A New Propagator for Two-Layer Neural Networks in Empirical Model Learning

Michele Lombardi[1] and Stefano Gualandi[2]

[1] DISI, Università of Bologna, Dipartimento di Informatica: Scienza ed Ingegneria
michele.lombardi2@unibo.it
[2] University of Pavia, Dipartimento di Matematica
stefano.gualandi@unipv.it

Abstract. This paper proposes a new propagator for a set of Neuron Constraints representing a two-layer network. Neuron Constraints are employed in the context of the Empirical Model Learning technique, that enables optimal decision making over complex systems, beyond the reach of most conventional optimization techniques. The approach is based on embedding a Machine Learning-extracted model into a combinatorial model. Specifically, a Neural Network can be embedded in a Constraint Model by simply encoding each neuron as a Neuron Constraint, which is then propagated individually. The price for such simplicity is the lack of a global view of the network, which may lead to weak bounds. To overcome this issue, we propose a new network-level propagator based on a Lagrangian relaxation, that is solved with a subgradient algorithm. The approach is tested on a thermal-aware dispatching problem on multicore CPUs, and it leads to a massive reduction of the size of the search tree, which is only partially countered by an increased propagation time.

1 Introduction

Pushed by research advancements in the last decades, Combinatorial Optimization techniques have been successfully applied to a large number of industrial problems. Yet, many real-world domains are still out-of-reach for such approaches. To a large extent, this is due to difficulties in the formulation of an accurate declarative model for the system to be optimized.

The Empirical Model Learning technique (EML), introduced in [1], has been designed to enable optimal decisions making over complex systems considered beyond the reach of traditional combinatorial approaches. In EML, an approximate model of the target system is extracted via Machine Learning. Such *empirical model* captures the effect of the user decisions on one or more observables of interest (e.g. a cost measure or a constrained parameter). Then, the empirical model is encoded using a combinatorial technology and embedded into a combinatorial model to perform optimization.

Currently, the EML approach has been instantiated using Artificial Neural Networks (ANN) and Constraint Programming, respectively as Machine Learning and Combinatorial Optimization technologies. Specifically, in [1] an ANN is

C. Schulte (Ed.): CP 2013, LNCS 8124, pp. 448–463, 2013.

employed to learn the effect of task mapping decisions on the temperature of a quad-core CPU. In [2], the authors tackle a workload dispatching problem on a 48-core system with thermal controllers: in this case, 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. ANNs are employed to predict the mapping-dependent efficiency loss, i.e the combined effect of the thermal physics and the action of the on-line controllers.

The use of automatically extracted models for cost computation has been previously employed in the context of metaheuristic methods. EML stands out from those approaches for two main reasons: 1) because it makes the empirical model a *component*, easy to integrate with traditional constraints; 2) because it makes the empirical model *active*, rather than an simple function evaluator. In [1] an [2], this is achieved by encoding each neuron in the ANN as a *Neuron Constraint*, which is then propagated to narrow the search space.

Using individual constraints for the neurons is simple, but the loss of the network global view may degrade the propagation effectiveness. To address this issue, we propose a new propagator for the most common ANN structure in practice, i.e. a two-layer, feed forward network. We assume to have sigmoid neurons in the hidden layer, since they are a common choice [15], but the method easily extends to any differentiable activation function. The new propagator does not replace the use of multiple Neuron Constraints, but provides tighter bounds (hence stronger filtering) on the network output variables. The bounds are obtained via a Lagrangian relaxation, with the Lagrangian multipliers being optimized via a subgradient method. We test the approach on a simplified version of the thermal-aware dispatching problem from [2]: the new propagator leads to a substantial (sometimes massive) reduction of the search tree size, in particular for larger instances. This is however partially countered by an increased propagation time. Fortunately, on the basis of a rather strong conjecture that we give at the end of the paper, we believe a complexity reduction is possible.

The paper is structured as follows: Section 2 provides background information. Section 3 describes our Lagrangian relaxation and its solution method, while Section 4 explains how the Lagrangian multipliers are optimized. Section 5 provides our experimental results and Section 6 the concluding remarks.

2 Background and Related Works

Artificial Neural Neworks: An ANN is a system emulating the behavior of a biological network of neurons. Each ANN unit (artificial neuron) corresponds to the following function:

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

where x_i are the neuron inputs and w_i are their weights, b is called the bias and z is the neuron output. All the terms are $\in \mathbb{R}$. Besides, $f : \mathbb{R} \to \mathbb{R}$ is

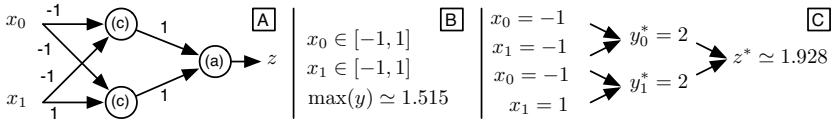

Fig. 1. A A two-layer, feed forward ANN. B Domains for the ANN inputs and actual output maximum. C Output bound computed by the existing propagators.

called *activation function* and is monotone non-decreasing. Some examples of activation functions follow:

$$(a)\ f(y) = y \quad (b)\ f(y) = \begin{cases} 1 \text{ if } y \geq 0 \\ -1 \text{ otherwise} \end{cases} \quad (c)\ f(y) = \frac{2}{1 + e^{-2y}} - 1 \tag{2}$$

Case (a), (b) and (c) respectively correspond to a linear, step and sigmoid neuron. Function (c) is called *tansig* and it is an accurate, faster to compute, approximation of $tanh(y)$. The neurons are connected in a network structure, the most common being an acyclic (i.e. feed-forward), two-layer graph. The first layer is referred as *hidden*, the second is called the *output layer* (because it provides network output). Typically, sigmoid neurons are employed in the hidden layer. Figure 1A shows an example of such a network. Each node represents a neuron, the weights are reported as label on the arcs, x_i are the network input and z is the network output. The weights of an ANN can be assigned automatically by minimizing the average square error on a known set of examples: there are many specifically designed, readily available, algorithms for this purpose [13,3,11].

