$

where β is a scalar parameter in]0, 2[. The term $z^{best} - \delta^{(k)}$ is an estimate for the optimum of L0: it is computed as the difference between the lowest bound found so far (referred to as z^{best}) and a scalar $\delta^{(k)}$, which is dynamically adjusted during the process. Hence, the step size is directly proportional to the distance of the current bound $z^*(\lambda^{(k)})$ from the estimated optimum, i.e. $z^{best} - \delta^{(k)}$. Larger $\delta^{(k)}$ values lead to larger step sizes.

The value of $\delta^{(k)}$ is non-vanishing, namely it is constrained to be larger than a threshold δ^*. This ensures to have always positive step sizes and prevents the subgradient optimization from getting stuck. We determine the δ^* value based on the multipliers from the first subgradient iteration (let them be $\lambda^{(0)}$). Specifically, we choose $\delta^* = \gamma z^*(\lambda^{(0)})$, with γ being a small, positive, scalar parameter. During search, we compute $\delta^{(k)}$ according to the following rules:

$$\delta^{(k+1)} = \begin{cases} \max(\delta^*, \nu\delta^{(k)}) & \text{if } z^*(\lambda^{(k)}) > z^{best} - \delta^{(k)} \\ \max(\delta^*, \mu z^*(\lambda^{(k)})) & \text{otherwise} \end{cases} \tag{38}$$

where ν, μ are again scalar parameters, ranging in]0, 1[. Informally speaking, if the last computed bound $z^*(\lambda^{(k)})$ does not improve over the estimated optimum $z^{best} - \delta^{(k)}$, then we reduce the current $\delta^{(k)}$ value, i.e. we make the estimated optimum closer to z^{best}. Conversely, when an improvement is obtained, the freshly computed bound becomes the new z^{best} and we "reset" $\delta^{(k)}$: this is done by choosing $\delta^{(k)}$ so that the estimated optimum is $\mu\%$ lower than the current best bound. Since the Polyak non-vanishing policy provides no termination criterion, we stop the subgradient optimization when a maximum number of iterations is reached.

Deflection Subgradient methods are known to exhibit a zig-zag behavior when close to an area where the cost function is non-differentiable. In this situation the convergence rate can be improved via deflection techniques [13]. In its most basic form (the one we adopt), a deflection technique consists in replacing the subgradient with the following vector in (36) and (37):

$$d^{(k)} = \alpha s^{(k)} + (1 - \alpha)d^{(k-1)} \tag{39}$$

where $d^{(k)}$ is the new search direction. The α parameter is a scalar in]0, 1], hence $d^{(k)}$ is a convex combination of the last search direction and the current subgradient. When using the deflection technique, the value β from (37) must be less than or equal to α for the method to converge.

The components s_j having alternating sign in consecutive gradients (i.e. such that $s_j^{(k)} s_j^{(k-1)} < 0$) tend to cancel each other in the deflected search direction. This behavior can be observed in Fig. 4. The figure depicts the bound value as a function of λ for the network from Fig. 1, together with the trace of the first 10 subgradient iterations. The use of deflection allows to get considerably closer to the best possible bound. Specifically, in 10 iterations we reach $\simeq 1.523$ against a true optimum of $\simeq 1.515$. As a comparison, the bound obtained by reasoninig on individual constraints was $\simeq 1.928$.

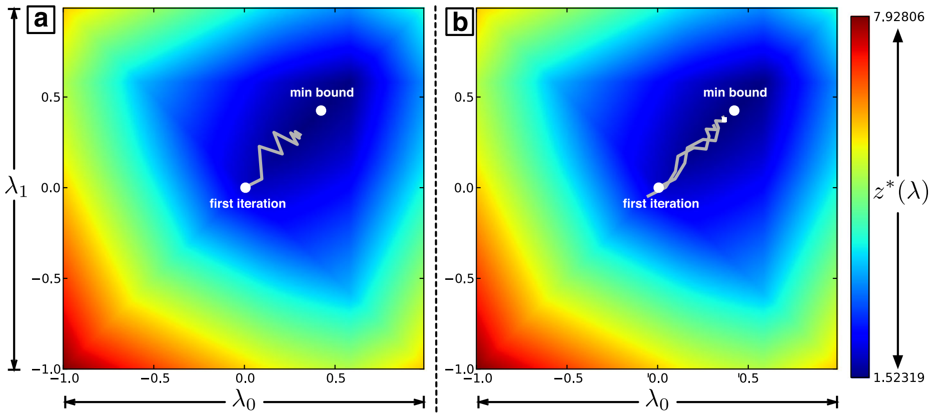

Fig. 4 **a** Subgradient optimization trace (10 iterations, no deflection). **b** Subgradient optimization trace (10 iterations, with deflection)

3.3 Computing bounds for the network input

In this section, we investigate the use of the Lagrangian relaxation to obtain bounds on the x and y variables, via a form of reduced-cost based filtering [14]. We will show that the technique is viable and efficient for the x variables (i.e. the ANN input), but too computationally expensive for the y variables.

We start by assuming to have access to a collection of functions $z^*_{x_i}(\lambda, x_i)$ and $z^*_{y_j}(\lambda, y_j)$, which specify how the Lagrangian (upper) bound for z depends on a single x_i or y_j variable. For sake of simplicity and with an abuse of notation, we will refer to such functions as $z^*(\lambda, x_i)$ and $z^*(\lambda, y_j)$.

The functions can be used to filter the x and y domains. Namely, we can: (i) compute the largest and lowest values of x_i that maintain feasibility given the domain of z; and then (ii) update $\overline{x}_i$ and $\underline{x}_i$ accordingly. Formally:

$$\overline{x}_i = \min\left(\overline{x}_i, \max\{x_i \ : \ z^*(\lambda, x_i) \geq \underline{z}\}\right) \tag{40}$$

$$\underline{x}_i = \max\left(\underline{x}_i, \min\{x_i \ : \ z^*(\lambda, x_i) \geq \underline{z}\}\right) \tag{41}$$

and similarly for $\overline{y}_j$, and $\underline{y}_i$. Other (possibly tighter) bounds can be obtained by reasoning on the Lagrangian *lower* bound on z. For sake of simplicity, here we limit the discussion to the filtering based on the upper bound.

The two functions $z^*(\lambda, x_i)$ and $z^*(\lambda, y_j)$ are always theoretically well defined, but they may be expensive to compute. Here we attempt to determine their complexity by relying on the following rewritten form of LP0(λ):

$$\max_{x,y} \ z(\lambda) = \hat{b} + \sum_{j=0}^{m-1} \lambda_j b_j + \sum_{i=0}^{n-1} \tilde{w}_i(\lambda) x_i + \sum_{j=0}^{m-1} h_j(y_j, \lambda) \tag{42}$$

$$\text{s.t.:} \qquad x_i \in \left[\underline{x}_i, \overline{x}_i\right] \qquad \forall i = 0..n-1 \tag{43}$$

$$y_j \in \left[\underline{y}_j, \overline{y}_j\right] \qquad \forall j = 0..m-1 \tag{44}$$

where the weights $\tilde{w}_i(\lambda)$ are as defined in (23) and the functions $h_j(y_j, \lambda)$ are as defined in (25). The values $\underline{y}_j$ and $\overline{y}_j$ are treated here as constant, despite they actually depend on the domain of the x variables: the reason for this simplification will be given later.

