Applied Intelligence (2023) 53:24963–24990
https://doi.org/10.1007/s10489-023-04745-8

Hierarchical multi-scale parametric optimization of deep neural networks

Sushen Zhang[1] · Vassilios S. Vassiliadis[2] · Bogdan Dorneanu[3] · Harvey Arellano-Garcia[3]

Accepted: 28 May 2023 / Published online: 31 July 2023

Abstract
Traditionally, sensitivity analysis has been utilized to determine the importance of input variables to a deep neural network (DNN). However, the quantification of sensitivity for each neuron in a network presents a significant challenge. In this article, a selective method for calculating neuron sensitivity in layers of neurons concerning network output is proposed. This approach incorporates scaling factors that facilitate the evaluation and comparison of neuron importance. Additionally, a hierarchical multi-scale optimization framework is proposed, where layers with high-importance neurons are selectively optimized. Unlike the traditional backpropagation method that optimizes the whole network at once, this alternative approach focuses on optimizing the more important layers. This paper provides fundamental theoretical analysis and motivating case study results for the proposed neural network treatment. The framework is shown to be effective in network optimization when applied to simulated and UCI Machine Learning Repository datasets. This alternative training generates local minima close to or even better than those obtained with the backpropagation method, utilizing the same starting points for comparative purposes within a multi-start optimization procedure. Moreover, the proposed approach is observed to be more efficient for large-scale DNNs. These results validate the proposed algorithmic framework as a rigorous and robust new optimization methodology for training (fitting) neural networks to input/output data series of any given system.

Keywords Deep neural networks · Hierarchical multi-scale search · Scaling factor · Sensitivity analysis · Finite difference · Automatic differentiation

1 Introduction

Due to their superior capabilities of extracting capabilities of extracting information from big sets of data in an automatic way [1, 2], deep neural networks (DNNs) are currently applied in many domains, ranging from material synthesis and manufacturing [3, 4], biomedical applications [5], health and safety [6], to waste valorization [7], power supply [8, 9] or process industry and Internet of Things [10–12]. The framework for their training has remained fixed and stable since their introduction and it mainly involves the optimization of all the neurons through the process of backpropagation [2, 13], often coupled with well-known algorithms based on gradient descent [14–17] or Levenberg–Marquardt approaches [18–21]. Seldom posed is the question of whether all layers of neurons should be optimized together in one run.

A major constraint in the deployment of DNNs in real-time applications relates to the limited computation power, storage capacity, energy and time available in practice in real-world industrial systems for decision-making [22]. Key approaches for the deployment of DNN-based solutions under these low-resource conditions are often split into two categories [23]: a) the design of hardware architecture able to handle efficient data flow mapping strategies and memory hierarchy, and b) the design of software approaches for trade-off optimization of the DNN models, with the aim to reduce the number of model parameters and operations.

These techniques focused on the software development are known as compression and acceleration methods in literature, with the main approaches focusing on pruning,

✉ Bogdan Dorneanu
dorneanu@b-tu.de

[1] Department of Chemical Engineering and Biotechnology, University of Cambridge, Cambridge, UK

[2] Cambridge Simulation Solutions LTD., Waterbeach, Cambridge, UK

[3] LS Prozess- Und Anlagentechnik, Brandenburg University of Technology Cottbus-Senftenberg, Cottbus, Germany

quantization, low-rank factorization and knowledge distillation. A detailed overview of these approaches, as well as the main challenges in their development and application, can be found in recent survey papers [22, 24–28].

Of these, pruning is a demonstrated powerful technique that removes redundant parameters and connections, while retaining highly important features of the DNN, which can be applied either during or after training. Among the various options, pruning can be used for removing unimportant weight connections (weight pruning), individual redundant neurons (neuron pruning), least important filters (filter pruning) and redundant layers of neurons (layer pruning). An important challenge in the application of pruning in the development of efficient sparse model architectures for DNNs relates to the fact that the pruning percentage is manual, and more sophisticated approaches are required where the pruning is decided automatically by tuning some hyperparameters [22, 28]. Furthermore, more structured pruning techniques are required for deciding the importance of the layers that can be removed [22].

Assuming the network is optimized partially in each iteration, there is the need to determine which layer to optimize first. A criterion suitable in this situation is the adoption of neural sensitivity analysis.

Neural sensitivity analysis has been widely adopted in the analysis of DNNs with the aim to demystify the'black-box' nature and add further metrics to identify how the network reacts to changes in the explanatory variables [29, 30]. Early stages of research put emphasis exclusively on how the output or the objective change with a perturbation in the input [13, 31, 32]. While this is an important step in understanding how a network responds to different data sets, the analysis is limited by only looking at the sensitivity of the explanatory variables. Moreover, research on neural sensitivity analysis often concentrates on the relative values of the weights or inputs manipulated to become a sensitivity measure [13, 31, 33, 34].

The sensitivity is an important measure when performing analysis on a neural network because it is indicative of the component importance for a neuron, a layer, or a block of layers [22, 30]. In particular, the sensitivity sets out the change in the value of the output or the objective function caused by a perturbation in the value of the component [35, 36].

This paper presents the development of a novel hierarchical multi-scale framework for the training of DNNs that incorporates neural sensitivity analysis for the automatic and selective training of neurons evaluated to be the most effective, based on the work in [37]. This contribution improves the description of the proposed algorithm, as well as utilizes more examples for illustrating the application and benefits of the training approach. Overall, the novel elements of this contribution include:

- The use of neural sensitivity analysis to evaluate the importance of the neuron. For this purpose, two methods for evaluating the sensitivity are used: the automatic differentiation and the finite difference method. The former is more rigorously developed, but more computationally expensive. The latter is easy to implement and highly flexible, but requires more parameter settings to obtain sensible results.
- The use of both first- (first derivatives) and second-order (second derivatives) information for the computation of the sensitivity measures is investigated.
- The adoption of a new definition of the sensitivity measure through a scaling factor that enables the evaluation of the sensitivity of the individual components of the network.

The present work focuses on the theoretical presentation and the introduction of the mathematical development of the proposed approach. In the following sections the neural sensitivity analysis approach is introduced. The analysis extends to all neurons in the network and can achieve its selective tuning. Subsequently, a novel framework for the training of DNNs, of moderate complexity, is proposed, where the sensitivity values guide a hierarchical multi-scale search down a binary tree representing the importance of the layer. This efficient algorithm to search for the most sensitive layer to tune during training is thereafter applied on several case studies, to illustrate the implementation and the characteristics of the proposed approach. The remainder of the paper is organized as follows: Section 2 provides an overview of the use of neural sensitivity analysis for the identification of the importance of the neurons, while the sensitivity analysis procedure based on scaling factors is introduced in Section 3. The implementations of the proposed hierarchical multi-scale approach based on the first- and second-order information are presented in Sections 4 and 5, respectively, and the two algorithms are compared in Section 6. The application of the approach to various case studies is illustrated in Section 7, while Section 8 discusses the main conclusions of this study, as well as provides some directions for future improvement and use of the proposed approach.

2 Background

2.1 Developments in neural sensitivity analysis

Early stages of research in neural sensitivity analysis focus either on the values of the weights of the network [38–40], or on the sensitivity of the input values [13, 31, 32]. The aim is to observe the influence of the input on the output or the objective function, *i.e.*, the explanatory capacity of the network [31, 35, 36].

Table 1 Sensitivity measures adopted in literature

Sensitivity measure	Equation	Reference
Numerical sensitivity measure	$\frac{\partial y_k}{\partial x_i} = f'\left(net_k\right) \sum_{j=1}^{L} v_{jk} f'\left(net_k\right) w_{ij}$	[32, 41]
Weight product	$WP_{ik} = \frac{x_i}{y_k} \sum_{j=1}^{L} w_{ij} v_{jk}$	[38, 42, 43]
Q factor	$Q_{ik} = \frac{\Sigma_{j=1}^{L}\left(\frac{w_{ij}}{\Sigma_{r=1}^{N} w_{rj}} v_{jk}\right)}{\Sigma_{i=1}^{N}\left(\Sigma_{j=1}^{L}\left(\frac{w_{ij}}{\Sigma_{r=1}^{N} w_{rj}} v_{jk}\right)\right)}$	[39, 44, 45]

* Notations are described in Sect. 9

There are several key measures that have been used in the context of neural sensitivity analysis, as demonstrated in Table 1. While a variety of measures have been proposed, the sensitivities are often based exclusively on the values of the weights. A more practical method is to provide perturbations in the inputs and observe the effects on the outputs [29, 30, 46]. A sensitivity analysis on the weights has been demonstrated to be ineffective in measuring the functionality of the network [32]. Sensitivity analysis on the input values is more widely adopted in image recognition [47–49] and engineering [32, 36, 50] research, but is limited in its application for cases with discrete inputs [32, 51].

More recent work in image processing demystifies the convolutional neural network (CNN) by perturbing a pixel or a small region in an image and observing its effect on the objective [52]. Other studies seek to find partial derivatives of the image classification results with regard to individual pixels and visualize it in a graph, as a measure of input sensitivity [53, 54].

The advantages of the perturbation method relate to the fact that it is simple to implement and communicates a clear message on how each variable interacts to give the objective value. In the case of the partial derivatives method, the advantage is that it is a more rigorously developed computational algorithm that can be easily interpreted.

However, while these methods all merit in their own design, the sensitivity analysis is exclusively focused on the input importance. Furthermore, an important challenge in the case of the perturbation methods is the combinatorial explosion that would occur when assessing the impact on the output of all the elements of the input and all their possible combinations [46]. As such, approaches that focus on determining the importance of individual neurons can be used to tackling this complexity issue.

2.2 Identification of layer/neuron importance

The objective of producing faster and more efficient network models can be achieved by developing new approaches for revealing hidden information such as importance of individual or layer of neurons [55]. Several methods have been proposed to identify the importance of neurons through Neural Interpretation Diagrams (NIDs) [56–58], which represent the relative magnitude of each connection weight by line thickness. The positive weights are viewed as"excitator signals" while the negative weights are viewed as"inhibitor signals". The diagram assumes that by tracking the path with thicker lines (higher positive weight values), it is possible to find input variables and neurons that are more important. However, such diagrams will be difficult to visualize when the amount of connections is large, *i.e.*, with a large number of neurons.

Other studies focus on visualizing the importance of neurons through a relevance score [59–61], calculated from the Layer-wise Relevance Propagation [62, 63]. This method is developed because images used as inputs contain a large number of pixels in each entry, thus making it impossible to disturb single pixels for sensitivity.

While these methods focus exclusively on visualization of the neural importance, they are either too simplistic (by constructing graphs based on raw weight values) or highly complicated (by defining an equation of the relevance score). The framework proposed in this work has a moderate complexity that allows for the neural importance to be evaluated for the purpose of selective tuning.

3 Sensitivity analysis based on scaling factors

With a variety of definitions of sensitivity values, most of the state-of-the-art focuses on the sensitivity of the input features. In this section a scaling factor is formulated into the structure of DNNs and two methods to perform the sensitivity analysis are proposed.

The scaling factor can be viewed as a controller of the significance of a network component (neurons, layers, or blocks of layers). The sensitivity of that component is defined to be equal to the partial derivative of the objective against the scaling factor associated with that component. Two approaches are considered for the calculation of the partial derivatives: the finite difference and the automatic

differentiation methods, which will be briefly discussed in the following sections.

3.1 Scaling factors

Before demonstrating the mathematical formulation of the sensitivity analysis, the scaling factors are introduced. The idea of a scaling factor for the training of DNNs is inspired from [67], where it is adopted for the sensitivity analysis of the predictive modification of biochemical pathways to optimize the selection of reaction steps. The scaling factor effectively allows the sensitivity ranking of each reaction step and the resulting analysis greatly simplifies the selection process. The advantage of this procedure is that it is fast to determine a minimal set of reaction steps. Under this framework, the scaling factor acts to determine the most sensitive pathways with regard to the overall performance of the biochemical process.

A similar analysis set is adopted for an Artificial Neural Network (ANN). The terminology ANN is used here instead of DNN because the analysis is applicable to all forms of ANNs, both shallow and deep. Further, the mathematical details of the approach are detailed.

Firstly, the ANN is defined as the following process:

$$z_{l,i,k} = f(y_{l,i,k}) \tag{1}$$

where z is the output from a neuron, y is the input to the neuron, $l = 1, \ldots, NL$ is the layer index, $i = 1, \ldots, NN_l$ is the neuron index within layer l, and $k = 1, \ldots, NK$ is the data point index.

In the next step, a scaling factor is introduced into the formulation of the ANN such that it pre-multiplies the output value of a particular neuron. The factor can effectively serve to represent the sensitivity of the neuron in the optimization process.

$$z_{l,i,k} = \theta_{l,i} \bullet f\left(y_{l,i,k}\right) \tag{2}$$

where

$$\theta_{l,i} = \begin{cases} 1, & \text{neuron exists} \\ 0, & \text{neuron does not exist} \end{cases}$$

The artificial parameters $\theta_{l,i} \in [0, 1]$ are the scaling factors.