Neuron Constraints: A Neuron Constraint is a constraint that encodes and propagates Equation (1) and is equivalent to the following pair of constraints:

$$(c_0) \quad z = f(y) \qquad\qquad (c_1) \quad y = b + \sum_{i=0}^{n-1} w_i x_i \tag{3}$$

where z, y and x_i are real-valued decision variables[1] with interval domain, i.e. $z \in [\underline{z}, \overline{z}]$, $y \in [\underline{y}, \overline{y}]$ and $x_i \in [\underline{x}_i, \overline{x}_i]$. The term y is called the neuron *activity*. It is possible to embed an ANN in a CP model by building a Neuron Constraint for each node in the network and by introducing decision variables to represent the output of each hidden neuron. Each Neuron Constraint is implemented either as a single entity or as an actual pair of constraints. In the second case, we must explicitly introduce a decision variable to model each neuron activity.

Motivating Example: The propagator for a Neuron Constraint enforces bound consistency on (c_0) and (c_1). For a single neuron, this leads to the tightest possible bounds on all the variables. However, this approach is much less effective once more complex networks are taken into account. Consider the two layer

[1] Note that 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 $\in \{0..100\}$).

network from Figure 1A, having *tansig* neurons in the hidden layer and a single linear neuron connected to the output.

Assuming both x_0 and x_1 range in $[-1, 1]$, the maximum possible value for the output z is $\simeq 1.515$ (see Figure 1B). Now, let y_j denote the activity of the j-th hidden neuron. The upper bound on z computed by the output neuron (i.e. z^*), is obtained by fixing both y_0 and y_1 to their maximum possible values (i.e. y_0^* and y_1^*). We have $y_0^* = 2$, obtained by fixing both x_0 and x_1 to -1. We have also $y_1^* = 2$, corresponding to $x_0 = -1$ and $x_1 = 1$. Therefore, z^* is $\simeq 1.928$. The loose bound is obtained since each neuron is propagated separately, thus allowing the network inputs to take incompatible values. This issue can be overcome by employing a global, network-level propagator, which is what this paper is about.

Related Works: Neural Networks have been used as cheap-to-compute cost function evaluators in the context of metaheuristics: in [4] a Genetic Algorithm exploits an ANN to estimate the performance of an absorption chiller. The work [14] proposes a custom heuristic for workload dispatching in a data center and uses an ANN for temperature estimation. In Control Theory, ANNs are employed on-line as predictors (i.e. dynamic system models) and their parameters are continuously adjusted according to the prediction error [6]. This is a specific case of *system identification* [12], which is the process of learning a (typically linear) system model to be used for on-line control, mostly at a local scale. A few works, such as [10], have employed ANNs for solution checking. Others have used ANNs as a surrogate system model for the back-computation of hidden parameters: in [8], this is done to estimate the condition of road pavement layers. Finally, in the OptQuest metaheuristic system [7], a neural network is trained during search with the aim to avoid trivially bad solutions.

As a common trait, in all the mentioned approaches the ANN is exploited in a rather limited fashion, namely as black-box function evaluator. Conversely, Empirical Model Learning has the ability to actively employ the extracted model to improve the performance of the optimization process.

3 Computing Bounds for the Network Output

In this work, we design a new propagator for computing bounds to the output of a two-layer, feed-forward network. Without loss of generality, we consider the problem of finding an upper bound for a single output variable, i.e. on solving:

$$\mathbf{P0}: \quad \max \; z = \hat{b} + \sum_{j=0}^{m-1} \hat{w}_j f(y_j) \tag{4}$$

$$\text{s.t.} \quad y_j = b_j + \sum_{i=0}^{n-1} w_{j,i} x_i \qquad \forall j = 0..m-1 \tag{5}$$

$$x_i \in [\underline{x}_i, \overline{x}_i] \qquad \forall i = 0..n-1 \tag{6}$$

where x_i are the network inputs (n in total), y_j are the activities of the hidden layer neurons (m in total). The term $w_{j,i}$ is the weight of the i-th input in activity of the j-th hidden neuron and b_j is the bias for the j-th hidden neuron. The term $\hat{w}_j$ is the weight of the output of the j-th hidden neuron in the activity of the output neuron and $\hat{b}$ is the bias for the output neuron. The z variable represents the activity of the output neuron: since all activation functions are monotone non-decreasing, an upper bound on z corresponds to an upper bound on the network output.