As already observed, the relaxed problem has no constraint involving multiple variables. Moreover, each term of the objective function depends on a single x_i or y_j variable. Hence, a change in the value of a variable has no effect on the value of the other variables in the optimal solution.

Filtering the *x* variables: Problem LP0(λ) is linear in x, and a closed form expression for $z^*(\lambda, x_i)$ is given by:

$$z^*(\lambda, x_i) = z^*(\lambda) + \tilde{w}_i(\lambda) \left(x_i - x_i^*(\lambda)\right) \tag{45}$$

where we recall that $z^*(\lambda)$ is the cost of the optimal solution of the relaxation (i.e. the original Lagrangian upper bound), and $x_i^*(\lambda)$ is the value of x_i in such solution. By relying on (45), we can compute the value of x_i that makes the Lagrangian bound equal to $\underline{z}$, i.e.:

$$z^*(\lambda, x_i) = \underline{z} \tag{46}$$

which is linear and solved by $x_i = \frac{\underline{z}-z^*(\lambda)}{\tilde{w}_i(\lambda)} + x_i^*(\lambda)$ if $\tilde{w}_i(\lambda)$ is not equal to zero. Then, we can compute the maximal and minimal values of x_i that maintain the upper bound feasible:

$$\max\{x_i \ : \ z^*(\lambda, x_i) \geq \underline{z}\} = \begin{cases} \infty \text{ if } \tilde{w}_i(\lambda) \geq 0 \\ \frac{\underline{z}-z^*(\lambda)}{\tilde{w}_i(\lambda)} + x_i^*(\lambda) \text{ otherwise} \end{cases} \tag{47}$$

$$\min\{x_i \ : \ z^*(\lambda, x_i) \geq \underline{z}\} = \begin{cases} -\infty \text{ if } \tilde{w}_i(\lambda) \leq 0 \\ \frac{\underline{z}-z^*(\lambda)}{\tilde{w}_i(\lambda)} + x_i^*(\lambda) \text{ otherwise} \end{cases} \tag{48}$$

from (40) and (41), we can then obtain the following filtering rules:

$$\text{if } \tilde{w}_i(\lambda) < 0 : \overline{x}_i \ = \min\left(\overline{x}_i, \frac{\underline{z} - z^*(\lambda)}{\tilde{w}_i(\lambda)} + x_i^*(\lambda)\right) \tag{49}$$

$$\text{if } \tilde{w}_i(\lambda) > 0 : \underline{x}_i \ = \max\left(\underline{x}_i, \frac{\underline{z} - z^*(\lambda)}{\tilde{w}_i(\lambda)} + x_i^*(\lambda)\right) \tag{50}$$

Both rules have $O(1)$ complexity, hence all the x variables can be filtered in $\Theta(n)$. Similarly to other techniques based on Lagrangian relaxations (e.g. [10]), tighter values of the Lagrangian bound $z^*(\lambda)$ do not necessarily translate in tighter bounds obtained via reduced-cost based filtering. For this reason, it is a good idea to filter the x variables whenever the multipliers λ are updated.

Finally, the bounds from (49) and (50) are not guaranteed to be tighter than those obtained by propagating individual Neuron Constraints: this is our second reason for using the original approach from [2] along with the new Lagrangian-based techniques.

Filtering the *y* variables Filtering rules for each y_j can in principle be obtained via the same process. This may be of interest since, along with the Lagrangian relaxation, in our approach we rely on the original Neuron Constraint filtering, which may be capable of exploiting tighter bounds on y_j to filter the x variables.

In order to obtain the filtering rules, we first need to determine a closed form for $z^*(\lambda, y_j)$, which is given by:

$$z^*(\lambda, y_j) = z^*(\lambda) + h_j(y_j, \lambda) - h_j(y_j^*(\lambda), \lambda) \tag{51}$$

Then, we need to solve the equation $z^*(\lambda, y_j) = \underline{z}$, i.e.:

$$z^*(\lambda) + \hat{w}_j f(y_j) - \lambda_j y_j - h_j(y_j^*(\lambda), \lambda) = \underline{z} \tag{52}$$

where we have expanded $h_j(y_j, \lambda)$ according to its definition. Unfortunately, (52) contains both a linear term (i.e. $\lambda_j y_j$) and a non-linear, non-convex and non-concave term (i.e. $f(y_j)$). This kind of equation cannot be solved in general by analytical methods, not even in the case where f is a sigmoid or another classical neuron activation function. Numerical solution approaches (e.g. Newton-Raphson) are viable, but they have considerably higher computational complexity.

This may appear to be in contrast with our results from Section 3.1, but we recall that in such case we were interested in finding stationary points: hence, we were employing the *derivative* of $h_j(y_j, \lambda)$, which lacks a linear term.

Because of the higher complexity, *in this paper we have chosen to perform no reduced-cost filtering on the y variables*. This is also the reason for treating the values $\underline{y}_j$ and $\overline{y}_j$ as constant in this section: taking into account their dependence on the x variables would require to solve problems even harder than (52).

3.4 Overhead reduction techniques

The Lagrangian-based propagation techniques from Sections 3.1–3.3 can provide tighter bounds w.r.t. propagating individual Neuron Constraints. As a major drawback, the new techniques have increased computational complexity, mainly due to the necessity to optimize the Lagrangian multipliers. In this section we outline four techniques for reducing the computational cost of the new propagator.

Low priority propagation: The new propagator can be scheduled with the lowest possible priority in the constraint solver. Hence, if infeasibility is detected by the (cheaper) Neuron Constraint propagation from [2], the Lagrangian propagator is not even activated.

Limit the number of subgradient iterations: A simple approach to manage the computation time consists in controlling the number of subgradient iterations. Fewer iterations result in a smaller overhead, but may reduce the quality of the Lagrangian multipliers and the opportunities for filtering the x variables (see Section 3.3).

Multiplier caching and dynamic iteration limit: We can cache the two sets of multipliers that led respectively to the best lower/upper bound in the past. During branching, the multiplier values are passed from parent node to child, and they are restored when backtracking. Caching may allow to reduce the number of subgradient iterations during search, with limited adverse effects on the multiplier quality. The underlying rationale is that small updates of the network inputs (such as those occurring during search) are expected to result in small modifications of the optimal multipliers. As a special case it is possible to optimize the multipliers once and for all at the root node: this is a frequently employed approach for propagators based on Lagrangian relaxations (see e.g. [7, 10]).

Activation budget: The new propagator has the largest potential to save search time when it is used in the top nodes in the search tree. Based on this idea, we tried to limit the propagator action via an *activation budget*. This is done by introducing an activation counter, which is cached from parent to child node during search and restored when backtracking. Once the activation counter reaches a given limit, the Lagrangian approach is disabled and we perform only the baseline Neuron Constraint propagation.