Subsequently, the sensitivity of the neuron of the neuron layer is calculated from the partial derivatives of the objective function with regard to the scaling factor introduced, to determine its existence. The mean square of errors (MSE) will be used as an objective function for the purpose of this study. In the following, the two procedures (based on numerical and automatic differentiation, respectively) will be used for determining the sensitivity value.

3.2 Sensitivity analysis through automatic differentiation

The automatic differentiation is an intuitive, rigorous method that calculates the value of the partial derivative of the objective function with respect to individual network components at machine precision [64, [65]. Although often difficult to derive, the automatic differentiation is efficient in its implementation. It can be applied to regular code with minimal change, and it allows branching, loops, and recursion [64]. Capabilities for automatic differentiation are well-implemented in most deep learning frameworks, and avoid the use of tedious derivations or numerical discretization during the computation of derivatives of all orders in space–time [66].

The sensitivity analysis procedure implies the solution of an optimization problem involving a set of parameters $\theta \in \mathrm{R}^{N_\theta}$, $N_\theta \geq 1$. The optimization problem is defined as follows:

$$\phi = \min_{x \in \mathcal{X}} f(x;\, \theta)$$

$$s.t. \quad h(x;\, \theta) = 0$$

Other constraints, equalities and/or inequalities, that nonetheless do not depend on θ, will be ignored.

For the solution of the optimization problem defined above, the Lagrangian function is considered as:

$$\mathcal{L}(x, \lambda; \theta) = f(x; \theta) + \lambda^T h(x; \theta) \tag{3}$$

where x is the set of tuning parameters (primal variables) of the optimization problem, λ is the set of Lagrange multipliers associated with the constraints $h(\cdot;\cdot)$, and θ is the set of scaling parameters introduced in the formulation to facilitate the calculation of sensitivities associated with the presence of individual or subsets of nodes, for example entire layers of an ANN, and their impact on the quality of the training fitting function.

At the optimal solution (x^*, λ^*), the Lagrangian function becomes:

$$\mathcal{L}(x^*, \lambda^*; \theta) = f(x^*; \theta) + 0 \tag{4}$$

The total derivative/gradient of the objective function with respect to θ at the optimal point $(x^*(\theta), \lambda^*(\theta))$, for a given value of the vector of parameters θ is calculated as follows:

$$\begin{aligned} \left.\frac{Df}{D\theta}\right|_{[x^*(\theta),\lambda^*(\theta),\theta]} &= \left.\frac{D\mathcal{L}}{D\theta}\right|_{[x^*(\theta),\lambda^*(\theta),\theta]} \\ &= \left.\left[\frac{\partial f}{\partial x}\frac{\partial x}{\partial \theta} + \lambda^T\left(\frac{\partial h}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial h}{\partial \theta}\right)\right]\right|_{[x^*(\theta),\lambda^*(\theta),\theta]} \\ &= \left.\left[\frac{\partial f}{\partial x}\frac{\partial x}{\partial \theta}\right]\right|_{[x^*(\theta),\lambda^*(\theta),\theta]} \end{aligned}$$

Subsequently, the form of the ANN neuron fitting constraints adopting the scaling factor in [67] is considered as:

$$h_{l,i,k} = z_{l,i,k} - \theta_{l,i} \bullet f\left(y_{l,i,k}\right) \tag{5}$$

where $z_{l,i,k}$ is the set of neurons outputs for each data point in our dataset, y is the set of variables in other equality constraints of the general form $g(y, z) = 0$.

The variables z are the ones being modified with the artificially introduced scaling parameters θ.

Other equations that are considered relate specifically to the influence of the node (l, i) at data point (k):

$$y_{l,i,k} = \sum_{j=1}^{NN_{l-1}} z_{l,j,k} \bullet W_{l,j,i} \tag{6}$$

where $l = 2, ..., NL, i, j = 1, ..., NN_l$, and $k = 1, ..., NK$.

An example formulation of the neural sensitivity analysis through automatic differentiation is provided in the Supplementary Material. In addition, other practical aspects including the implementation into simple matrix multiplication are described. Moreover, the CPU time and memory analysis of running neural sensitivity analysis using automatic differentiation are provided.

3.3 Sensitivity analysis through finite differences

An alternative to gauge the sensitivity of neurons is the finite difference method. This approach is less rigorous and produces less accurate results when applied to DNNs [68]. However, it is easy in terms of implementation into more sophisticated forms. Moreover, it can serve as a reference to the results obtained from automatic differentiation.

In the following, the mathematical formulation for the procedure adopting the finite difference method is described. In order to generate more accurate results, the central differences approach is used in this case. Thus, the finite differences method is defined to be:

$$f'(x) = \lim_{\epsilon \to 0} \frac{f\left(x + \frac{1}{2}\epsilon\right) - f\left(x - \frac{1}{2}\epsilon\right)}{\epsilon} \tag{7}$$

For the illustration of the procedure, an example formulation will be utilized and then extended to a generalized cases. Assuming a network with the architecture [2, 2, 2], the sensitivity of each neuron is calculated by first solving the optimization problem described in the previous section with the following constraints:

$$z_{1,1,k} - f\left(y_{1,1,k}\right) = 0 \tag{8}$$

$$y_{1,1,k} - W_{1,1,0} + W_{1,1,1} \bullet x_{1,k} + W_{1,2,1} \bullet x_{2,k} \tag{9}$$

$$z_{1,2,k} - f\left(y_{1,2,k}\right) = 0 \tag{10}$$

$$y_{1,2,k} - W_{1,2,0} + W_{1,2,1} \bullet x_{1,k} + W_{1,2,2} \bullet x_{2,k} \tag{11}$$

$$v_{1,k} - W_{2,1,0} + W_{2,1,1} \bullet z_{2,1,k} + W_{2,1,2} \bullet z_{2,2,k} \tag{12}$$

$$v_{2,k} - W_{2,2,0} + W_{2,2,1} \bullet z_{2,1,k} + W_{2,2,2} \bullet z_{2,2,k} \tag{13}$$

Thereafter, the following preliminary optimized weight matrices are obtained:

$$\text{For Layer1}: \begin{bmatrix} W_{1,1,0} & W_{1,1,1} & W_{1,2,1} \\ W_{1,2,0} & W_{1,2,1} & W_{1,2,2} \end{bmatrix}$$

$$\text{For Layer2}: \begin{bmatrix} W_{2,1,0} & W_{2,1,1} & W_{2,1,2} \end{bmatrix}$$

The matrices of the scaling factor for each layer are constructed to be of the same size as the weight matrices.

$$\text{For Layer1}: \begin{bmatrix} \theta_{1,1,0} & \theta_{1,1,1} & \theta_{1,2,1} \\ \theta_{1,2,0} & \theta_{1,2,1} & \theta_{1,2,2} \end{bmatrix}$$

$$\text{For Layer2}: \begin{bmatrix} \theta_{2,1,0} & \theta_{2,1,1} & \theta_{2,1,2} \end{bmatrix}$$

The values of the scaling factors θ control the impact of the weights on the objective function.

To find the sensitivity of the θ's with regard to the objective function, their values are individually perturbed by $\left(\pm\frac{1}{2}h\right)$, and the changes in the objective function are observed. The perturbation resulting in the largest objective function change corresponds to the most sensitive neuron and vice versa.

During the process implementation, all the scaling factors are initially set to $\theta = 1 + \frac{1}{2}\epsilon$, where ϵ is a small user-defined value, and the value of the objective function is determined. In the next step, the value of the θ's is changed to $\theta = 1 - \frac{1}{2}\epsilon$. Finally, the scaling factors are multiplied element-wise with the weights in each layer to gain a new objective value. This is effectively an element-wise matrix optimization.

To find the sensitivity for blocks of layers, all the values of the scaling factors, θ connected with the block of layers are perturbed at the same time, and the value of the objective function is evaluated. In this way, the sensitivities of any number of layers can be identified in combination through finite differences. This is easier compared with the automatic differentiation, where complicated derivatives need to be derived in order to evaluate the sensitivities of blocks of layers.

In the following the formulation of the approach based on finite differences is generalized. For $k = 1, ..., NK$ data points, the output of layer l given input form layer $l-1$ is:

$$z_{l,i,k} = f\left(y_{l,i,k}\right) \tag{14}$$

where:

$$y_{l,i,k} = \sum_{j=1}^{NN_{l-1}} W_{l,i,j} \bullet z_{l-1,j,k} + W_{l,i,0} \tag{15}$$

for $l = 1, 2, ..., NL$, $i = 1, 2, ..., NN_l$, and $j = 1, 2, ..., NN_{l-1}$. In Eq. (14), $W_{l,i,0}$ is the bias term.

The weights are subsequently obtained through the optimization of Eqs. (14) and (15), and the following matrices of θ's are defined for each layer of weights:

$$\text{For Layer 1 :} \quad \begin{bmatrix} \theta_{l,1,0} & \cdots & \theta_{l,1,NN_{L-1}} \\ \vdots & \ddots & \vdots \\ \theta_{l,NN_l,0} & \cdots & \theta_{l,NN_l,NN_{L-1}} \end{bmatrix}$$

where $\theta = 0$ or 1.

The values of $\frac{\partial MSE}{\partial \theta_{l',i'}}$ are found by perturbing the scaling factors θ based on Eq. (7).

An example formulation of the neural sensitivity analysis and the tuning of a small network through finite differences is provided in the Supplementary Material.

3.4 Validation with automatic differentiation

As discussed in the previous sections, the results from finite differences are less rigorously derived and calculated compared to the ones obtained from automatic differentiation. However, both methods serve to produce the same evaluation of sensitivities of individual neurons with respect to the objective. Thus, the two methods can serve to validate each other.

In both cases, for the example presented in the Supplementary Material, it was identified that the last layer has the highest sensitivity values, followed by layer 2, then by layer 1. Both results corroborate with each other in terms of neuron importance. The difference is that in the second and in the last layer, the calculated raw sensitivity values are different. For example, the sensitivities of the Layer 2 calculated using the automatic differentiation approach are of the order 10^{-3}, whereas in the case of the finite differences, they are of the order $10^{-5} - 10^{-4}$. This result can be attributed to the finite difference method being less rigorous, which can introduce errors based on the shape of the objective function.

It can be assumed that the deviation from the original point is minimal such that an accurate estimator of the gradient can be determined. Thus, under this assumption, the finite difference approach is as effective as the automatic differentiation method when used to perform ranking of neurons.

Further, the advantages of the two methods are compared. The key advantages of the automatic differentiation are:

- It does not rely on approximations and the calculated results are rigorously proved.
- It is faster than the finite difference method for a small network, when not counting the optimization step.
- All θ values are set to 1 or 0, hence easier to optimize or calculate.

Whereas in the case of the finite difference method, the following key advantages can be listed:

- It does not require an optimization run in order to generate the sensitivity values.
- It is easy to formulate, *i.e.*, no complicated mathematical derivations are required to find the sensitivity values.
- It is possible, without further derivations, to calculate the sensitivity values of a whole layer.
- The sensitivity values do not need to be limited to one neuron or one layer. Any block of layers or sub-layers can be evaluated for sensitivity with respect to the objective function.

Overall, the two methods have their own advantages. However, for more advanced applications, the finite difference method is easier to implement and more flexible to be adopted for different scenarios.

4 Hierarchical multi-scale topological and parametric optimization of DNNs

4.1 Hierarchical multi-scale structural contribution analysis of DNN's

The key idea behind the hierarchical multi-scale analysis of mathematical models of any type of system is to start from the highest level of abstraction, *i.e.*, the entire structure as an Input/Output (I/O) system. Then from this view-point, to effectively employ a set bisection approach that proceeds each time downwards in terms of the scale of detail considered at the nodes of an ever-expanding binary search tree. This continues with a bisection to refine the level of detail considered to be optimized during the major iterations of the proposed design algorithm.

Refinement of the binary search-tree sub-levels (nodes) continues until user-defined criteria are met or the final level of mathematical detail is reached during the refinement iterations of the algorithm.