Problem Relaxation: Problem **P0** is non-linear, non-convex and cannot be solved in polynomial time in general. Therefore, we resort to a relaxation in order to obtain a scalable solution approach. Specifically, we employ a Lagrangian relaxation for Constraints (5), obtaining:

$$\textbf{LP0}(\lambda): \quad \max_{x,y} \; z(\lambda) = \hat{b} + \sum_{j=0}^{m-1} \hat{w}_j f(y_j) + \tag{7}$$

$$+ \sum_{j=0}^{m-1} \lambda_j \left(b_j + \sum_{i=0}^{n-1} w_{j,i} x_i - y_j \right) \tag{8}$$

$$x_i \in [\underline{x}_i, \overline{x}_i] \qquad \forall i = 0..n-1 \tag{9}$$

$$y_j \in [\underline{y}_j, \overline{y}_j] \qquad \forall j = 0..m-1 \tag{10}$$

where λ is the vector of Lagrangian multipliers λ_j, acting as parameters for the relaxation. The notations x and y refer to the vectors of the x_i and y_j variables. Constraints (10) have been added to prevent **LP0**(λ) from becoming unbounded. The values $\underline{y}_j$ and $\overline{y}_j$ are chosen so that Constraints (10) are redundant in the original problem. In particular:

$$\underline{y}_j = b_j + \sum_i \begin{cases} w_{j,i}\underline{x}_i \text{ if } w_{j,i} \geq 0 \\ w_{j,i}\overline{x}_i \text{ otherwise} \end{cases} \tag{11}$$

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 all points where Constraints (5) are satisfied, we have $z = z(\lambda)$, for every possible λ. Therefore, the set of solutions of **LP0**(λ) contains all the solutions of **P0**, with the same objective value. Hence the optimal solution $z^*(\lambda)$ of **LP0**(λ) is always a valid bound on the optimal solution z^* of **P0**.

Solving the Relaxation: Problem **LP0**(λ) can be decomposed into two independent subproblems **LP1**(λ) and **LP2**(λ) such that:

$$z^*(\lambda) = \hat{b} + \sum_{j=0}^{m-1} \lambda_j b_j + z^*_{LP1}(\lambda) + z^*_{LP2}(\lambda) \tag{12}$$

with:

$$z^*_{LP1}(\lambda) = \max_x \; z_{LP1}(\lambda) = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{m-1} \lambda_j w_{j,i} \right) x_i \qquad \textbf{LP1}(\lambda) \tag{13}$$

$$\text{s.t.} \quad x_i \in [\underline{x}_i, \overline{x}_i] \qquad \forall i = 0..n-1 \tag{14}$$

$$z^*_{LP2}(\lambda) = \max_y \; z_{LP2}(\lambda) = \sum_{j=0}^{m-1} (\hat{w}_j f(y_j) - \lambda_j y_j) \qquad \textbf{LP2}(\lambda) \tag{15}$$

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

The two subproblems can be addressed separately.

Solving LP1(λ): Problem **LP1(λ)** can be solved by assigning each x_i either to $\underline{x}_i$ or to $\overline{x}_i$, depending on the sign of the reduced weight $\tilde{w}_i(\lambda) = \sum_{j=0}^{m-1} \lambda_j w_{j,i}$. In detail:

$$z^*_{LP1}(\lambda) = \sum_{i=0}^{n-1} \begin{cases} \tilde{w}_i(\lambda)\, \overline{x}_i \text{ if } \tilde{w}_i(\lambda) \geq 0 \\ \tilde{w}_i(\lambda)\, \underline{x}_i \text{ otherwise} \end{cases} \tag{17}$$

The process requires nm steps to compute the reduced weights and n steps to obtain the final solution, for a worst case time complexity of $O(nm)$.

Solving LP2(λ): Problem **LP2(λ)** can be further decomposed into a sum of maximization problems of non-linear, non-convex, monovariate functions with box constraints. Each of the subproblems is in the form:

$$\max_{y_j} \; g_j(y_j, \lambda) = \; \hat{w}_j f(y_j) - \lambda_j y_j \tag{18}$$

$$\text{s.t. } y_j \in [\underline{y}_j, \overline{y}_j] \tag{19}$$

Each subproblem can be solved analytically, in case f is differentiable, which is a very realistic assumption given that in most practical applications f is a sigmoid. In such case, the objective function from Equation (18) will have a shape similar to the one depicted in Figure 2A. Hence the maximum can be found by comparing the value of $g_j(y_j, \lambda)$ on $\underline{y}_j$, on $\overline{y}_j$ (depending on the value of $\hat{w}_j$ and λ_j) or on the y_j value corresponding to a local maximum. The presence of at most one local maximum is guaranteed by the fact that both $f(y_j)$ and $\lambda_j y_j$ are mononote. Now, for the local minimum and maximum the derivative of $g_j(y_j, \lambda)$ will be null, i.e. $\hat{w}_j f'(y_j) - \lambda_j = 0$. Assuming a *tansig* activation function, this means that:

$$\hat{w_j} \frac{4e^{-2y_j}}{\left(1 + e^{-2y_j}\right)^2} - \lambda_j = 0 \tag{20}$$