4 Experimental Results

In this section, we discuss our experimental results. We focus on comparing the original Neuron Constraint from [2] with our new approach, where the individual neuron propagation is strengthened via Lagrangian-based bounds.

For the comparison to be fair, we want the two approaches to explore the same search space, which is ensured by solving the benchmark instances to optimality using a static search strategy. Despite being far from perfect (see [35]), this approach is very popular for comparing propagators in CP.

We do not compare our method to alternative approaches such as Local Search or Mixed-Integer Non-Linear Programming. A comparison of this kind would indeed be very interesting, but it is outside the scope of this paper.

In the following paragraphs, first, we present in Section 4.1 our application and our benchmark instances, then, we present our experimentation in three stages: in Section 4.2 we assess the quality of the Lagrangian bounds (on the ANN output) at the root of the search tree. During this stage we also performed a manual tuning (not discussed in detail) of the parameters related to the subgradient optimization process. In Section 4.3 we investigate the effect of the overhead reduction techniques and try to determine an optimal configuration. Finally, in Section 4.4 we report results for a larger-scale experimentation.

4.1 The target problem and the benchmark

Our benchmark consists in a number of instances of the thermal-aware job mapping problem presented in [3]. A number of jobs needs to be executed on a 48-core CPU by Intel called SCC (Single Chip Cloud Computer [20]), the research prototype that evolved into the current Xeon Phi. Each CPU core has a thermal controller, which reacts to overheating by reducing the operating frequency until the temperature is safe. The frequency reduction causes a loss of efficiency that depends: (i) on the workload of the core, (ii) on the workload of the neighboring cores, (iii) on the thermal physics, and (iv) on the policy of the controller itself.

The platform is simulated via an internally developed tool based on the popular Hotspot system [21] and tuned on a physical SCC platform. Using a simulator rather than the real chip allowed a research group cooperating with us to test advanced control policies that are outside the scope of this paper.

Job durations are not known. Therefore, in an attempt to balance the total workload duration, we require that each core runs a similar number of jobs. Without (great) loss of generality we assume the number of jobs n is a multiple of the number of cores m (in our case, $m = 48$), and that exactly the same number of jobs must be mapped to each core. The general case can be handled via the Weighted Average Constraint from [8].

The goal of our optimization problem is finding a job mapping that maximizes the lowest core efficiency. This permits to achieve good overall performance and avoid thermal hot-spots, i.e. excessively warm areas in the chip. We use a vector of integer variables p to model the task mapping, with $p_i = k$ iff task i is mapped to core k. We use a vector of continuous variables e to model the efficiency of each core, i.e. e_k gives the efficiency of the k-th core. Using these variables, we model the job mapping problem as follows:

$$\max_{p} \quad \min_{k=0..m-1} e_k \tag{53}$$

$$\text{s.t.} \quad \texttt{gcc}\left(p, [0..m-1], \frac{n}{m}\right) \tag{54}$$

$$\texttt{ann}_{\langle f,g \rangle}\left(x^k, e^k, w^k, b^k, \hat{w}^k, \hat{b}^k\right) \qquad \forall k = 0, \ldots, m-1 \tag{55}$$

$$p_i \in [0, \ldots, m-1] \qquad \forall i = 0, \ldots, n-1, \tag{56}$$

We use the gcc constraint to have exactly ${}^{n}/_{m}$ jobs per core. We use the ann constraint from Section 3 to encode m Artificial Neural Networks, each modeling how the efficiency e^k of core k depends on the platform workload x^k.

The artificial neural networks In our job mapping problem, each ANN has the following features: (i) it is two-layer and feed forward, (ii) it uses tanh as activation function at both the hidden and output layers, (iii) it has two hidden neurons, (iv) it has weights and bias values (i.e., vectors $w^k, b^k, \hat{w}^k, \hat{b}^k$) trained using the Levenberg-Marquardt algorithm [30]. Further details about the ANN training are out of the scope of this paper. All our networks are available for download.[3]

The ANNs inputs x^k in (55) are functions of the platform workload. In detail, each task i is characterized by a Clock Per Instruction (CPI) value cpi_i, measuring the degree of its CPU usage: lower CPI values correspond to more computation intensive (and heat generating) tasks, while higher CPI values are symptomatic of idle cycles due to frequent memory access. Each network has four, CPI based, inputs: (i) the average CPI of the jobs mapped on the considered core k, corresponding to a fresh $acpi_k$ variable; (ii) the average $acpi_k$ of the neighbouring cores, corresponding to a fresh $ncpi_k$ variable; (iii) the average $acpi_k$ on all the remaining platform cores, corresponding to a fresh $rcpi_k$ variable; (iv) the minimum CPI of the jobs mapped on core k, corresponding to a fresh $mcpi_k$ variable. Formally, using the platform workload, each vector x^k is composed by four elements:

$$x_0^k = acpi_k = \frac{1}{{}^{n}/_{m}} \sum_{i=0}^{n_t-1} cpi_i \, [[p_i = k]] \qquad \forall k = 0..m-1 \tag{57}$$

$$x_1^k = ncpi_k = \frac{1}{|N(k)|} \sum_{h \in N(k)} acpi_k \qquad \forall k = 0..m-1 \tag{58}$$

$$x_2^k = rcpi_k = \frac{1}{m - |N(k)| - 1} \sum_{\substack{h \notin N(k), \\ h \neq k}} acpi_k \qquad \forall k = 0..m-1 \tag{59}$$

$$x_3^k = mcpi_k = \min_{i=0..n-1} \left(cpi_i + M \, [[p_i \neq k]]\right) \qquad \forall k = 0..m-1 \tag{60}$$

[3] At https://bitbucket.org/m_lombardi/constraints-15-ann-lag-resources

where the set $N(k)$ contains the indices of the cores neighboring k, while M is the maximum cpi_i value (the choice is done so that jobs not mapped on core k are ignored in the computation of the minimum). The notation $[[cst]]$ stands for the reification of constraint cst. Even if not shown in the formulas (for sake of simplicity), each ANN input is normalized into the range $[-1, 1]$ via a linear transformation.

We have experimented with two sets of neural networks, which differ in their number of inputs, denoted *ANN1* and *ANN2*, respectively. *ANN1* has all the four variables just described, while *ANN2* lacks x_3^k, i.e., the $mcpi_k$ input. Both networks are practically relevant: ANN1 tends to be more precise than ANN2, but also more complicated to embed in optimization approaches. The two sets of networks have the same number of hidden neurons, but they are very different in terms of weights.

The benchmarks Due to the need to solve each instance to completeness, we had to focus on a restricted-size version of the original problem. In particular, we focus on optimizing a *subset* of the platform cores, assuming that the rest of the job mapping is fixed. Moreover, when optimizing a subset of cores, the efficiency of all the remaining cores is disregarded in the cost function. This setup is general enough to allow assessing the performance of our propagator: the only peculiarity is that in each benchmark instance only a subset of the 48 ANNs will be considered in the model.