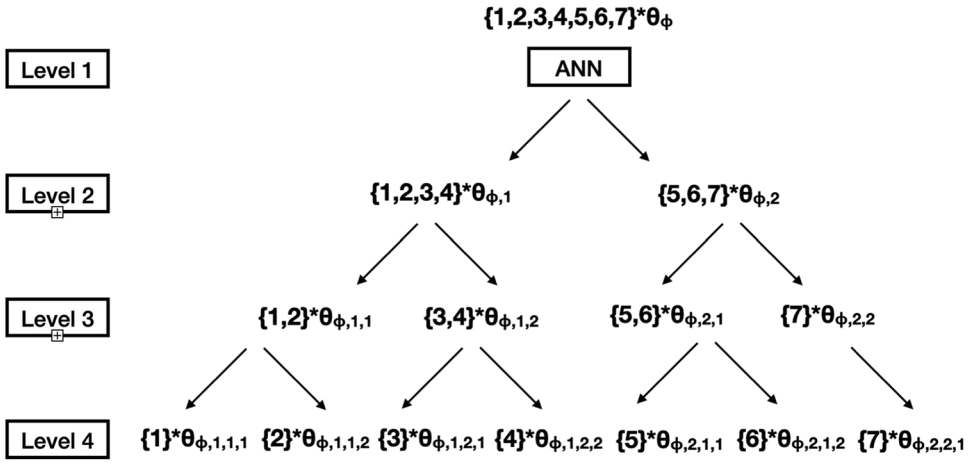


Fig. 1 The binary tree partitioning of the DNN with θ sensitivity parameters

To illustrate the approach, an example DNN is considered, with the following set of layers: an Input layer, an Output layer, and 7 hidden layers:

$$\text{DNN Layer Set} = \{\text{Input}, 1, 2, 3, 4, 5, 6, 7, \text{Output}\}$$

The structural optimization of the DNN topology (architecture) is going to be based on evaluating the novel structural sensitivity measures described in Section 3. Furthermore, to facilitate an efficient multi-scale maneuver of identifying rapidly large parts of the DNN that require structural optimization, it is necessary to introduce an arbitrary and yet a very natural way of viewing the hierarchically partitioning/decomposing decisions within the algorithm for the automated design optimization of the network. The binary tree partitioning of the DNN and the artificial θ-parameters for the structural sensitivity calculations are illustrated in Fig. 1.

Thus, following the proposed structural sensitivities calculation scheme, in 3 finite difference steps, the most sensitive/important layer in the DNN can be identified from the original 7 hidden layers set, as illustrated in Fig. 2.

Generally speaking, as an upper bound on the number of steps required to identify the most contributing layer in the DNN, at the most a number of $log_2 NL$ steps must be performed, where $_{NL}$ is the number of hidden layers in the network. Similarly, if each layer has a fixed number of neurons, the identification of the most sensitive neuron in the previously identified hidden layer will require at the most $log_2 NN$ finite difference steps, where $_{NN}$ refers to the number of neurons in a particular layer.

Overall, the effort to identify a single artificial neuron is given by:

$$\propto \lceil log_2 NL \rceil + \lceil log_2 NN \rceil \tag{16}$$

Noticeably, the hierarchical multi-scale analysis of the DNN system requires a logarithmic number of steps to reach any level of"scale" of encapsulation based on a binary, set bisection analysis tree.

It is noted that, for very large-scale DNNs, descending to the level of individual neurons or even layers is not required. In fact, in these situations, entire blocks of neurons within a layer, as well as entire blocks of hidden layers can be added or removed at a time according to the values of the structural sensitivity at any level of analysis using the binary tree set partitioning. In this case, each level of the binary tree signifies a different block size for local removal or addition in the topology/architecture of the DNNs designed and structurally optimized with the proposed scheme.

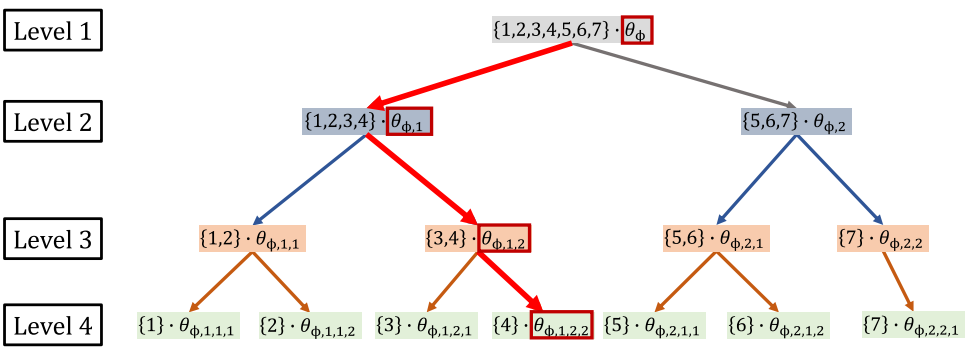


Fig. 2 Identification of the most important layer by assessing the structural sensitivity within the binary tree

With this scheme, the entire process can be fully automated in a real-time implementation of what is in essence a true "unsupervised" Machine Learning, a highly efficient and novel DNN design algorithm.

4.2 Network tuning

In the following section, an implementation of the proposed hierarchical scheme is used to tune the weights of the network. During tuning, the value of other weights is kept constant and the network is re-optimized only by changing the values of the neurons identified to be "most sensitive". An iterative process follows, where new sensitivity values are determined and the network is re-tuned, until a satisfactory objective value is found. Any method can be utilized for the optimization of the network, *i.e.*, backpropagation [69] or derivative-free algorithms (*e.g.*, Genetic Algorithms) [70].

In the following, the backpropagation method will be applied as an illustrative example. This choice is justified by the fact that this algorithm is the most often used to train DNNs, being accepted as the most successful learning procedure for these type of networks [71]. The parameters are updated using a rule corresponding to the optimization scheme. For the simplest Stochastic Gradient Descent method, the update rule is:

$$W' = W - \alpha \bullet \frac{dMSE}{dW} \tag{17}$$

where W's are the weights, α is the learning rate, and MSE is the objective function.

The number of iterations used in each optimization step can be arbitrarily small, as empirical implementation has demonstrated that even a small number of iterations can bring good optimization results by iteratively focusing only on the most sensitive layers. This is demonstrated in Section 4.2.1.

To find the most sensitive neuron layer to optimize, the binary tree search method is implemented. Starting from the first level of division, where the whole network is split into two parts, the focus is on the section of the network that has a higher sensitivity. Subsequently, the most sensitive block of layers is divided into two blocks, and so on until the smallest unit has a single layer, always choosing the higher sensitivity value along the path. In this way, the most sensitive layer can be found efficiently with $O\left(log_2 N\right)$ steps, where N is the total number of layers of the network.

At each iteration, the layer is optimized only by using the backpropagation method. After the optimization, another binary tree search is performed, and the procedure is repeated until the convergence criterion is fulfilled.

Initial implementation showed that the selection of the layers is often trapped in a loop, where the neuron having the highest sensitivity is always the same and is optimized repeatedly. To overcome this hurdle, a randomized algorithm is implemented, utilizing a probabilistic random factor in the selection of the neuron to optimize, similar to the ϵ-greedy algorithm in reinforcement learning [72–74]. This implementation of the random factor is a trade-off between"exploration" and"exploitation", as the neuron with lower sensitivity is allowed to have the chance to be optimized.

Another issue that is solved by introducing the random factor is related to the search when the left branch and the right branch of the binary tree have roughly the same sensitivity, which makes the selection process difficult. This is also frequently observed and requires a level of randomness in the selection of the layer to optimize.

When practically equal sensitivities are obtained at a branching point, this indicates the same influence for the left and right branches, respectively. This means that, to choose to branch left or right, the fact that the weight indicated by the structural sensitivity of a branching point should reflect the"probability" of that branch to be chosen must be taken into account. Overall, the proposed algorithm is deterministic, and will remain so with a judicious choice of a randomization step to break the problem of having to choose numerically equal branching at points.

The algorithm is developed as such:

Suppose S_{left} and S_{right} are the respective absolute values of the structural sensitivities at a branching point calculated during the level exploration phase. These sensitivities are used to arrive at the next layer whose tuning weights are to be optimized together, while holding all other weights of all other layers constant, *i.e.*,:

$$S_{left} = AbsoluteValue(s_{left}) \tag{18}$$

$$S_{right} = AbsoluteValue(s_{right}) \tag{19}$$

where s_{left} and s_{right} are the sensitivity values on the left and right branch at the branching point.

Using the absolute values of the structural sensitivities at a branching binary search tree exploration step, choose the left or right branch based on a randomized algorithm which is applied irrespective of the values of the structural sensitivities. The following steps must be followed:

1. Generate a random real number (the random factor) between 0 and 1:

$$r = RandomReal([0, 1]) \tag{20}$$

2. Calculate the probability for the left and right branch, respectively:

$$Probability_{BranchLeft} = S_{left} / \left(S_{left} + S_{right}\right) \quad (21)$$

$$Probability_{BranchRight} = 1 - Probability_{BranchLeft} \quad (22)$$

3. Make use of the random factor. If branch left probability is greater than the random factor, branch left. If not, branch right.

The random factor is used throughout the binary search process and when used for a sufficiently long time, it will be able to break"ties" among the branching points in the binary search tree. The pseudo-code of the algorithm is presented in Table 2. Furthermore, the hierarchical multi-scale training procedure is illustrated in Fig. 3.

4.2.1 Example tuning

In this section, the proposed procedure is run for an example of the tuning of a DNN. As the focus is only on the tuning functionality, no changing of architecture is involved yet, and the adopted architecture is arbitrary.

In this case, an architecture of [4, 5, 5, 5, 5, 5, 5, 5, 5, 2] is adopted. The optimization is performed with the hyperparameters tabulated in Table 3. The total number of data points used is 10,000.

The parameters in Table 3 have the following meaning: the Learning rate determines how fast the gradient is reduced in each step, the Number of epochs is the number of times the optimization is run during each iteration. The Maximum number of iterations determines the total number of iterations the algorithm performs, while ϵ is the trust region bound.

Figure 4 demonstrates the change in sensitivity values over the process of optimizing the network one layer at a time. The 3D bar chart on the top of each figure illustrates how the sensitivity values change for each division along the architecture. Thus, 2 values are observed in the first level of division, 4 in the second level, etc. This is because the total number of layers are divided by 2 each time. The line plot on the bottom represents the variations in the value of the objective function. The figures represent the optimized results at iterations 1, 20, 40, 60, 80 and 100, respectively.

From Fig. 4, it can be observed that the sensitivities are quickly equilibrated with a minimal number of iterations. In the first iteration (Fig. 4a), the sensitivity value is quite high for division on the right-hand-sides. Only three bars are visible in this case because their sensitivity values are so high that the display for the other bars is suppressed. This demonstrates that the sensitivity values are significantly different at the initiation stage.

At iteration 20 (Fig. 4b), the sensitivity values have been equilibrated, *i.e.*, the values are very similar (although in raw numbers they are different). The values of the objective has been decreasing and there are three plateaus in the objective value. These plateaus correspond to local minima and this result demonstrates that the proposed algorithm is capable of overcoming the local minima. A local minima that takes a long time to be overcome is displayed in Fig. 4d, where a large plateau is observed after 17 iterations, which is shown to be overcome in Fig. 4e.

At iteration 100 (Fig. 4e), the objective value keeps decreasing while the sensitivity values are roughly similar. This demonstrates that all neurons are playing an important part in the optimization process and the objective value is decreasing with further optimizations.

With the introduction of the random factor, the layers that are optimized rotate among available layers instead of looping around a few. The distribution of the layers being optimized during the 100 iterations is plotted in Fig. 5. It can be observed that there is a reasonable distribution of layers being optimized and the optimization is no longer stuck in a loop and only optimizing a single layer with the highest sensitivity.

4.2.2 Comparison with the end-to-end backpropagation algorithm

As mentioned previously, the conventional method to optimize DNNs is using backpropagation methods [75, 76]. These methods are considered essential and the de facto solution for the efficient training and good generalization of large-scale DNNs [77]. Thus, the performance of the proposed algorithm is compared with the end-to-end backpropagation method. The iterations for both the end-to-end backpropagation and the hierarchical multi-scale search algorithms are set with the same values of the hyperparameters (Table 4). Each time, the end-to-end backpropagation algorithm optimizes all layers instead of one selected layer, as in the case of the proposed algorithm. The same architecture of the network ([4, 5, 5, 5, 5, 5, 5, 5, 5, 2]) is used as the one proposed for the example tuning in Section 4.2.1.

The end-to-end backpropagation algorithm is run for 1,000 number of epochs in each iteration. The number of epochs is arbitrarily set this large to ensure optimization to the optimal point. Having the same settings for the two algorithm allows comparison between them.

In backpropagation, the derivative of the objective function with regard to the weights is calculated at every epoch [78]. Therefore, the check is performed every round instead of for every 50 rounds.

Moreover, for comparison purposes, the same convergence criterion is adopted for the two algorithms investigated.