By substituting $u = e^{-2y_j}$ in Equation (20), we get:

$$4\hat{w}_j u - \lambda_j \left(1 + 2u + u^2\right) = 0 \tag{21}$$

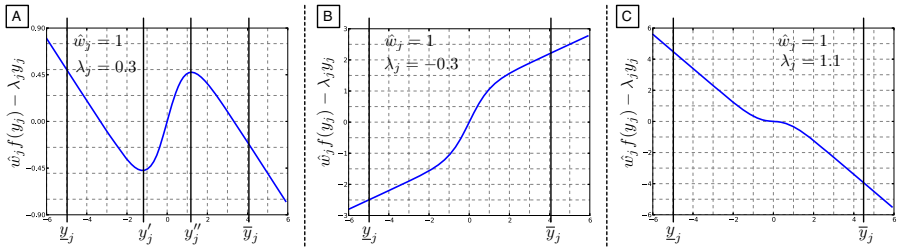

Fig. 2. Shapes for the cost function in the decomposition of **LP2(λ)**

Note that if $\lambda_j = 0$, then no local maximum exists. The same holds if $\hat{w}_j = 0$. Hence it is safe to assume $\lambda_j, \hat{w}_j \neq 0$ and we can get:

$$u^2 + \left(2 - 4\frac{\hat{w}_j}{\lambda_j}\right) u + 1 = 0 \tag{22}$$

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:

$$\left(2 - 4\frac{\hat{w}_j}{\lambda_j}\right)^2 - 4 \geq 0 \quad \Leftrightarrow \quad \frac{\hat{w}_j}{\lambda_j}\left(\frac{\hat{w}_j}{\lambda_j} - 1\right) \geq 0 \tag{23}$$

I.e. if $\hat{w}_j$ and λ_j are equal in sign and if $|\hat{w}_j| \geq \lambda_j$ (or equivalently if ${}^{\hat{w}_j}/_{\lambda_j} > 1$). In this case, the y_j values corresponding to the local minimum and maximum are given by:

$$y'_j = -\frac{1}{2}\log u', \quad y''_j = -\frac{1}{2}\log u'' \tag{24}$$

It can be shown that u' and u'' are guaranteed to be positive. If the conditions to have a real-valued solution for Equation (22) do not hold, then the maximum corresponds to either $\underline{y}_j$ or $\overline{y}_j$: in detail, if $sign(\hat{w}_j) \neq sign(\lambda_j)$, then this happens because both $f(y_j)$ and $-\lambda_j y_j$ are non-decreasing or non-increasing (see Figure 2B). If $|\hat{w}_j| < \lambda_j$ no local maximum exists (see Figure 2C), because the derivative of the *tansig* is always ≤ 1.

Hence, the solution of each subproblem in the decomposition of **LP2(λ)** can be found by solving Equation (20) (if the conditions are met) and by comparing the value of $g_j(y_j\lambda)$ for at most four y_j values. The process takes constant time. Solving **LP2(λ)** requires m such computations, plus nm iterations to obtain all $\underline{y}_j$ and $\overline{y}_j$. The overall worst case time complexity is $O(nm)$.

Summary: For a fixed value of all the Lagrangian multipliers λ, the relaxed subproblem **LP0(λ)** can therefore be solved via the process described in Algorithm 1, to yield an upper bound on z. In the algorithm, $z^*(\lambda)$ denotes the bound, while $x_i^*(\lambda)$ and $y_j^*(\lambda)$ are the values of x_i and y_j in the corresponding solution. We recall that a feasible solution for **LP0(λ)** may be infeasible for **P0**. The algorithm to compute a lower bound is analogous.

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

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

4 Optimizing the Lagrangian Multipliers

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

$$\mathbf{L0}: \quad \min_{\lambda} z^*(\lambda) \tag{25}$$

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 such that the corresponding solution of **LP0(λ)** does not change for very small variations of the multipliers, i.e. $x^*(\lambda') = x^*(\lambda'')$ and $y^*(\lambda') = y^*(\lambda'')$, with $\|\lambda' - \lambda''\| \to 0$. Then $z^*(\lambda)$ is differentiable in λ' and in particular:

$$\frac{\partial z^*(\lambda')}{\partial \lambda_j} = s_j = b_j + \sum_{i=0}^{n-1} w_{j,i} x_i^*(\lambda') - y_j^*(\lambda') \tag{26}$$

Equation (26) is obtained by differentiating the objective of **LP0(λ)** under the above mentioned assumptions. When such assumptions do not hold, the s_j values provide a valid *subgradient*. The optimum value of **L0** can therefore be found via a subgradient method, by starting from an assignment $\lambda^{(0)}$ and iteratively applying the update rule:

$$\lambda^{(k+1)} = \lambda^{(k)} - \sigma^{(k)} s^{(k)} \tag{27}$$

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

Step Update Policy: We have chosen to employ the corrected Polyak step size policy with non-vanishing threshold from [5]. This guarantees the convergence to the optimal multipliers (given infinitely many iterations), with bounded error. Other policies from the literature are more accurate, but have a slower convergence rate, which is in our case *the* critical parameter (since we will run the subgradient method within a propagator). In detail, we have:

$$\sigma^{(k)} = \beta \frac{z^*(\lambda^{(k)}) - (z^{best} - \delta^{(k)})}{\|s^{(k)}\|^2} \tag{28}$$

where β is a scalar value in $]0, 2[$. The term $z^{best} - \delta^{(k)}$ is an estimate of the **L0** optimum: it is computed as the difference between the best (lowest) bound found so far z^{best}, and a scalar $\delta^{(k)}$ dynamically adjusted during search. Hence, the step size is directly proportional to the distance of the current bound from the estimated optimal one, i.e. $z^*(\lambda^{(k)}) - (z^{best} - \delta^{(k)})$. The larger $\delta^{(k)}$, the larger the estimated gap w.r.t the best bound and the larger the step size.