Formally, let π be an initial mapping of all jobs, and in particular let $p_i(\pi)$ be the core where job i is mapped. Let R be the considered subset of cores. Then what we do is: (i) posting the additional constraint $p_i = p_i(\pi)$ for all jobs such that $p_i(\pi) \notin R$, and (ii) considering the modified cost function $\min_{k \in R} e_k$.

We obtained two distinct benchmarks via the same process. In particular, we start from a single fixed mapping of 288 jobs to the 48 cores on the platform. Then, we obtain different instances by selecting random subsets of three cores. Our first benchmark consists of 46 instances and is employed in Section 4.3 for assessing the impact of the overhead reduction techniques. The second benchmark consists of 200 instances and is employed for the last stage of our experimentation, in Section 4.4.

4.2 Bound tightness

We started our experimentation by evaluating the tightness of the Lagrangian bounds outside of the context of our benchmark problem. This was done by forcing all network inputs to range in $[-1, 1]$: since all inputs are assumed to be normalized, this is the same as leaving their domain unrestricted.

We compared the z bounds from Section 3.1 with real maxima/minima obtained via the non-linear (global) solver Couenne [6]. Obtaining the exact z range took usually a few seconds for each network (which unfortunately is still too much for a direct use within a constraint). The non-linear solver incurred in numerical issues in a few cases, which were fixed by making very small adjustments to the input ranges.

Subgradient optimization parameters During this experimentation we manually tuned most of the parameters of our subgradient optimization procedure. For the deflection technique we always use $\alpha = 0.5$ and we keep $\beta = \alpha$. The threshold parameter δ^* in the Polyak policy is re-initialized every time the constraint is triggered (hence only once during this experimentation), using $\gamma = 0.01$ and the bound computed in the first subgradient iteration. Therefore the bound estimate in the policy is always at least 1 % better than the Lagrangian bound from the first subgradient iteration. The attenuation factor ν, used when no improvement is obtained, is fixed to 0.75. The μ coefficient is 0.25 at the first constraint activation, which is the only one considered in this experimentation. In Section 4.3 and 4.4 however, we switch to using $\mu = 0.05$ after the first constraint activation, so as to enable finer updates. At the first constraint activation, all multipliers are initialized to zero. The tuning could be improved via an experimentation based on factorial design, or by relying on a tool such as ParamILS [22], but this is left as part of future work.

Results We experimented with different subgradient iteration limits. In all cases computing the Lagrangian bounds took a few milliseconds, with larger numbers of iterations leading to larger run-times. The comparison was performed for all the 48 networks in *ANN1* and *ANN2*. The results are displayed in Fig. 5, which reports bound values for the activity of the output neuron (i.e. the z variable from Section 3.1). For each network/core, we visualize the bounds obtained via different approaches as overlapping bars of different colors. As a baseline, we use the bounds obtained by reasoning on the neurons individually (white bars).

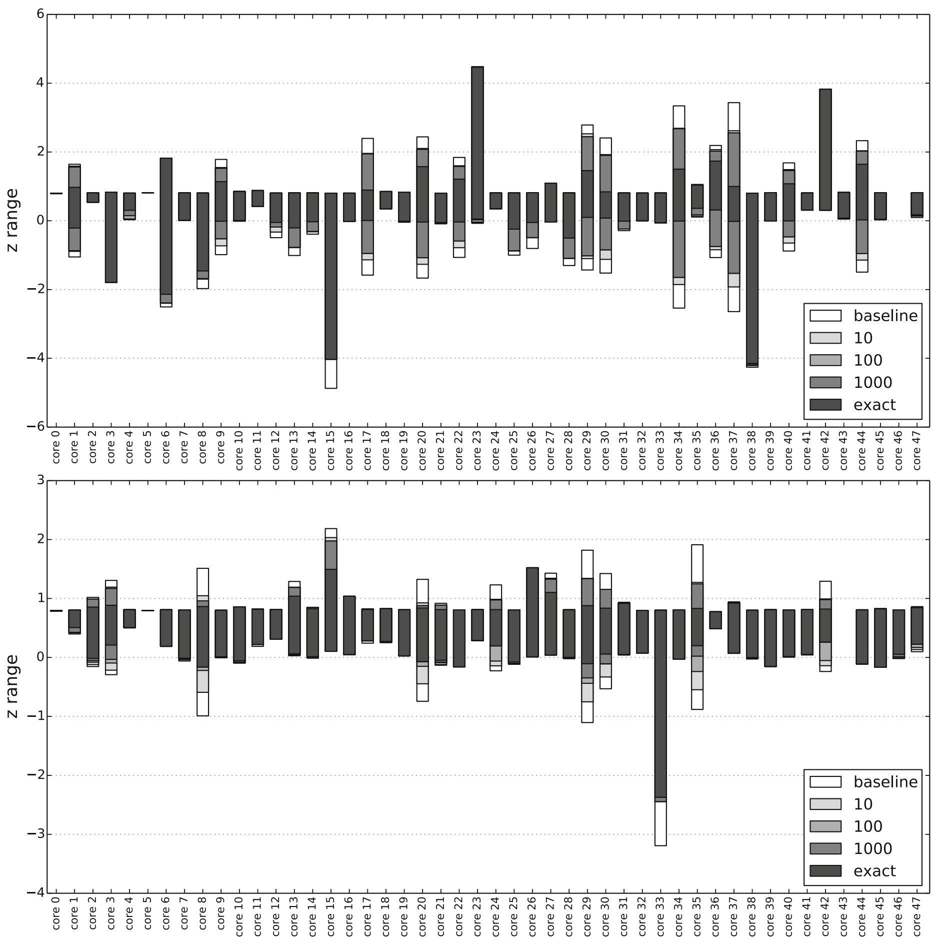

Fig. 5 Bound tightness for the neural networks in *ANN1* (*top*) and *ANN2* (*bottom*)

Darker colors refer to the Lagrangian bounds after 10, 100, and 1,000 subgradient iterations. An even darker shade of grey is used for the exact bounds. The z ranges for core 46 in *ANN1* and core 43 in *ANN2* were so large than displaying them would make the rest of the figure poorly readable: for this reason they have been omitted. In both cases, however, all the approaches (including the baseline) gave very similar bound values.

For many of the platform cores (20 cores in *ANN1*, 28 cores in *ANN2*) the baseline propagation is sufficient to provide tight bounds. When the baseline bounds are not tight, the Lagrangian relaxation does enable improvements, despite the improved bounds never match the exact z range. A limit of 100 iterations is typically sufficient to obtain the best Lagrangian bound on *ANN1*, while performing more iterations is sometimes worthwhile for *ANN2*.

4.3 Effect of the overhead reduction techniques

The next step of our experimentation is testing the new propagator in action (i.e. during search) and investigating the effectiveness of the overhead reduction techniques from Section 3.4. The experimentation was performed over the first of our benchmarks, containing 46 instances where a subset of 18 jobs must be mapped on a subset of 3 platform cores.