Table 2 Pseudo-code of the algorithm for the network tuning

Algorithm 1 Network tuning adopting the hierarchical multi-scale parametric optimization

Read training and testing I/O data points
Initialise: Architecture of neural network
Initialise: Sensitivity (θ) matrices and weight (W) matrices for each layer
Initialise: A user-defined infinitesimal value ϵ used for perturbation

procedure REFRESH_THETA(Network Structure
    Define θ matrices based on the Network Structure. Set θ matrices values to be 1
    return θ matrices
end procedure

procedure PERTURB_THETA(Network Structure, Perturbation, Leftmost Layer To Perturb, Number Layers To Perturb, Perturbation Side)
    REFRESH_THETA(Network Structure)
    Choose Perturbation side
    Layers to perturb are found by:
    if <Perturbation side on the left> **then**
        Left bound to perturb = Leftmost Layer To Perturb
        Right bound to perturb = Leftmost Layer To Perturb + $\frac{1}{2}$ · Number Layers To Perturb
    else
        Right bound to perturb = Leftmost Layer To Perturb + $\frac{1}{2}$ · Number Layers To Perturb + 1
        Right bound to perturb = Leftmost Layer To Perturb + Number Layers To Perturb
    end if
    for <layers in the left bound to right bound> **do**
        Set θmatrices values to be 1−Perturbation
    end for
    return θ matrices
end procedure
procedure WEIGHT_TIMES_THETA(Network Structure, Theta Matrices, Weight Matrices)
    Calculate the element-wise product of θ and W matrices
    Assign the values to New Weight Matrices
    return New Weight Matrices
end procedure

procedure CALCULATE_SENSITIVITIES(Theta Matrices, Weight Matrices, Network Structure, Leftmost Layer To Calculate, Number Of Layers To Calculate)
    Initialise Weight Matrices
    Forward propagate the weights in Weight Matrices
    Calculate the objective function value, denoted as Original Loss

    Initialise θ with REFRESH_THETA(Network Structure)
    PERTURB_THETA(Network Structure, ϵ, Leftmost Layer To Calculate, Number Of Layers To Calculate, Left Side)
    New Weight Matrices = WEIGHT_TIMES_THETA(Network Structure, Theta Matrices, Weight Matrices)
    Forward propagate the weights in New Weight Matrices
    Calculate the objective function value, denoted as Left Side Loss
    Left Sensitivity = (Left Side Loss - Original Loss) / ϵ

    Initialise θ with REFRESH_THETA(Network Structure)
    PERTURB_THETA (Network Structure, ϵ, Leftmost Layer To Calculate, Number Of Layers To Calculate, Right Side)
    New Weight Matrices = WEIGHT_TIMES_THETA (Network Structure, Theta Matrices, Weight Matrices)
    Forward propagate the weights in New Weight Matrices
    Calculate the objective function value, denoted as Right Side Loss
    Right Sensitivity = (Right Side Loss - Original Loss) / ϵ
    return Left Sensitivity, Right Sensitivity
end procedure

Table 2 (continued)

```
procedure TRAIN_MODEL(X Train, Y Train, Maximum Number of Epochs, Left Bound, Right Bound, Iteration Number)
    if <Iteration Number == 1> then
        Initialise parameters randomly
    else
        Populate previously populated parameters
    end if
    for <Increasing Epochs> do
        Forward propagate X Train values in the network
        Calculate the objective value
        Backpropagate to find the gradients of parameters and update Parameters
        for <layers in Left Bound to Right Bound> do
            Alter layer parameters by the update rule
        end for
    end for
    return Parameters, Cost
end procedure

procedure CALCULATE_PARTIAL_DERIVATIVE(Gradients, Norm=1)
    if <Norm == 1> then
        Value = Sum of absolute values of Gradients
    else if <Norm == 0> then
        Value = Maximum value in Gradients
    end if
    return Value
end procedure
```

```
procedure FIND_LEFT_RIGHT_BOUND(Sensitivity)
    Align Sensitivity into Levels
    for <each Level> do
        Obtain Left Sensitivity and Right Sensitivity
        Calculate Left Probability and Right Probability
        Generate a Random Number between 0 and 1
        if <Left Probability >= Random Number> then
            Choose left side
        else
            Choose right side
        end if
        Identify Leftmost Layer of interest, and Number of Layers Branch (left and right)
        if <Last Level> then
            Left Bound = Leftmost Layer
            Right Bound = Leftmost Layer + 1
        else
            Left Bound = Leftmost Layer of interest
            Right Bound = Leftmost Layer + Number of Layers in Branch
        end if
    end for
    return Left Bound, Right Bound
end procedure
```

```
Initialise: Parameters, Backpropagation Derivative, Left Bound, Right Bound, Left Bound List, Right Bound List
Input: X Train, Y Train, Network Architecture, Tolerance, Maximum Number of Epochs, Maximum Number of Iterations
Count: Iteration Number
while <Backpropagation Derivative > Tolerance> do
    Left Bound = last element in Left Bound List
    Right Bound = last element in Right Bound List
    Parameters, Cost = TRAIN_MODEL (X Train, Y Train, Maximum Number of Epochs, Left Bound, Right Bound,
    Iteration Number)
    Left Bound, Right Bound = FIND_LEFT_RIGHT_BOUND (Sensitivity)
    if <Iteration Number % 50 == 0> then
        Backpropagate to obtain gradients of MSE with respect to weights
        Backpropagation Derivative = CALCULATE_PARTIAL_DERIVATIVE (Gradients)
    end if
    if <Iteration Number > Maximum Number of Iterations> then
        Break
    end if
    Append Left Bound to Left Bound List
    Append Right Bound to Right Bound List
end while
Output: Cost, Left Bound List, Right Bound List, Parameters
```

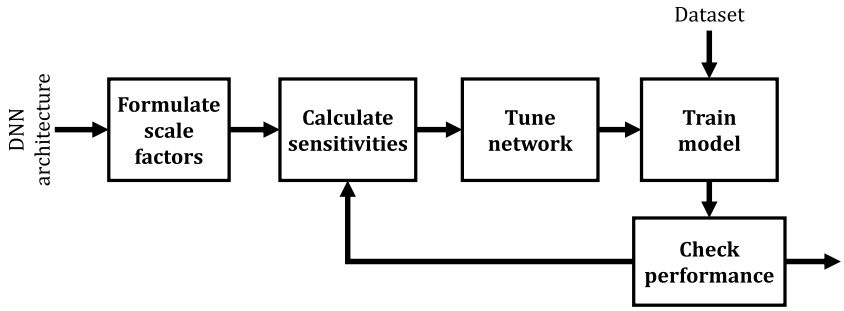


Fig. 3 Hierarchical multi-scale training procedure of DNNs

Hyperparameters The values of the hyperparameters used in the comparison studies of the two methods are listed in Table 4. They are selected arbitrarily with a random search for optimality.

Objective The objective value obtained from the end-to-end optimization is 0.992952, while for the hierarchical multi-scale method the value is 0.992834. These values of the objectives are comparable, with the hierarchical multi-scale search method reaching a slightly lower objective value. The objective function values are plotted against the number of iterations in Fig. 6.

Optimization time The CPU time taken for the end-to-end optimization is 111.587 s, while 2,773.410 s are required for the hierarchical multi-scale approach. Although optimizing to a smaller objective value, the hierarchical multi-scale search method is taking longer.

Weight values The values of the weights provide information on whether the two algorithms arrive at the same local minima. Their distribution is plotted in Fig. 7, where it can be observed that the weights are distributed in a similar way. This indicates that there is a high possibility that the two methods optimize to the same local minima, which is further corroborated with the fact that the values of the objective function are very similar.

Table 3 The hyperparameters used in formulation for the network tuning example

Hyperparameter	Value
Architecture	[4, 5, 5, 5, 5, 5, 5, 5, 2]
Trust region bound, ϵ	0.001
Learning rate	0.01
Number of epochs in each iteration	1,000
Upper limit on number of iterations	100

5 Use of second-order information in the binary search tree

5.1 Mathematical formulation

In this section, the hierarchical multi-scale approach is modified to include second order information on the structural sensitivity. This is based on the local second order Taylor expansion of a function $f(x, y)$:

$$f(x+\Delta x, y+\Delta y) = f(x,y) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(\Delta x)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial y^2}(\Delta y)^2 + \frac{\partial^2}{\partial x \partial y}(\Delta x \Delta y) + \textit{Higher Order Terms } (H.O.T.) \tag{23}$$

If second order information is recorded at least at the bottom layer of the binary decomposition tree 2 variables can be modified simultaneously if required during the simulation and/or optimization tasks. Furthermore, the structural sensitivity model is calculated to be quadratic at every branching point, *e.g.*, by computing first and second order finite differences to obtain the information for left ("1") and right ("2"), respectively:

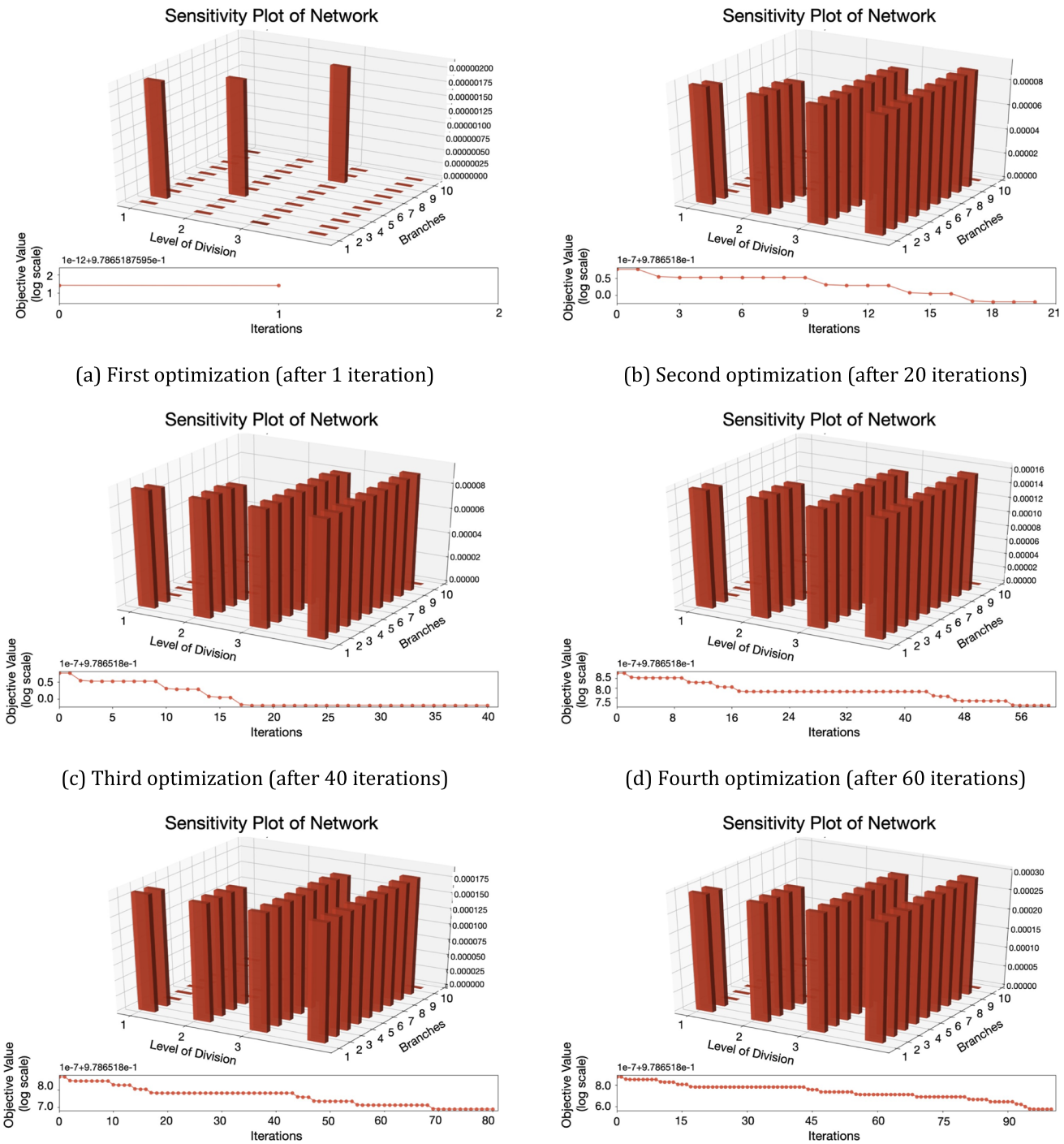
$$f(\theta_1+\Delta\theta_1, \theta_2+\Delta\theta_2) = f(\theta_1,\theta_2) + \frac{\partial f}{\partial \theta_1}\Delta\theta_1 + \frac{\partial f}{\partial \theta_2}\Delta\theta_2 + \frac{1}{2}\frac{\partial^2 f}{\partial {\theta_1}^2}(\Delta\theta_1)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial {\theta_2}^2}(\Delta\theta_2)^2 + \frac{\partial^2}{\partial\theta_1\partial\theta_2}(\Delta\theta_1\Delta\theta_2) + \textit{Higher Order Terms } (H.O.T.) \tag{24}$$

In the following, the approach to exploit the second order information in the binary tree-based decomposition search will be demonstrated. To this end, in order to decide which branch to follow, assuming that this happens only for a single branch, and only in the last level of decomposition, one variable is changed at a time, for consistency.