The value of $\delta^{(k)}$ is non-vanishing, which means it is constrained to be larger than a threshold δ^*. This ensures to have $\sigma^{(k)} > 0$ and prevents the subgradient optimization from getting stuck. We determine the δ^* value when the propagator is first executed at the root of the search tree. Specifically, we choose $\delta^* = \gamma z^*(\lambda^{(0)})$, with γ being a small positive value. During search, we compute $\delta^{(k)}$ according to the following rules:

$$\delta^{(k+1)} = \begin{cases} \max(\delta^*, \nu\delta^{(k)}) & \text{if } z^*(\lambda^{(k)}) > z^{best} - \delta^{(k)} \\ \max(\delta^*, \mu z^*(\lambda^{(k)})) & \text{otherwise} \end{cases} \tag{29}$$

where $\nu, \mu \in]0, 1[$. In practice, 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, we "reset" $\delta^{(k)}$, i.e. we assume that the estimated optimum is $\mu\%$ lower than z^{best}.

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. In its most basic form (the one we adopt), a deflection technique consists in replacing the subgradient in Equation (27) and (28) with the following vector (see [5]):

$$d^{(k)} = \alpha s^{(k)} + (1 - \alpha) d^{(k-1)} \tag{30}$$

where $d^{(k)}$ is called search direction and α is a scalar in $]0, 1]$, meaning that $d^{(k)}$ is a convex combination of the last search direction and the current subgradient.

The components s_j having alternating sign in consecutive gradients (such that $s_j^{(k)} s_j^{(k-1)} < 0$) tend to cancel one each other in the deflected search direction.

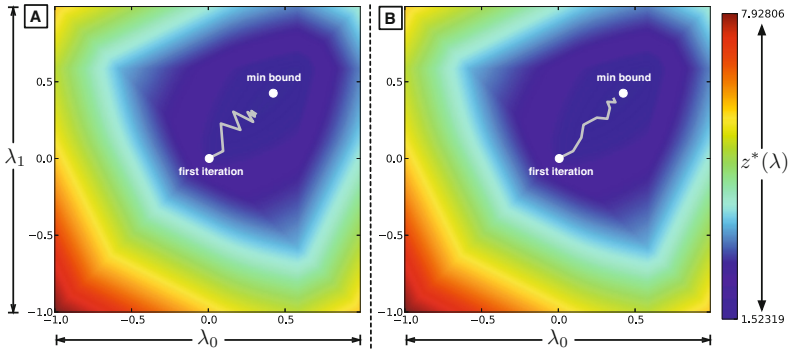

Fig. 3. A Subgradient optimization trace (10 iterations, no deflection). B Subgradient optimization trace (10 iterations, with deflection).

This behavior can be observed in Figure 3. This depicts the bound value as a function of λ for the network from Figure 1, together with the trace of the first 10 subgradient iterations. The use of the deflection allows to get considerably closer to the best possible bound ($\simeq 1.523$ in this case). Note that bound is not tight (the actual network maximum is $\simeq 1.515$), but it remarkably better than the value obtained from the propagation of individual neuron constraints ($\simeq 1.928$). When using the deflection technique, the value β from Equation (28) must be $\leq \alpha$ for the method to converge.

Propagator Configuration: We stop the subgradient optimization after a fixed number of steps. At the end of the process, we keep the best multipliers λ^* we have found and the corresponding bound $z^*(\lambda^*)$. We compute both an upper and a lower bound on the network output. The bound computation algorithm does not replace the propagation of individual neuron constraints, that we implement as pair of separated constraints as from Equation (3). We rely on individual neuron constraints to perform propagation on the network inputs, and for computing the bounds $\underline{y}_j$, $\overline{y}_j$ on the activity of the hidden neurons.

The new propagator is scheduled with the lowest possible priority in the target Constraint Solver. When the constraint is propagated for the first time, we perform 100 subgradient iterations, starting with all-zero multipliers ($\lambda_j^{(0)} = 0 \ \forall j = 0..m-1$). After that, when the constraint is triggered we perform only 3 iterations, starting from the best multipliers λ^* from the last activation. We keep the multipliers also when branching from a node of the search tree to one of its children, as a simple (but important) form of incremental computation.

We always use $\alpha = 0.5$ for the deflection and we keep $\beta = \alpha$. We re-initialize δ^* every time the constraint is triggered, using $\gamma = 0.01$. Therefore, the correction factor $\delta^{(k)}$ is always at least 1% of the computed bound computed at the first subgradient iteration. The attenuation factor ν for $\delta^{(k)}$, used when no improvement is obtained, is fixed to 0.75. The μ factor, used to reset $\delta^{(k)}$ when the estimated bound is improved, is 0.25 for the first constraint propagation and 0.05 for all the following ones. This choice is done on the basis that small

updates of the network inputs (such as those occurring during search) result in small modifications of the optimal multipliers.

5 Experimental Results

Target Problem: We have tested the new propagator on a simplified version of the thermal-aware workload dispatching problem from [2]. A number of tasks need to be executed on a multi-core CPU. 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 on the workload of the core, on that of the neighboring cores, on the thermal physics, and on the controller policy itself. An ANN is used to obtain an approximate model of the efficiency of each core, as a function of the workload and the room temperature. We target a synthetic quad core CPUs, simulated via an internally developed tool based on the popular Hotspot system [9]. A training set has been generated by mapping workloads at random on the platform and then obtaining the corresponding core efficiencies via the simulator. We have then trained a two-layer ANN for each core, with *tansig* neurons in the hidden layer and a single linear neuron in the output layer.