Our code is implemented on top of the Google or-tools solver[4] and all experiments are performed on an Intel Core i7, 2.3 GHz. A time limit of 300 seconds was enforced on all runs, but it was never exceeded. Our code, datasets, and results are available for download.[5]

Solution technique Each instances is solved using Depth First Search guided by an ad-hoc variable- and value-selection heuristic. Our search strategy is based on the idea that the average CPI on each core is inversely correlated with its efficiency level. Therefore, CPI-balanced solutions have a good chance to be efficiency-balanced as well. By steering search towards CPI-balanced mappings we are likely to find high quality solutions and boost the optimization process. Intuitively, our strategy always tries to map the most computation intensive job on the least loaded core. Formally, the process is as follows:

– We select for branching the (non-mapped) job i^* with lowest CPI.
– We compute for each core k the value of the expression:

$$\sum_{\substack{p_i \text{ bound,} \\ p_i = k}} (M - cpi_i) \tag{61}$$

where M is the largest possible CPI value. The expression has the following two properties:

1. It grows if the workload on k is computation intensive.
2. It grows with the number of jobs mapped on core k.

– Let k^* be the index of the core with the smallest value for Expression (61). On the left branch we post $p_{i^*} = k^*$, on backtracking we post $p_{i^*} \neq k^*$

This search strategy proved to work better than the classical min-size-domain rule on this problem.

[4]Google OR-tools, at https://developers.google.com/optimization/

[5]At https://bitbucket.org/m_lombardi/constraints-15-ann-lag-resources

The variables (i.e. the jobs) are considered by the new heuristic in a static order. The value (i.e. core) selection rule is not static in a strict sense, but it is based on past decisions rather than on the current variable domains. Hence, propagation cannot change the order in which the cores are considered. As a consequence, *we have the guarantee that more powerful propagation leads to smaller search trees*. This enables a fair comparison of different propagators: pruning a value has the effect of skipping part of the search tree, but does not affect the branching decisions in an unpredictable fashion.

Reference propagator configuration In this section we report results for an initial setup for the Lagrangian propagator, obtained by: (i) fixing the subgradient iteration limit to 100 for the first constraint activation; (ii) employing multiplier caching; and (iii) disabling multiplier updates (i.e. fixing the iteration limit to 0) for all subsequent constraint activations. In other words, we use the multipliers from the root node in the whole search tree, which is a common strategy to reduce the overhead of Lagrangian-based propagators (see [7, 10]).

Figure 6 shows the results of this evaluation, in particular the number of branches (in log scale) and the solution time (in seconds). The propagation obtained by individual Neuron

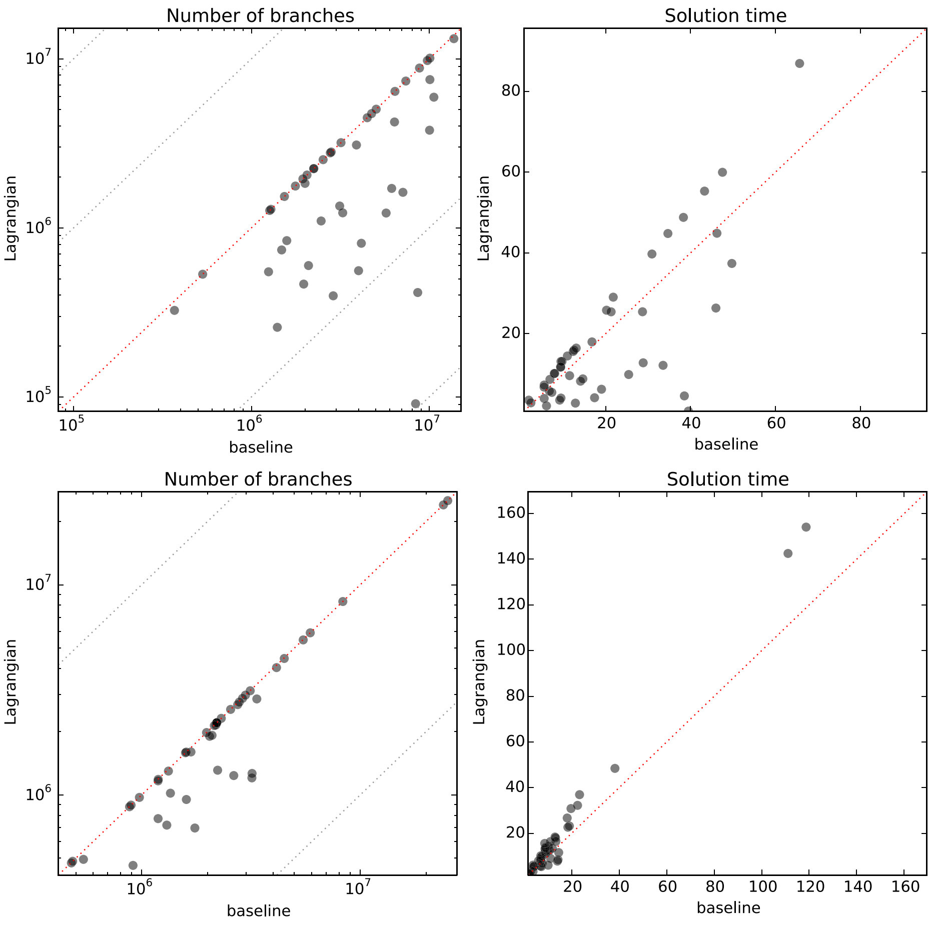

Fig. 6 Results for the reference configuration for *ANN1* (*top*) and *ANN2* (*bottom*)

Constraints is used as a baseline (x-axis). On the y-axis we report the results of our approach, where the baseline propagation is augmented via Lagrangian bounds on the network input and output.

On *ANN1*, the Lagrangian approach manages to considerably decrease the size of the search tree, in a significant number of instances. The improvement is typically enough to compensate for the increased propagation time. The approach is not analogously effective on *ANN2*, despite the number of branches is still reduced by a factor of 2-5 in some cases.

Subgradient iterations at the first activation We started our analysis by investigating the effect of increasing the number of subgradient iterations at the first constraint activation. In principle, improving the multipliers could provide considerable benefits with a small impact on the solution time. In practice, however, we found that increasing the number of initial subgradient iterations from 100 to 10,000 does not significantly affect the propagation strength and the solution time: we obtained only a minor improvement on *ANN2* for 1,000 iterations. These results (not reported in any figure) are consistent with the first stage of our experimentation, suggesting that 100 iterations are usually enough for the multipliers to converge approximately to their optimal values.

Subgradient iterations during search Next, we tried to assess the effect of performing multiplier updates during search. This was done by relying on multiplier caching and performing a small number of subgradient iterations at each activation of the constraint.