Given a quadratic model in $\delta_1 \triangleq \Delta\theta_1$ and $\delta_2 \triangleq \Delta\theta_2$:

$$f(\theta_1+\delta_1, \theta_2+\delta_2) = a + b\delta_1 + c\delta_2 + d\delta_1\delta_2 + e{\delta_1}^2 + f{\delta_2}^2 \tag{25}$$

and a trust region:



(a) First optimization (after 1 iteration)

(b) Second optimization (after 20 iterations)

(c) Third optimization (after 40 iterations)

(d) Fourth optimization (after 60 iterations)

(e) Fifth optimization (after 80 iterations)

(f) Sixth optimization (after 100 iterations)

Fig. 4 Optimization results for the example formulation using 10,000 data points

$$\begin{cases} -\epsilon_1 \leq \delta_1 \leq \epsilon_1 \\ -\epsilon_2 \leq \delta_2 \leq \epsilon_2 \end{cases}$$

the above quadratic model (even considering only sufficiently small quantized discrete-size steps: $\delta_1 = \pm\epsilon_1$, $\delta_2 = \pm\epsilon_2$) can be optimized and targets can be assigned to change the next sublevel values by δ_1^*, δ_2^* :

$$\theta_1^{new} = \theta_1^{old} + \delta_1^*$$

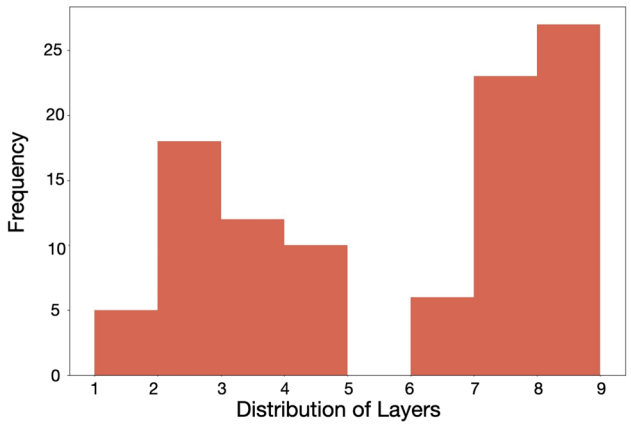


Fig. 5 The distribution of the layers selected in the optimization process

$$\theta_1^{new} = \theta_2^{old} + \delta_2^*$$

Whether the optimal steps δ_1^*, δ_2^* are quantized or continuously valued within the trust region, they require cascading down the evaluation tree of the model. Coming to a point where there is a multiple linked leaf, deciding on a simple updating criterion will become very challenging due to this coupling.

An alternative way is to use the quadratic model to extract further information in computing the left-and-right selection probabilities in the non-deterministic, randomised left-or-right selector at the binary tree branching points.

Given:

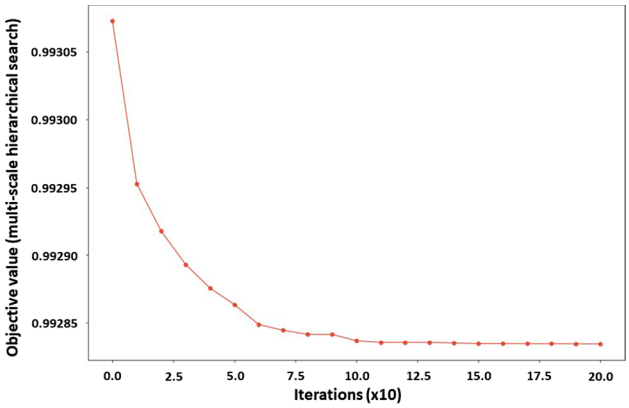
$$q(\delta_1, \delta_2) = a + b\delta_1 + c\delta_2 + d\delta_1\delta_2 + e{\delta_1}^2 + f{\delta_2}^2 \tag{26}$$

a local approximation model for the two gradient elements is obtained:

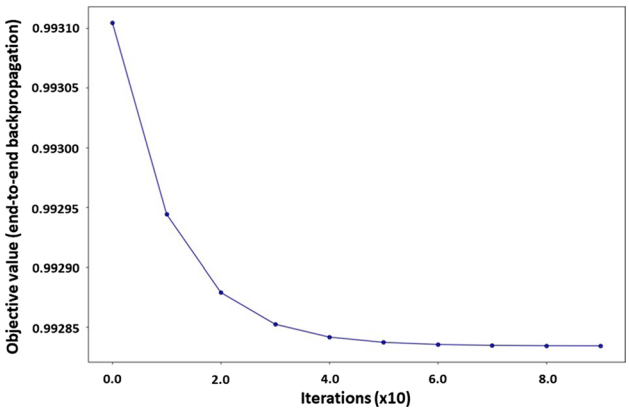
$$\frac{\partial q}{\partial \delta_1} = b + d\delta_2 + 2e\delta_1 \tag{27}$$

Table 4 The hyperparameters used for the performance comparison between the end-to-end backpropagation and the hierarchical multi-scale search methods

Hyperparameter	Value
Architecture	[4, 5, 5, 5, 5, 5, 5, 5, 5, 2]
Trust region bound,ϵ	0.001
Learning rate	0.0001
Number of epochs in each iteration	1,000
Upper limit on number of iterations	200
Tolerance	0.001
Number of data points	10,000



(a) Hierarchical multi-scale search



(b) End-to-end backpropagation

Fig. 6 Comparison between the optimization performance of the end-to-end backpropagation and the hierarchical multi-scale search methods

$$\frac{\partial q}{\partial \delta_2} = c + d\delta_1 + 2f\delta_2 \tag{28}$$

Both expressions above are independent of the trust region parameters.

To modify the selector (non-deterministic randomized element) in a simple way, the selection is based on the average value of the above gradient elements within the trust region of interest. Therefore:

$$\left[\frac{\partial q}{\partial \delta_1}\right] = \frac{1}{4\epsilon_1\epsilon_2}\int_{\theta_1-\epsilon_1}^{\theta_1+\epsilon_1}\int_{\theta_2-\epsilon_2}^{\theta_2+\epsilon_2}\left(b + d\delta_2 + 2e\delta_1\right)d\delta_2 d\delta_1 \tag{29}$$

where the $\frac{1}{4\epsilon_1\epsilon_2}$ term comes from $\int_A \int 1 \bullet d\delta_2 d\delta_1$, for $A = \{\theta_1 - \epsilon_1 \le \delta_1 \le \theta_1 + \epsilon_1, \theta_2 - \epsilon_2 \le \delta_2 \le \theta_2 + \epsilon_2\}$.

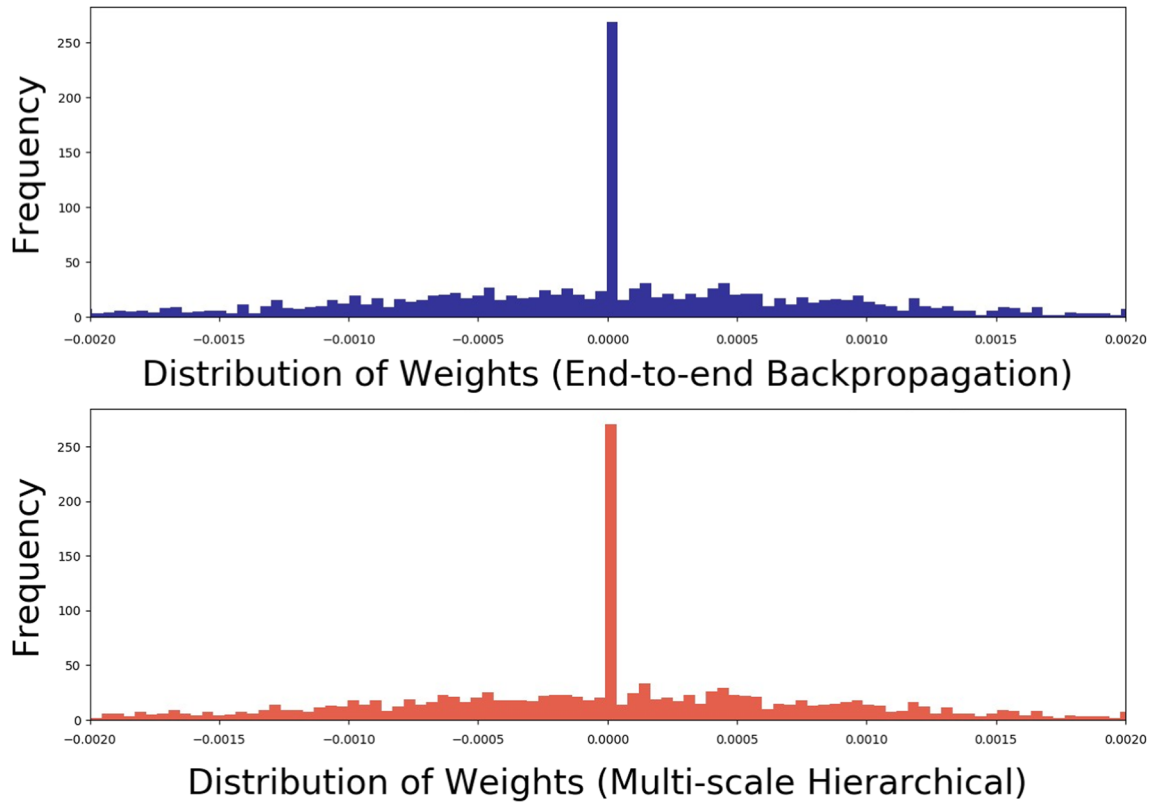


Fig. 7 The distribution of weights obtained from end-to-end backpropagation and hierarchical multiscale optimization

$$\begin{aligned}\left[\frac{\partial q}{\partial \delta_1}\right] &= \frac{1}{4\epsilon_1\epsilon_2}\int_{\theta_1-\epsilon_1}^{\theta_1+\epsilon_1}\left[b+d\delta_2+2e\delta_1\right]_{\theta_2-\epsilon_2}^{\theta_2+\epsilon_2}d\delta_1 \\ &= \frac{1}{4\epsilon_1\epsilon_2}\int_{\theta_1-\epsilon_1}^{\theta_1+\epsilon_1}\Big[b(\theta_2+\epsilon_2)+\frac{1}{2}d(\theta_2+\epsilon_2)^2 \\ &+2e\delta_1(\theta_2+\epsilon_2)-b(\theta_2-\epsilon_2)-\frac{1}{2}d(\theta_2-\epsilon_2)^2-2e\delta_1(\theta_2-\epsilon_2)\Big]d\delta_1\end{aligned} \tag{30}$$

$$\begin{aligned}\left[\frac{\partial q}{\partial \delta_1}\right] &= \frac{1}{4\epsilon_1\epsilon_2}\int_{\theta_1-\epsilon_1}^{\theta_1+\epsilon_1}\left(2b\epsilon_2+2d\theta_2\epsilon_2+4e\delta_1\epsilon_2\right)d\delta_1 \\ &= \frac{1}{4\epsilon_1\epsilon_2}\left[\left(2b\epsilon_2+2d\theta_2\epsilon_2\right)\delta_1+2e{\delta_1}^2\right]_{\theta_1-\epsilon_1}^{\theta_1+\epsilon_1} \\ &= \frac{1}{4\epsilon_1\epsilon_2}\left[\left(2b\epsilon_2+2d\theta_2\epsilon_2\right)\bullet 2\epsilon_1+2e\epsilon_2\bullet 4\theta_1\epsilon_1\right]\end{aligned} \tag{31}$$

Thus, the end result is:

$$\left[\frac{\partial q}{\partial \delta_1}\right] = b+d\theta_2+2e\theta_1 \tag{32}$$

From Eq. (26), the following is obtained symmetrically:

$$\left[\frac{\partial q}{\partial \delta_2}\right] = b+d\theta_1+2e\theta_2 \tag{33}$$

Both Eqs. (32) and (33) are independent of the trust region parameters. The perturbation of the θ parameters is such that:

$$\theta_1 = \theta_2 \triangleq 1+\tau \tag{34}$$

where τ is an infinitesimal value.

From Eqs. (32) and (33) the following is obtained:

$$\left[\frac{\partial q}{\partial \delta_1}\right] = b+d+2e \tag{35}$$

$$\left[\frac{\partial q}{\partial \delta_2}\right] = b+d+2f \tag{36}$$

which are independent of the trust region parameters as well.

These equations are going to be used instead of the point-wise gradient elements for θ_1 and θ_2 in order to select which branch to follow, left or right.