Each task i is characterized by a value cpi_i, measuring the degree of its CPU usage: lower cpi_i values correspond to more computation intensive (and heat generating) tasks. An equal number of tasks must be mapped on each core. The input of the ANN is the average cpi_i of each core and the room temperature t. The goal is to find a task-to-core mapping such that no efficiency is below a minimum threshold θ. We use the vector of integer variables p to model the task mapping, with $p_i = k$ iff task i is mapped to core k. Our model is as follows:

$$gcc\,(p, [0..n_c - 1], {}^{n_t}/_{n_c}) \tag{31}$$

$$acpi_k = \frac{n_c}{n_t} \sum_{i=0}^{n_t-1} cpi_i(p_i = k) \qquad \forall k = 0..n_c - 1 \tag{32}$$

$$e_k = \hat{b}_k + \sum_{j=0}^{n_h-1} \hat{w}_{k,j} y_{k,j} \qquad \forall k = 0..n_c - 1 \tag{33}$$

$$y_{k,j} = tansig\left(b_{k,j} + \sum_{h=0}^{n_c-1} w_{k,j,h} acpi_h + w_{k,j,n_c} t\right) \qquad \begin{array}{l}\forall k = 0..n_c - 1,\\ \forall j = 0..n_h - 1\end{array} \tag{34}$$

$$e_k \geq \theta \qquad \forall k = 0..n_c - 1 \tag{35}$$

$$p_i \in \{0..n_c - 1\} \qquad \forall i = 0..n_t \tag{36}$$

where n_t is the number of tasks and n_c is the number of cores (4 in our case). In (31) we use the *gcc* global constraint to have exactly ${}^{n_t}/_{n_c}$ tasks per core. For simplicity, we assume n_t is a multiple of n_c. Constraints (32) are used to obtain the average cpi_i per core (i.e. the $acpi_k$ variables). Constraints (33) and (34) define the ANN structure and are implemented using Neuron Constraints. The

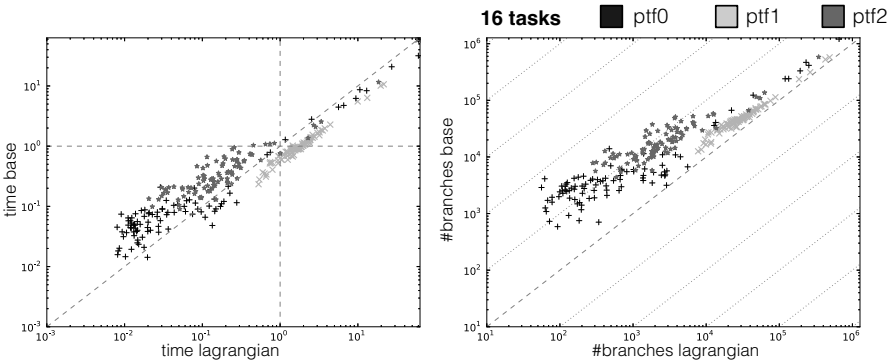

Fig. 4. Results for the 16 task workloads, on platforms 0-2

value n_h is the number of hidden neurons per ANN ($n_h = 5$ in our case), $\hat{b}_k$ is the bias of the output neuron in the ANN for core k, $\hat{w}_{k,j}$ are the neuron weights. Similarly, $b_{k,j}$ is bias of the hidden neuron j in the neural network for core k, while $w_{k,j,h}$, w_{k,j,n_c} are the weights. The value t is the room temperature, which is fixed for each problem instance. Each e_k variable represents the efficiency of the core k and is forced to be higher than θ by Constraints (35).

Experimental Setup: We tested two variants of the above model, where the new Lagragian propagator is respectively used (*lag*) and not used (*base*). A comparison with an alternative approach (e.g. a meta-hueristic using the ANN as a black-box), although very interesting, is outside the scope of this paper, which is focused on improving a filtering algorithm. We solve the problem via depth-first search by using a static search heuristic, namely by selecting for branching the first unbounded variable and always assigning the minimum value in the domain. The choice of a static heuristic allows a fair comparison of different propagators: pruning a value at a search node has the effect of skipping the corresponding sub-tree, but does not affect the branching decisions in an unpredictable fashion. As an adverse side effect, static heuristics are not well suited to solve this specific problem. Therefore, we limit ourselves to relatively small instances with either 16 or 20 tasks, which are nevertheless be sufficient to provide a sound evaluation. We consider 100 task sets for each size value. We performed experiments on 6 synthetic quad-core platforms, effectively testing $4 \times 6 = 24$ networks. For each combination of task set size and platform, we have empirically determined an efficiency threshold θ such that finding a feasible solution is non-trivial in most cases. Each experiment is run with a 60 seconds time limit. This is usually enough to find a solution, but it is never sufficient for proving infeasibility (which appears to take a very long time, mainly due to the chosen search heuristic). We have implemented everything on top of the Google or-tools solver. All the tests are run on a 2.8 GHz Intel Core i7.