The results of this experimentation are reported in Fig. 7, using box plots. In particular, we show the values of the ratio between the number of branches and the solution time of our approach w.r.t. the baseline. The line in the middle of each box correponds to the median, the box boundaries are the first and third quartile, and the "whiskers" correpond to the minimum and maximum. In other words, 25 % of the values are contained in the interval between the lower whisker and box boundary, another 25 % of the values is between the lower box boundary and the median, and so on. Values lower than 1 correspond to cases where the new propagator beats the baseline. We show separate boxes for the new propagator and a restricted version of the approach, where the Lagrangian bounds are employed only for the z variable. This version corresponds in fact to the method we introduced in [27].

Introducing even a few subgradient iterations during search affects significantly the effectiveness of our Lagrangian approach: the number of branches is reduced by a factor from 3 to 100 in 50 % of the instances on the *ANN1* networks. The improvement is less dramatic, but still considerable, for the *ANN2* networks. The advantage is more pronounced for the version of the propagator that employs reduced-cost based filtering, with the performance gap being particularly large on the *ANN2* networks with 5 iterations. The lower number of branches does not translate into a reduction of the solution time, because of the cost of performing additional subgradient iterations at all search nodes.

Activation budget The activation budget approach introduced in Section 3.4 was designed as a simple technique to limit the propagator activation to the upper-most part of the search tree. The main underlying idea is that running a powerful propagator in that region has the highest pruning potential (since the underlying search subtrees are big) and the lowest overhead (since the number of upper nodes is comparatively small).

The approach is not strictly equivalent to a depth limit, because propagators may get activated multiple times per search node. As an advantage, the activation budget can be easily implemented in off-the-shelf available solvers, which to not provide access to the depth of search nodes.

In Fig. 8 we show the effect of introducing an activation budget, when the number of subgradient iteratons during search is fixed to 5 and 10. Introducing a 36 unit budget has a very small impact on the propagation strength and provides some benefits in terms of run time. Reducing the budget to 18 units has a more significant, negative, effect on the number of branches, but it provides very large improvements in terms of run time. The propagator configuration with 5 subgradient iterations and budget 18 beats the baseline over more than 75 % of the instances with *ANN1* and over ∼50% of the instances with *ANN2*. Using reduced-cost based filtering on the network inputs provides the best results on *ANN1*, while on *ANN2* the restricted version of the propagator behaves slightly better in terms of solution time.

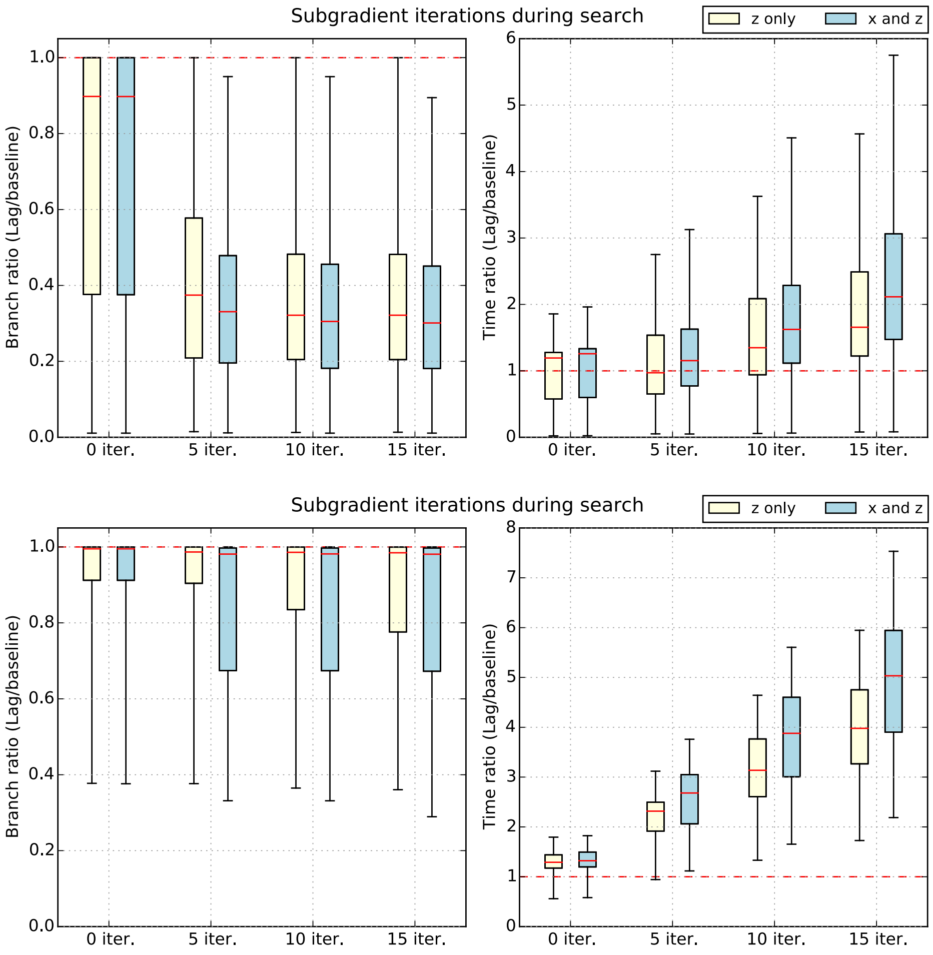

Fig. 7 Effect of increasing the number of subgradient iterations during search, for *ANN1* (*top*) and *ANN2* (*bottom*)