The randomized selector step is then defined by Eq. (20) and:

$$r_g = \frac{\left|\frac{\partial q}{\partial \delta_1}\right|}{\left|\frac{\partial q}{\partial \delta_1}\right|+\left|\frac{\partial q}{\partial \delta_2}\right|} \tag{37}$$

If $r \le r_g$, the left branch (θ_1) is selected. If $r \ge r_g$, the right branch (θ_2) is selected.

The previous results on the average value of the gradients require the calculation of the second and first order derivatives by finite differences.

The first order derivatives can be computed by:

$$\frac{\partial f(\theta_1,\theta_2)}{\partial \theta_1} \approx \frac{f(\theta_1+h_1,\theta_2)-f(\theta_1,\theta_2)}{h_1}+O(h_1) \tag{38}$$

or:

$$\frac{\partial f(\theta_1,\theta_2)}{\partial \theta_1} \approx \frac{f(\theta_1+h_1,\theta_2)-f(\theta_1-h_1,\theta_2)}{2h_1}+O({h_1}^2) \tag{39}$$

while the second order derivative:

$$\begin{aligned}\frac{\partial^2 f}{\partial {\theta_1}^2} &\approx \frac{\frac{f(\theta_1+h)}{\partial\theta_1}-\frac{f(\theta_1-h)}{\partial\theta_1}}{2h} \approx \frac{\frac{f(\theta_1+h,\theta_2)-f(\theta_1,\theta_2)}{h}-\frac{f(\theta_1,\theta_2)-f(\theta_1-h,\theta_2)}{h}}{2h} \\ &\approx \frac{1}{2h^2}\left[f(\theta_1+h,\theta_2)-2f(\theta_1,\theta_2)+f(\theta_1-h,\theta_2)\right]\end{aligned} \tag{40}$$

In the same way the values of $\frac{\partial f}{\partial \theta_2}$ and $\frac{\partial^2 f}{\partial {\theta_2}^2}$ are obtained symmetrically.

The second order mixed derivatives are:

$$\begin{aligned}\frac{\partial^2 f}{\partial\theta_1\partial\theta_2} &\approx \frac{\frac{f(\theta_1,\theta_2+h_2)}{\partial\theta_1}-\frac{f(\theta_1,\theta_2-h_2)}{\partial\theta_1}}{2h_2} \\ &\approx \frac{\frac{f(\theta_1+h_1,\theta_2+h_2)-f(\theta_1-h_1,\theta_2+h_2)}{2h_1}-\frac{f(\theta_1+h_1,\theta_2-h_2)-f(\theta_1-h_1,\theta_2-h_2)}{2h_1}}{2h_2}\end{aligned} \tag{41}$$

In summary:

$$\begin{aligned}\frac{\partial^2 f}{\partial\theta_1\partial\theta_2} \approx \frac{1}{4h_1h_2}\Big[&f(\theta_1+h_1,\theta_2+h_2)-f(\theta_1-h_1,\theta_2+h_2) \\ &-f(\theta_1+h_1,\theta_2-h_2)+f(\theta_1-h_1,\theta_2-h_2)\Big]\end{aligned} \tag{42}$$

Furthermore, the heuristic:

$$\Delta_x = 100 \bullet \sqrt{MachinePrecision} \bullet \max\{|x|, 1.0\} \tag{43}$$

is used for the finite difference calculations.

5.2 Results and analysis

An example run of the optimization is performed adopting the second-order information, as described in the previous section, to select left or right at the branching point. Similarly, the architecture of [4, 5, 5, 5, 5, 5, 5, 5, 5, 2] is adopted for the network, as an example to investigate the effect of incorporating second-order information. The number of data points used is 10,000. The same set of hyperparameters are adopted as enlisted in Table 3.

Figure 8 demonstrates the sensitivity and the objective function values over the course of the optimization, at iterations 1, 10, 20, 30, 40 and 50, respectively. The optimization reaches the tolerance value at exactly 50 iterations.

From these results, it can be observed that the objective value decreases over time, with convergence to a low value within approximately 20 iterations. Moreover, a quick equilibration of the sensitivity values is observed with this tuning scheme. Within 10 iterations, the values of the sensitivities become almost equal across layers, and the equilibrium is maintained over further iterations.

The distribution of the layers selected in the optimization process is plotted in Fig. 9. It can be observed that the first and the last layers are more frequently updated. However, with the addition of the random factor, other layers have a possibility of being selected as well.

Compared to the optimization using only the first-order information, for the proposed scheme, the same effect of equilibration and fast convergence is observed. However, in this case, the values of the sensitivities are higher due to the introduction of ϵ^2 at the denominator, as shown in Eq. (29).

The other difference is that the proposed approach converges faster, within 50 rounds of iterations, indicating a more direct search direction brought about by the utilization of the second-order information.

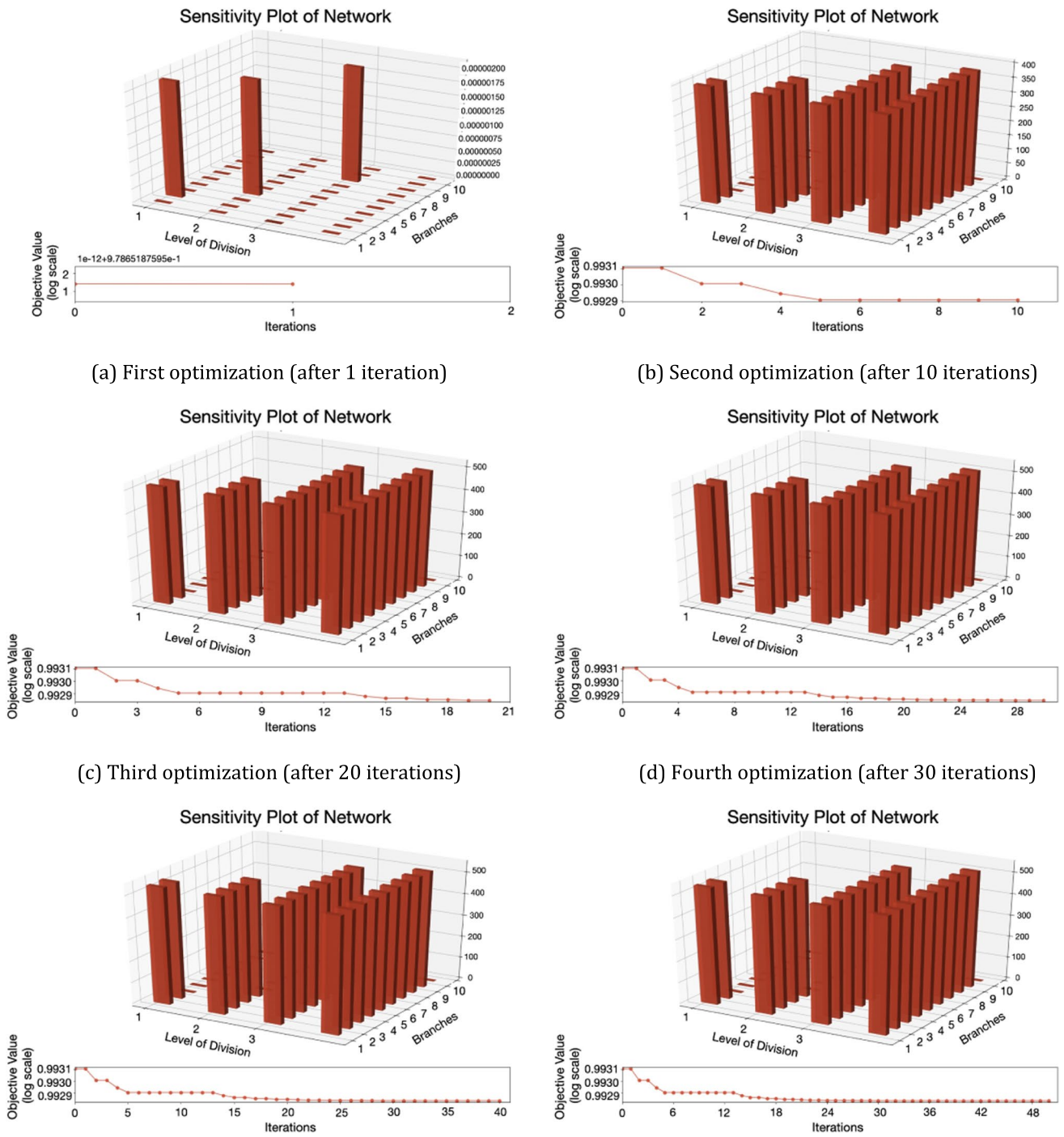
5.3 Comparison with end-to-end training

The results of including the second order information into the multiscale hierarchical search approach is now compared with the end-to-end backpropagation results.

Hyperparameters The hyperparameters adopted for both the end-to-end and the hierarchical multi-scale training are enlisted in Table 3. It is the same set of hyperparameters used in the case of the optimization using the first-order information.

Objective The objective value obtained from the end-to-end optimization is 0.99283426, while for the hierarchical multi-scale search method a value of 0.99283424 is obtained. The values are comparable, with the hierarchical multi-scale optimization arriving again at a slightly lower objective.

Optimization time The CPU time required for the end-to-end optimization is 358.165 s in this case. In comparison, the CPU time in the case of the hierarchical multi-scale approach is 762.714 s. If the number of iterations is compared, the backpropagation takes 99 iterations to reach the convergence criterion, whereas the hierarchical multi-scale method requires only 50 iterations. The plot of the optimization process is shown in Fig. 10. Separate graphs are drawn since the number of iterations is different and, thus, they are not directly comparable. One iteration in the end-to-end backpropagation optimization



(a) First optimization (after 1 iteration)

(b) Second optimization (after 10 iterations)

(c) Third optimization (after 20 iterations)

(d) Fourth optimization (after 30 iterations)

(e) Fifth optimization (after 40 iterations)

(f) Sixth optimization (after 50 iterations)

Fig. 8 Optimization results for the example formulation adopting second-order information and using 10,000 data points

corresponds to one update of the weights across all layers, whereas one iteration in the hierarchical multi-scale search approach corresponds to one update of a single layer obtained from the binary tree search.

Weight values Similarly, from the distribution plots presented in Fig. 11, it can be observed that the weights are roughly similar, thus the two algorithms are highly likely to optimize to the same local minima, conclusion further

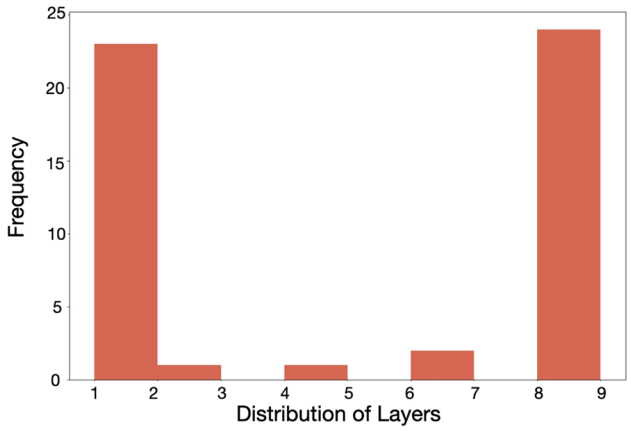
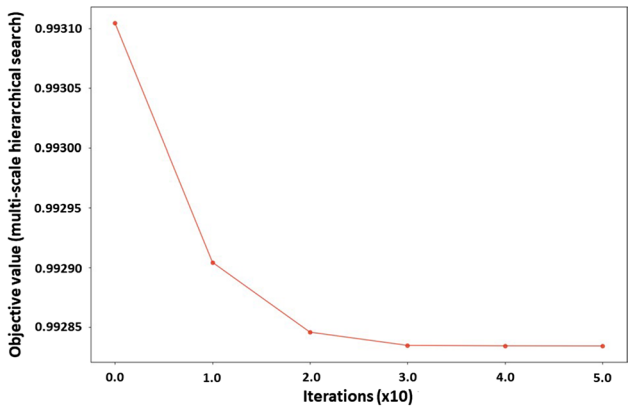
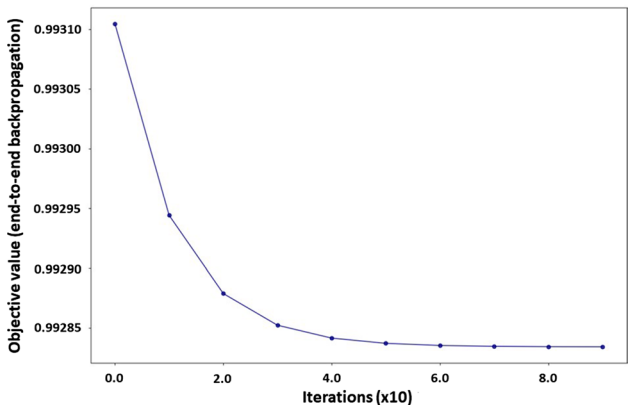


Fig. 9 The distribution of the layers selected in the optimization process adopting second-order information



(a) Hierarchical multi-scale search



(b) End-to-end backpropagation

Fig. 10 Comparison between the optimization performance of the end-to-end backpropagation and the hierarchical multi-scale search methods

corroborated by the similarity in the values of the objective function obtained.