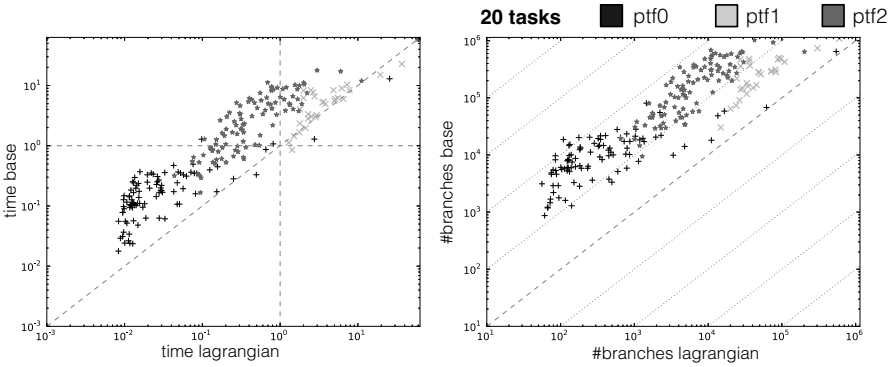

Fig. 5. Results for the 20 task workloads, on platforms 0-2

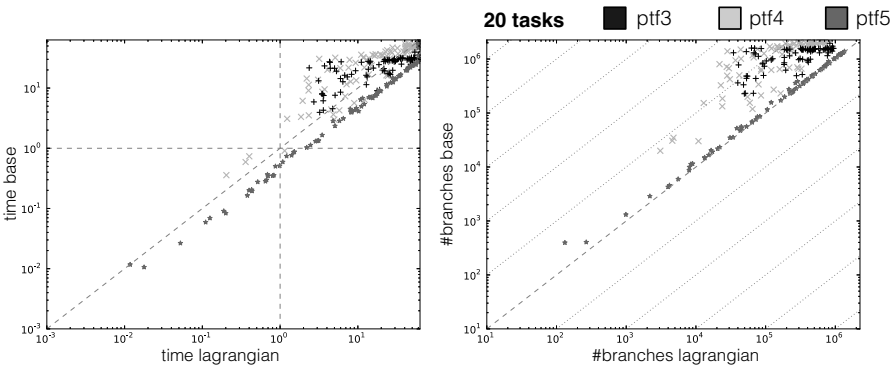

Fig. 6. Results for the 16 task workloads, on platforms 3-5

Results: The results of our experimentation are reported in Figures 4, 5, 6, 7. Each of them refers to 100 instances (with either 16 or 20 tasks) tested on three different platforms, and contains two scatter plots in log scale. The left-most diagram reports the solution times, with *lag* on the x axis and *base* on the y axis. Each instance is represented by a point and different colors and markers are used to distinguish between different platforms. Points above the diagonal represent instances where an improvement was obtained. A horizontal and a vertical line highlight the position of 1-second run times. The right-most plot is similar, except that it shows the number of branches and refers only to the instances for which a solution was found by both approaches. Each of the dotted diagonal lines represents a one-order-of-magnitude improvement.

The dramatic good news here is that the novel propagator achieves an impressive reduction in the number of branches, in a significant number cases. The gain may be as large as 2-3 orders or magnitude. This is an important result, pointing out that the bound improvement provided by the Lagrangian relaxation is far from negligible. Interestingly, the benefits tend to be higher for larger instances: a reasonable explanation for this behavior is that additional propagation is

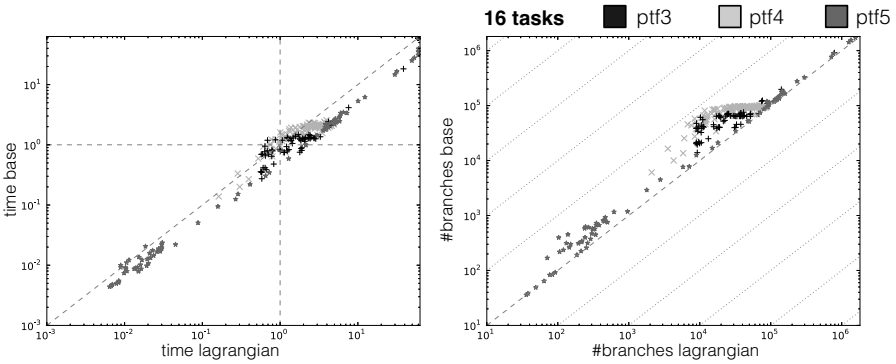

Fig. 7. Results for the 20 task workloads, on platforms 3-5

performed relatively high in the search tree, thus pruning larger subtrees as the instance size grows.

On the flip side, the new propagator comes with a considerable computational burden at each search node. As a general trend, on the 16 task instances this is sufficient to counter the benefits of the smaller number of branches: the *lag* approach therefore tends to be slower than the *base* one, although not much slower. For the 20 task workloads, there is a significant gain in solution time on platform 2 and 1, and a slight improvement on platform 0. The novel propagator behaves nicely on platform 3 and 4 as well, solving more instances than *base* respectively for the first 28 and 47 seconds. The method reports however a larger number of time-outs at the end of the 60 seconds. The *base* approach considerably outperforms the *lag* one on platform 5.

In general, the effectiveness of the Lagrangian propagator is non-uniform across different platforms: the reduction in the number of branches is much larger for platforms 0, 2, 3 and 4 than it is for platforms 1 and 5. This rises interest in investigating techniques to identify the network weight configurations that are more likely to benefit from the new propagator. The results seem to be much more consistent for different workloads on a single platform, although this may be due in part to the way our task sets are generated.

Finally, it is worth noting that the higher scalability (on the time side) of the Lagrangian approach is in part due to the use of subgradient optimization. We recall from Section 4 that for each constraint activation (except for the first one) we perform only 3 subgradient iterations. Since such number is fixed regardless of the number of tasks, the computational cost of the new propagator grows proportionally slower as the instances become larger.

6 Concluding Remarks

Summary: We have introduced a novel propagator for two-layer, feed forward ANNs, to be used in Empirical Model Learning. The new propagator is based on a Lagrangian relaxation, which is solved for a fixed assignment of the multipliers

via a fast, dedicated, approach. The multipliers themselves are optimized via a subgradient method. The current implementation works for *tansig* sigmoids in the hidden layer, but targeting other activation functions should be easy enough, provided they are differentiable.