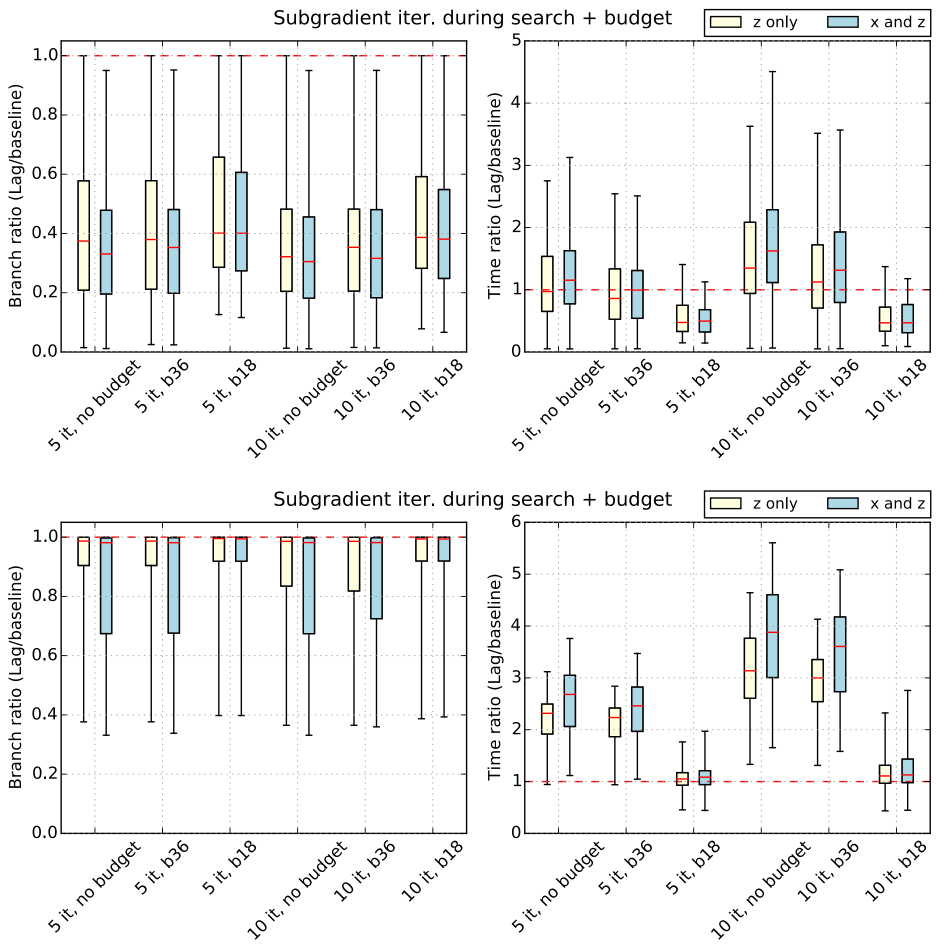

Fig. 8 Effect of introducing an activation budget, for *ANN1* (top) and *ANN2* (bottom)

4.4 Evaluation on a large-scale benchmark

With the goal to assess the performance of our new propagator in a more reliable fashion, we performed one last set of experiments on the second of our benchmarks (counting 200 instances). For the new propagator we used the best configuration identified in Section 4.3: an activation budget of 18 units, 1,000 initial subgradient iterations, multiplier caching with 5 subgradient iterations during search, and reduced-cost based filtering on the network input. As a baseline, we still have the individual Neuron Constraint propagation from [2].

The results of this evaluation are reported in Fig. 9. On the *ANN1* networks, the Lagrangian-based propagator enables a reduction of the number of branches by a factor of 2-10 in the large majority of instances. Thanks to the overhead reduction techniques, the solution time is also significantly improved. Moreover, when the smaller number of branches does not pay off for the additional computational effort, the gap in solution time between the Lagrangian approach and the baseline is still not large.

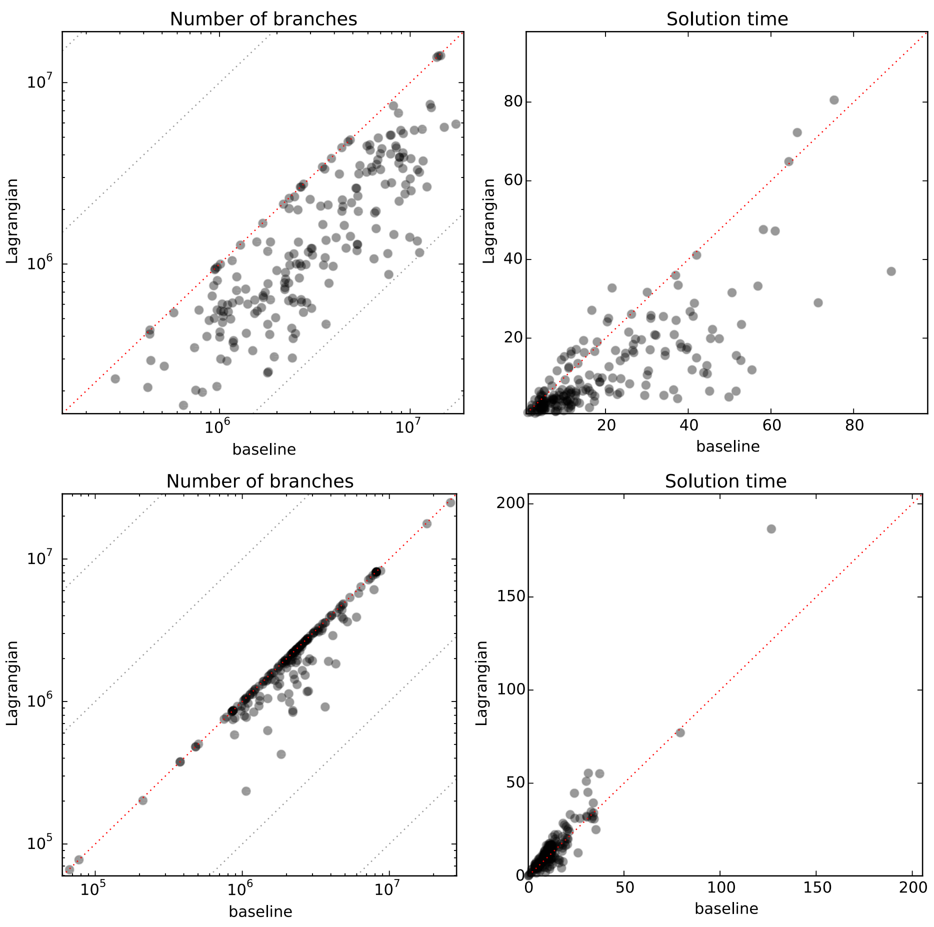

Fig. 9 Results on the large-scale benchmark, for *ANN1* (*top*) and *ANN2* (*bottom*)

The situation is different for *ANN2*, for which the Lagrangian approach is often not capable of pruning more than the baseline. However, improvements in terms of both the number of branches and the solution time still occur for a significant number of instances.

Considerations on the experimental results On the large-scale benchmark, the new Lagrangian propagator managed to provide a sharp improvement on many problem instances. The approach was able of reducing the search tree size by a factor of 2 to 10. Due to the increased propagation time, the improvements translated reliably into time gains only via the use of overhead reduction techniques: our simple activation budget proved quite effective in that respect. We experimented with other techniques (e.g. running the propagator once every two constraint activations, or randomizing the multipliers), but we obtained poor or non-significant results (not reported in this paper).

The simple filtering technique for the input variables from Section 3.3 proved effective in reducing the number of branches in a number of instances. This is a significant result, especially if we take into account that the ANN input in our problem consists of features obtained via sums (see Section 4.1), which notoriously have poor back-propagation.

The effectiveness of the Lagrangian approach, in particular the cost-based filtering, seems to be strongly dependent on the weights of the target network. The difference appears to be quite sharp: on *ANN1* the new propagator prunes much more than the baseline, and updating the multipliers during search is crucial. The performance of the new approach is more limited on *ANN2*, and updating the multiplier seems to be less effective. Based on some preliminary observations, we conjecture that the different performance is connected to specific networks in *ANN1* and *ANN2*, and in particular to (as yet unidentified) properties of the network weights.

5 Conclusions and future research directions

We have introduced a new propagator for two-layer, feed-forward Artificial Neural Networks, relying on a non-linear Lagrangian relaxation to compute bounds on the output and input variables. The quality of the computed bounds depends on the value of the Lagrangian multipliers, which are optimized using a subgradient algorithm. The approach can yield better bounds compared to the current state of the art (a method based on propagating each neuron individually): this is confirmed via an experimentation on a thermal-aware workload dispatching problem. As a drawback, the need to perform multiplier optimization significantly increases the propagation time. In our experiments, we managed to reduce the overhead to an acceptable level via three simple techniques: low priority propagation, multiplier caching, and an activation budget. Our final propagator setup outperformed the state of the art in a large number of cases, in terms of both search tree size and solution time.