6 Comparison between the first- and the second-order search algorithms

The performance of the binary tree search adopting first-order information is compared against the search utilizing second-order information by assuming the same set of hyperparameters to evaluate the search criterion. Overall, the optimization adopting second-order information is faster, with a total CPU time of 762.714 s and a total number of iterations of 50. This draws comparison to the optimization adopting first-order information, with a total CPU time of 2,773.410 s and a total number of iterations of 200.In the analysis above, the number of epochs in each iteration (which means used to optimize each layer of the network) is fixed to be 1,000.

To further compare how the incorporation of second-order information improves the optimality search by reducing the operation time, a convergence criterion is inserted during each optimization iteration. This convergence criterion is defined as follows:

$$\| \frac{\partial MSE}{\partial W_{layer}} \|_{inf} < tolerance \tag{44}$$

where W_{layer} refers to the weights in the layer being optimized, and the *tolerance* is a user-defined input value which can be equal to the convergence criterion of the whole optimization problem.

By comparing the infinity norm of the weights in the layer tuned to a tolerance value, further optimization of the layer is stopped if the value of this *tolerance* is met. Thus, the number of epochs in each iteration will be less than 1,000.

In comparison, the convergence criterion for the whole optimization problem is:

$$\| \frac{\partial MSE}{\partial W_{network}} \|_{inf} < tolerance \tag{45}$$

where $W_{network}$ refers to all the weights in the network.

Practical implementation shows that 10^{-4} is a better tolerance value for each iteration, while for the overall optimization a 10^{-3} value should be used.

To demonstrate how the incorporation of second-order information improves the optimization speed, the average value of the number of epochs, the CPU time and the infinity norm of the gradient values at each iteration are recorded. The comparison results are shown in Table 5. The data is split into two sets, one used for training the network ("Training set") and one for a validation step ("Testing set").The

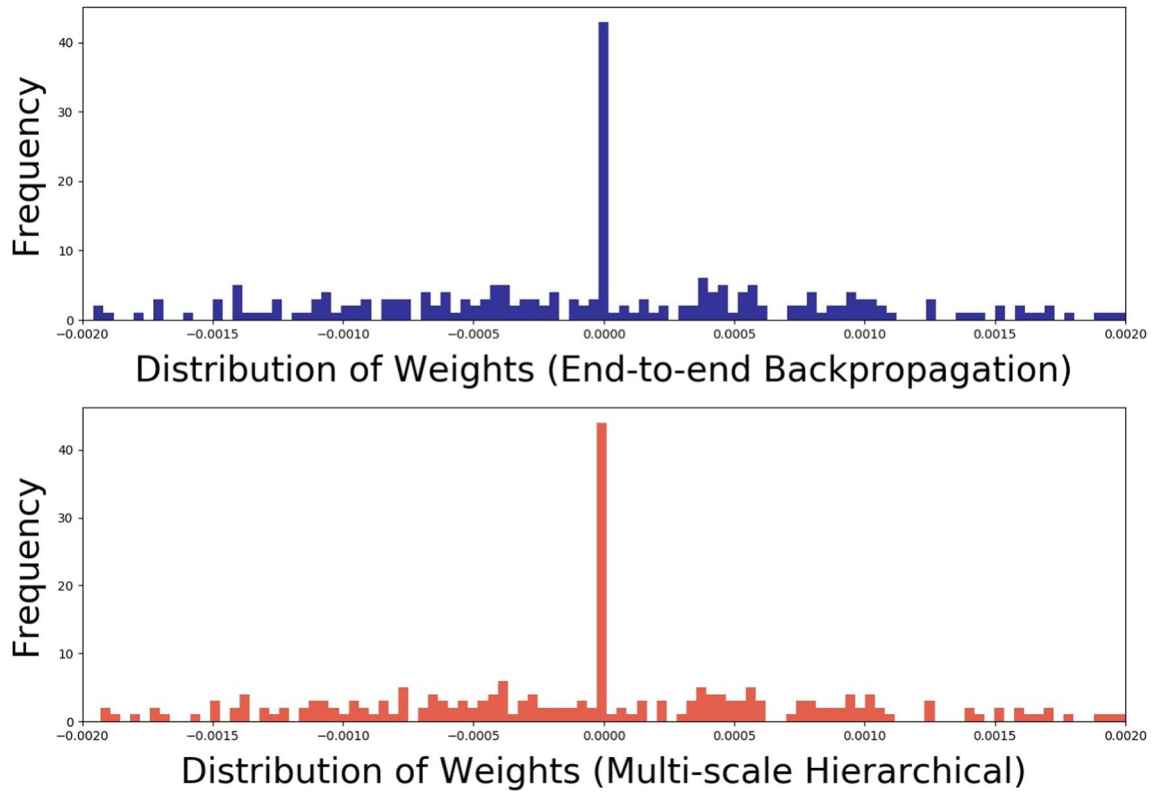


Fig. 11 The distribution of the weights obtained from the end-to-end backpropagation and hierarchical multi-scale search algorithms with second-order information

average values are obtained at each iteration. The total values are obtained for all iterations.

After adding a convergence criterion at each iteration, it is observed that the adoption of the second-order sensitivity as the selection benchmark requires less time compared to the first-order. The average CPU time and the average number of epochs in each iteration is smaller when the second-order information is adopted. Although the total number of iterations is higher for the second-order case, this is compensated with less search time at each iteration.

Comparing with the classical backpropagation method, the second-order approach requires a comparable, or even less CPU time per iteration. However, routinely, the backpropagation method is the fastest algorithm overall with the second least total number of iterations. Although this approach outperforms both the first- and the second-order processes in CPU time, the first-order hierarchical multi-scale search requires a lower number of iterations, while the second-order is faster per iteration.

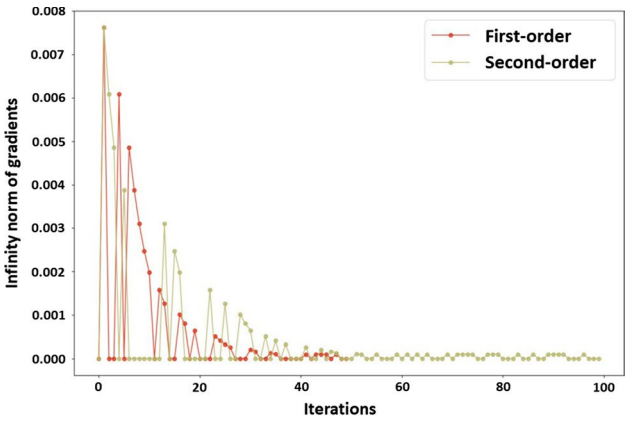
Table 5 Comparison between the binary tree search based on first- and second-order information, against backpropagation

Order	First	Second	Backpropagation
Training set MSE	0.98809	0.83099	0.98177
Testing set MSE	0.98810	0.83009	0.98057
Average epochs	405.7	203.4	200.0
Total iterations	51	101	99
Total CPU time (s)	350.4	310.9	297.3
Average CPU time (s)	5.746	3.013	3.022

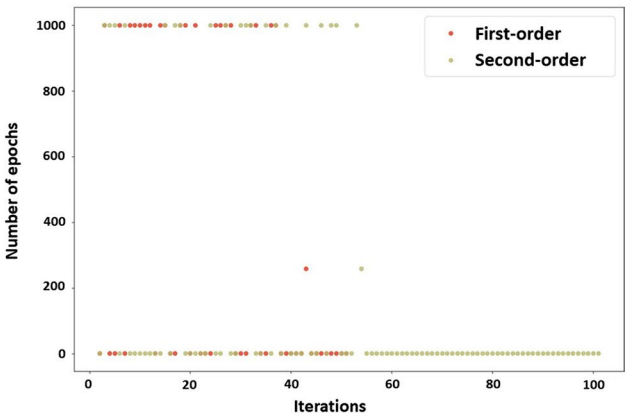
The convergence of $\| \frac{\partial MSE}{\partial W_{network}} \|_{inf}$ is plotted across the number of iterations in Fig. 12a, where it can be observed that both methods converge almost roughly at the same time (around 50 iterations). The only reason that the second-order search takes longer is due to the fact that the tolerance value has not been reached in exact numerical values. Relaxing the value of the tolerance parameter can speed up the process.

In Fig. 12b, the scatter plot of the number of epochs at each iteration against the number of iterations is presented. It can be observed that the values are almost separated into two categories: either reaching the maximum number of epochs allowed in each iteration (1,000), or using only one epoch to reach the tolerance, with the exception of only two points. Some explanations for this behavior are that either the tolerance is difficult to reach, or the layer is already optimized so a polarized distribution of the number of epochs is obtained.

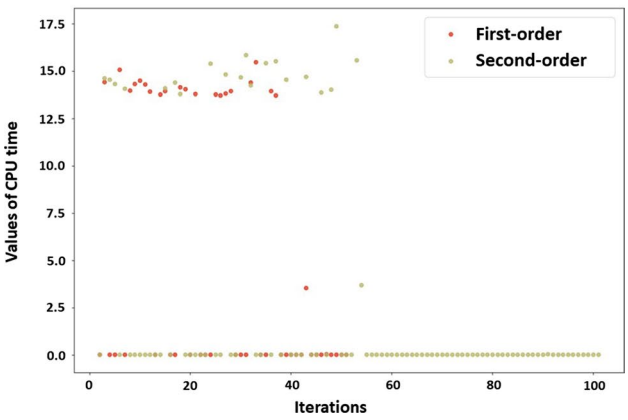
Comparing with Fig. 12c, the distribution of the CPU time is similar to the results presented in Fig. 11b for the number of epochs, indicating a close relationship between these



(a) Gradient across iterations



(b) Epochs across iterations



(c) CPU time across iterations

Fig. 12 Comparison of the convergence rate of optimization using first-order vs second-order information

parameters during each iteration, with fluctuations due to the speed of performing different calculations.

Overall, the second-order information-based approach performs better than the first-order in terms of the total performance, as well as the performance per iteration. It also has the potential to converge in fewer iterations compared to the first-order approach with a different definition of the tolerance.

7 Application to large scale problems and other datasets

The advantage of the hierarchical multi-scale method is not obvious when the network size is small, as demonstrated in the previous sections.

The complete process of automatic differentiation consists of two steps:

1) An NLP optimization to obtain the weights and the neuron outputs, and
2) Iterations to calculate the sensitivities.

For a network of 8 layers, the number of variables that need to be adjusted is equal to the number of layers multiplied with the number of weights (25) and biases (5), in total 30 variables. If the network is trained using the full backpropagation approach, a number of 240 variables is adjusted during each iteration, with the appropriate calculation of the gradients (240 elements). In the case of the multi-scale hierarchical approach, if the optimization is performed layer-by-layer, only 30 variables is adjusted at every iteration, resulting in a much smaller optimization problem.

For a structure of [4, 5, 3, 2, 2], the CPU times required for the calculation of sensitivities of 100 data points are tabulated in Table 6.

The operating system used to run the algorithm is a macOS Big Sur version 11.0.1 (20B29) and the coding language is Python. The processor used to run this code is a 2.3 GHz Quad-Core Intel Core i5 with a memory of 8 GB 2133 MHz LPDDR3.

It can be observed that most of the computation time is spent on the optimization process (97.7% of CPU time). This is an uncontrollable process as a standard optimizer is used in these examples. The calculation of the sensitivities is fast, taking only ~ 0.03 s in total (2.30% of CPU time). Thus, this is quite an effective method as long as the optimizer is sufficiently efficient. A disadvantage of the algorithm is that it

Table 6 The CPU time for the sensitivity calculation of a neural network with the architecture [4, 5, 3, 2, 2] using 100 data points

Steps	Optimization	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Total
CPU Time (s)	1.23300	0.00071	0.01920	0.00787	0.00073	1.26151

Table 7 The comparison between the binary tree search based on first- and second-order information, against backpropagation, for a network with 20 layers

Order	First	Second	Backpropagation
Training set MSE	0.94697	0.94631	0.93835
Testing set MSE	0.97121	0.93880	0.86888
Average epochs	397.26	397.26	200.00
Total iterations	50	50	97
Total CPU time (s)	1,022.4	1,153.5	912.7
Average CPU time (s)	18.681	18.402	9.376

requires the storage of a matrix of size ($NN, NN \times NK$), where NN is the total number of neurons and NK is the total number of data points.