The novel propagation does not replace the existing ones, but allows the computation of tighter bound on the ANN output variables. The approach manages to obtain a substantial reduction of the number of branches (up to 2-3 orders of magnitude) in our test set. The method seems to work best for comparatively larger instances. On the other side, the new propagation is computationally expensive, countering in part the benefits of the smaller search tree. Nevertheless, a gain in terms of solution time is obtained in a significant number of cases.

Future Work: A natural direction for future research is devising a way to filter the x_i variables, based on the Lagrangian relaxation. Second, the highest priority for future developments is achieving a reduction in the computation time, in order to fully exploit the reduction in the number of branches. This goal can be pursued (1) via the application of additional incremental techniques or (2) by improvements in the multiplier optimization routine. The computation of $\underline{y}_j$, $\overline{y}_j$ can be easily be made incremental, since they are linear expression. The incremental update of the **LP0(λ)** solution upon changes in λ is trickier, since all the multipliers tend to change after every subgradient iterations. We believe however that the convergence of the multiplier optimization routine offers large room for improvements, on the basis of the following conjecture.

The Conjecture: Let us assume that the relaxed problem $z^*(\lambda)$ from Section 4 is differentiable for the optimal multipliers λ^*. As a consequence, it must hold $\frac{\partial z^*(\lambda^*)}{\partial \lambda_j} = 0$ for every λ_j. Now, the partial derivatives are given by Expressions (26), which also represents the violation degree of Constraints (5). Therefore, if $z^*(\lambda)$ is differentiable in λ^*, then the relaxation solution $x^*(\lambda^*)$, $y^*(\lambda^*)$ is feasible for the original problem and the bound is tight. This means that the original problem can be solved via convex optimization.

Since we know problem **P0** is non-convex and hard to solve in general, we expect the above situation to be symptomatic of tractable subclasses, which can be probably identified by an analysis of the network weights. For example we know that, if the products $w_{j,i}\,\hat{w}_j$ have constant sign $\forall j$, then propagating the individual Neuron Constraints is sufficient to compute tight bounds on z.

Therefore, we expect that non-trivial Lagrangian bounds correspond to non-differentiable points of $z^*(\lambda)$. Such non-differentiable areas are given in our case by a set of hyperplanes in $\mathbb{R}^m$ (i.e. on the space of the multipliers), with the coefficients of the hyperplanes being easy to compute. This information can be exploited to focus the search for the optimal λ to a much smaller space, improving the rate of convergence and decreasing the overall computation time.

References

1. Bartolini, A., Lombardi, M., Milano, M., Benini, L.: Neuron Constraints to Model Complex Real-World Problems. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 115–129. Springer, Heidelberg (2011)
2. Bartolini, A., Lombardi, M., Milano, M., Benini, L.: Optimization and Controlled Systems: A Case Study on Thermal Aware Workload Dispatching. Proc. of AAAI (2012)
3. Belew, R.K., McInerney, J., Schraudolph, N.N.: Evolving networks: Using the genetic algorithm with connectionist learning. In: Proc. of Second Conference on Artificial Life, pp. 511–547 (1991)
4. Chow, T.T., Zhang, G.Q., Lin, Z., Song, C.L.: Global optimization of absorption chiller system by genetic algorithm and neural network. Energy and Buildings 34(1), 103–109 (2002)
5. Frangioni, A., D'Antonio, G.: Deflected Conditional Approximate Subgradient Methods (Tech. Rep. TR-07-20). Technical report, University of Pisa (2007)
6. Ge, S.S., Hang, C.C., Lee, T.H., Zhang, T.: Stable adaptive neural network control. Springer Publishing Company, Incorporated (2010)
7. Glover, F., Kelly, J.P., Laguna, M.: New Advances for Wedding optimization and simulation. In: Proc. of WSC, pp. 255–260 (1999)
8. Gopalakrishnan, K., Ph, D., Asce, A.M.: Neural Network Swarm Intelligence Hybrid Nonlinear Optimization Algorithm for Pavement Moduli Back-Calculation. Journal of Transportation Engineering 136(6), 528–536 (2009)
9. Huang, W., Ghosh, S., Velusamy, S.: HotSpot: A compact thermal modeling methodology for early-stage VLSI design. IEEE Trans. on VLSI 14(5), 501–513 (2006)
10. Jayaseelan, R., Mitra, T.: A hybrid local-global approach for multi-core thermal management. In: Proc. of ICCAD, pp. 314–320. ACM Press, New York (2009)
11. Kiranyaz, S., Ince, T., Yildirim, A., Gabbouj, M.: Evolutionary artificial neural networks by multi-dimensional particle swarm optimization. Neural Networks 22(10), 1448–1462 (2009)
12. Ljung, L.: System identification. Wiley Online Library (1999)
13. Montana, D.J., Davis, L.: Training feedforward neural networks using genetic algorithms. In: Proc. of IJCAI, pp. 762–767 (1989)
14. Moore, J., Chase, J.S., Ranganathan, P.: Weatherman: Automated, Online and Predictive Thermal Mapping and Management for Data Centers. In: Proc. of IEEE ICAC, pp. 155–164. IEEE (2006)
15. Zhang, G., Patuwo, B.E., Hu, M.Y.: Forecasting with artificial neural networks: The state of the art. International Journal of Forecasting 14(1), 35–62 (1998)