There are several possible directions for future research: here we highlight three of them. First, our conjecture about the link between properties of the ANN weights and the performance of the Lagrangian-based propagation should be verified. If such properties could be identified (e.g. via Machine Learning, see [16]), it would be possible to decide at model construction time the propagation approach to use for ANN constraints. Second, several of the techniques we proposed rely on parameterized heuristics. The effect of some of the parameters has been assessed experimentally in this paper, but making the approach robust (and possibly adaptive) will require further research. In particular, tuning the value of the activation budget for specific problem instances, or specific networks, seems particularly important. Finally, from another perspective the relatively small time required by the Couenne solver to provide tight bounds suggests the possibility to design novel MINLP-CP hybrids.

References

1. Audet, C. (2014). A survey on direct search methods for blackbox optimization and their applications. In *Mathematics Without Boundaries* (pp. 31–56): Springer.
2. Bartolini, A., Lombardi, M., Milano, M., & Benini, L. (2011). Neuron Constraints to Model Complex Real-World Problems. In *Proc. of CP* (pp. 115–129).
3. Bartolini, A., Lombardi, M., Milano, M., & Benini, L. (2012). Optimization and Controlled Systems: A Case Study on Thermal Aware Workload Dispatching. Proc. of AAAI.
4. Basheer, I.A., & Hajmeer, M. (2000). Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods*, *43*(1), 3–31.
5. Belew, R.K., McInerney, J., & Schraudolph, N.N. (1991). Evolving networks: Using the genetic algorithm with connectionist learning. Proc. of Artificial Life, 511–547.
6. Belotti, P., Lee, J., Liberti, L., Margot, F., & Wächter, A. (2009). Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, *24*(4-5), 597–634.

7. Bergman, D., Ciré, A.A., & van Hoeve, W.-J. (2015). Lagrangian bounds from decision diagrams. *Constraints*, *20*(3), 346–361.
8. Bonfietti, A., & Lombardi, M. (2012). The weighted average constraint. In *Proc. of CP* (pp. 191–206): Springer.
9. Bonfietti, A., Lombardi, M., & Milano, M. (2015). Embedding decision trees and random forests in constraint programming. In *Proc. of CPAIOR* (pp. 74–90).
10. Cambazard, H., & Fages, J.-G. (2015). New filtering for atmostnvalue and its weighted variant: A lagrangian approach. *Constraints*, *20*(3), 362–380.
11. Chow, T.T., Zhang, G.Q., Lin, Z., & Song, C.L. (2002). Global optimization of absorption chiller system by genetic algorithm and neural network. *Energy and Buildings*, *34*(1), 103–109.
12. Conn, A.R., Scheinberg, K., & Vicente, L.N. (2009). Introduction To Derivative-free Optimization, volume 8. Siam.
13. d'Antonio, G., & Frangioni, A. (2009). Convergence analysis of deflected conditional approximate subgradient methods. *SIAM Journal on Optimization*, *20*(1), 357–386.
14. Focacci, F., Lodi, A., & Milano, M. (1999). *Cost-based domain filtering*.
15. Ge, S.S., Hang, C.C., Lee, T.H., & Zhang, T. (2010). Stable adaptive neural network control. Springer Publishing Company, Incorporated.
16. Gent, I.P., Kotthoff, L., Miguel, I., & Nightingale, P. (2010). Machine learning for constraint solver design – A case study for the alldifferent constraint. CoRR, abs/1008.4326.
17. Glover, F., Kelly, J.P., & Laguna, M. (1999). New Advances for Wedding optimization and simulation. In *Proc. of WSC* (pp. 255–260). IEEE.
18. Gopalakrishnan, K., & Asce, A.M. (2009). Neural Network Swarm Intelligence Hybrid Nonlinear Optimization Algorithm for Pavement Moduli Back-Calculation. *Journal of Transportation Engineering*, *136*(6), 528–536.
19. Gualandi, S., & Malucelli, F. (2012). Resource constrained shortest paths with a super additive objective function. In *Proc. of CP* (pp. 299–315): Springer.
20. Howard, J., Dighe, S., Vangal, S.R., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., Riepen, M., Gries, M., & et al. (2011). A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, *46*(1), 173–183.
21. Huang, W., Ghosh, S., & Velusamy, S. (2006). HotSpot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on VLSI*, *14*(5), 501–513.
22. Hutter, F., Hoos, H.H., Leyton-Brown, K., & Stùtzle, T. (2009). Paramils: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267–306.
23. Jayaseelan, R., & Mitra, T. (2009). A hybrid local-global approach for multi-core thermal management. In *Proc. of ICCAD* (pp. 314–320): ACM Press.
24. Kiranyaz, S., Ince, T., Yildirim, A., & Gabbouj, M. (2009). Evolutionary artificial neural networks by multi-dimensional particle swarm optimization. *Neural Networks*, *22*(10), 1448–1462.
25. Lemaréchal, C. (2001). Lagrangian relaxation. In *Computational Combinatorial Optimization* (pp. 112–156): Springer.
26. Ljung, L. (1999). System identification. Wiley Online Library.
27. Lombardi, M., & Gualandi, S. (2013). A new propagator for two-layer neural networks in empirical model learning. In *Proc. of CP* (pp. 448–463).
28. Montana, D.J., & Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proc. of IJCAI* (pp. 762–767).
29. Moore, J., Chase, J.S., & Ranganathan, P. (2006). Weatherman: Automated, Online and Predictive Thermal Mapping and Management for Data Centers. In *Proc. of ICAC* (pp. 155–164). IEEE.
30. Moré, J.J. (1978). The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis* (pp. 105–116): Springer.
31. Queipo, N.V., Haftka, R.T., Shyy, W., Goel, T., Vaidyanathan, R., & Tucker, P.K. (2005). Surrogate-based analysis and optimization. *Progress In Aerospace Sciences*, *41*(1), 1–28.
32. Sellmann, M. (2004). Theoretical foundations of cp-based lagrangian relaxation. In *Proc. of CP* (pp. 634–647): Springer.
33. Sellmann, M., & Fahle, T. (2003). Constraint programming based lagrangian relaxation for the automatic recording problem. *Annals of Operations Research*, *118*(1–4), 17–33.
34. Slusky, M.R., & van Hoeve, W.J. (2013). A lagrangian relaxation for golomb rulers. In *Proc. of CPAIOR* (pp. 251–267): Springer.
35. Van Cauwelaert, S., Lombardi, M., & Schaus, P. (2015). Understanding the potential of propagators. In *Proc. of CPAIOR* (pp. 427–436).
36. Zhang, G., Patuwo, B.E., & Hu, M.Y. (1998). Forecasting with artificial neural networks: The state of the art. *International Journal of Forecasting*, *14*(1), 35–62.