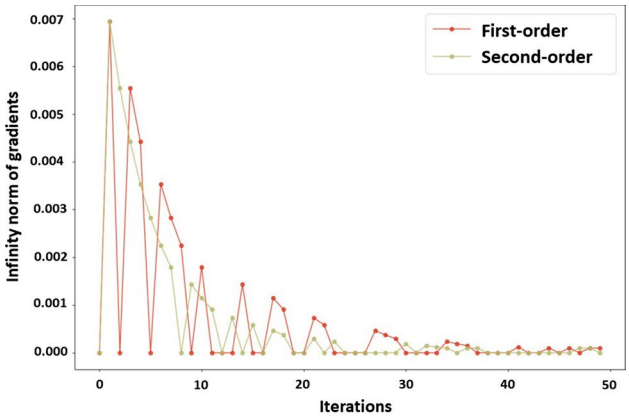
As for the small network used for the illustrative case study, the benefits of utilizing the proposed approach are not clear, the influence of a much larger network will be investigated in this section. Moreover, as this is a simulated case study, the advantages of the proposed framework will be further demonstrated by applying it to other datasets publicly available. For this purpose, the well-known and widely-accepted University of California Irvine (UCI) Machine Learning Repository [79][1] is chosen.

7.1 Large scale problems

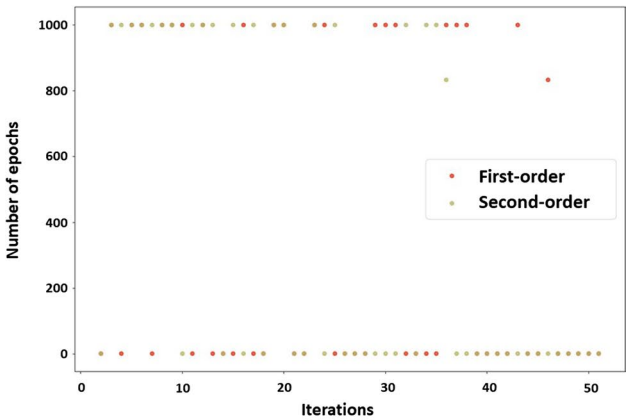
The hierarchical multi-scale search algorithm is implemented to a network with 20 hidden layers of 5 neurons each. The performance of the first-, the second-order and the backpropagation methods is illustrated in Table 7. The average values are obtained at each iteration. The total values are obtained for all iterations.

From these results, it can be observed that the backpropagation is still the fastest algorithm in terms of the total CPU time and CPU time per iteration. The second-order method is slightly faster than first in terms of the average CPU time per iteration. The number of iterations is the same for the first- and second-order methods in this case, both significantly lower compared to the backpropagation method. This demonstrates the effectiveness of the sensitivity-based selection method in optimizing the network to a required tolerance.

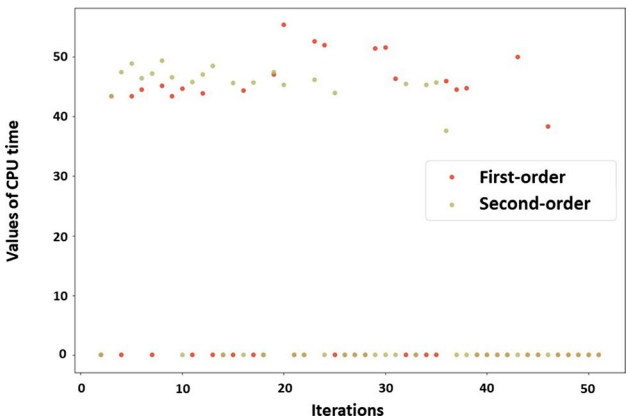
Figure 13 demonstrates the performance of the first- against the second-order method. From Fig. 13a, the rate of convergence is roughly similar for these two methods. Compared to the results from the smaller network (Fig. 12), it can be observed that the training time and the number of epochs are separated into two categories, either costing the maximum to optimize a particular layer, or immediately reaching the convergence criterion with 1 epoch. This is demonstrated in the polarization of data points in Figs. 13b and 13c.



(a) Gradient across iterations



(b) Epochs across iterations



(c) CPU time across iterations

Fig. 13 Comparison of the convergence rate of optimization using first- vs second-order information for a network with 20 layers

[1] The repository can be found online at https://archive.ics.uci.edu/ml/datasets.php

Table 8 Comparison between the binary tree search based on first- and second-order information, against the backpropagation method, for a network with 50 layers

Order	First	Second	Backpropagation
Training set MSE	1.00000	1.00000	1.00001
Testing set MSE	1.00000	1.00000	1.04468
Average epochs	208.83	416.67	200.00
Total iterations	100	50	100
Total CPU Time (s)	3,271.9	3,746.6	2,529.4
Average CPU time (s)	24.519	48.912	25.206

The number of layers is then increased to 50. The results are summarized in Table 8 and Fig. 14. The average values are obtained for each iteration. The total values are obtained for all iterations.

From Table 8, it can be observed that the first-order method is now the fastest in terms of average CPU time, followed by the backpropagation method, demonstrating the advantage of the multi-scale search for large-scale networks. However, the second-order method has a key advantage with the lower number of iterations.

Observing Fig. 14, the second-order method initially seems to be converging more slowly compared to the first-order, later reaching similar speeds of convergence after several iterations. As the speed of convergence differs from that of a 20-layer model, it can be inferred that there is no fixed dominance of the first- over the second-order method and vice versa.

Similarly, it can be concluded that there is a polarization in terms of data points, indicating either immediate convergence or very slow convergence of a particular layer. Based on these results, it is postulated that the slow convergence occurs on layers that contribute with significant changes to the overall model performance, indicating the unequal importance of different layers inside the network.

7.2 Other datasets

A comparison between the end-to-end backpropagation and the hierarchical approach is performed for three UCI datasets:

I. The Combined Cycle Power Plant (CCPP). The input dimension is 4 and the output dimension is 1. The structure of the network is [4, 3, 2, 2, 1]. The total number of data points is 9,568.
II. The Appliances Energy Prediction (AEP). The input dimension is 24 and the output dimension is 2. The structure of the network is [24, 10, 5, 5, 3, 2]. The total number of data points is 10,000.

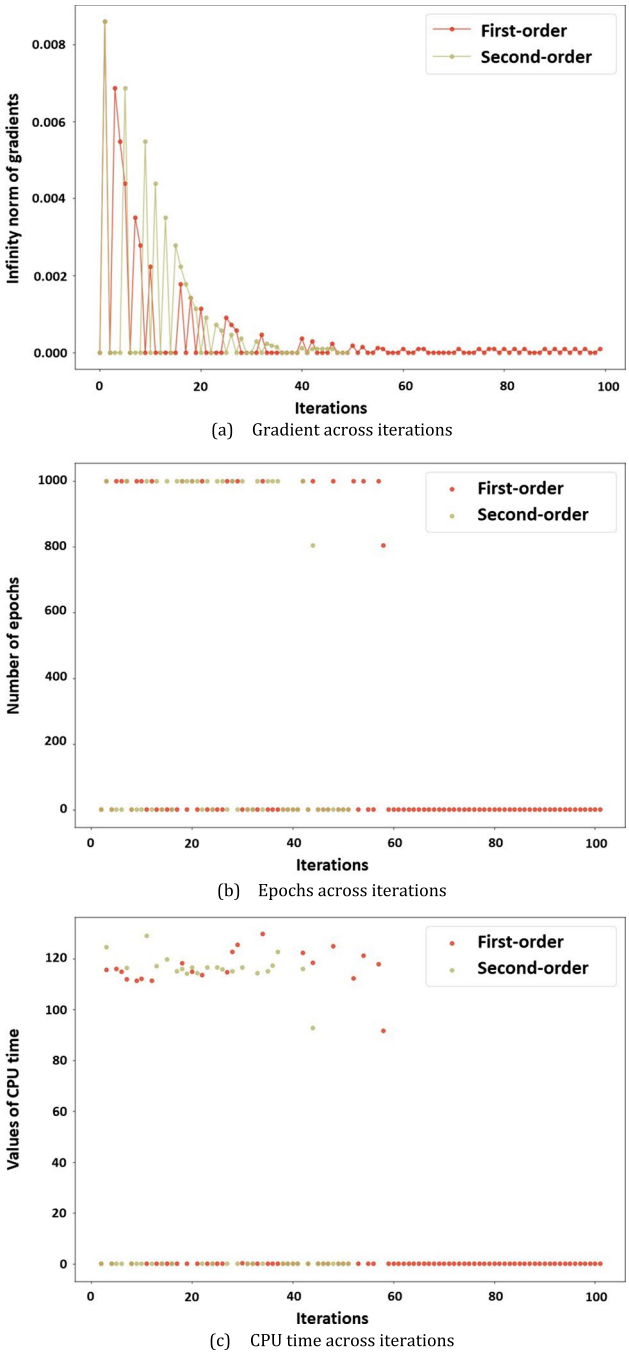


Fig. 14 Comparison of the convergence rate of optimization using first- vs second-order information in a network with 50 layers

III. The Temperature Forecast (TF). The input dimension is 21 and the output dimension is 2. The structure of the network is [21, 10, 5, 5, 3, 2]. The total number of data points is 10,000.

A more detailed description of these datasets is presented in the Supplementary Material.

Table 9 Comparison between the multiscale hierarchical search based on first- and second-order information, against end-to-end backpropagation for the UCI *Dataset I: CCPP*

Parameters and results	First-order	Second-order	End-to-end
Learning rate	0.05	0.05	0.05
Tolerance	0.01	0.01	-
Inner tolerance	0.01	0.01	-
Number of epochs	20	20	-
Maximum number of iterations	50	50	-
Trust region bound, ϵ	0.001	0.001	-
Time of execution (s)	1.06170	2.49700	0.24170
Final MSE	0.22814	0.20021	0.20920

Table 10 Comparison between the multiscale hierarchical search based on first- and second-order information, against end-to-end backpropagation for the UCI *Dataset II: AEP*

Parameters and results	First-order	Second-order	End-to-end
Learning rate	0.05	0.05	0.05
Tolerance	0.005	0.005	-
Inner tolerance	0.005	0.005	-
Number of epochs	30	30	-
Maximum number of iterations	60	60	-
Trust region bound, ϵ	0.001	0.001	-
Time of execution (s)	2.80974	7.15047	0.47615
Final MSE	1.00010	0.99979	1.00006

Table 11 Comparison between the multiscale hierarchical search based on first- and second-order information, against end-to-end backpropagation for the UCI *Dataset III: TF*

Parameters and results	First-order	Second-order	End-to-end
Learning rate	0.05	0.05	0.05
Tolerance	0.005	0.005	-
Inner tolerance	0.005	0.005	-
Number of epochs	30	30	-
Maximum number of iterations	60	60	-
Trust region bound, ϵ	0.001	0.001	-
Time of execution (s)	1.50658	4.92685	0.29743
Final MSE	1.00000	1.00000	2.38036

The results of the comparison and the hyperparameters used are tabulated in Tables 9, 10 and 11. In these tables, the Learning rate determines how fast the gradient is reduced in each step, the Tolerance determines the break-off value compared to the sum of gradients to indicate the termination of the algorithm, while the Inner tolerance is compared to the value of the maximum gradient in the network at every iteration to determine a stop criterion. The Number of epochs is the number of times the optimization is run during each iteration. The Maximum number of iterations determines the total number of iterations the algorithm performs, while ϵ is the infinitesimal difference adopted in the finite difference method. The values are optimized through trial-and-error methods and are unified for a clear comparison.

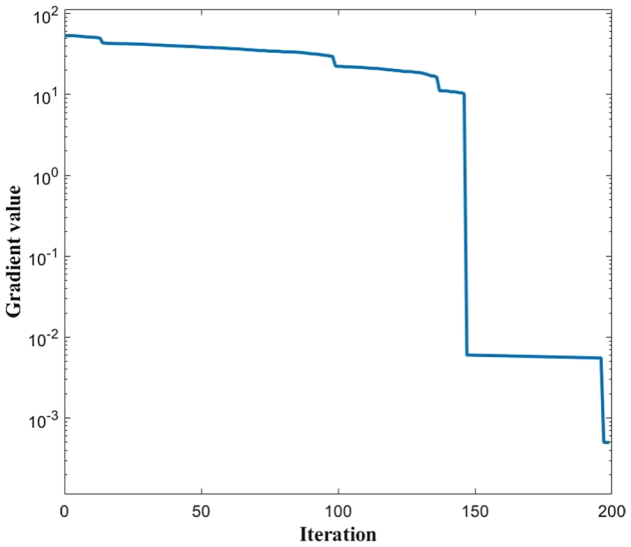


Fig. 15 The evolution of the gradient values for the last layer, for a network with 20 layers

From these results, several characteristics of the proposed method can be extracted.

Firstly, with appropriate hyperparameters, the hierarchical approach is able to optimize to a better solution compared to the end-to-end backpropagation method. Although the computation is not faster, in particular applications where a trained model must be deployed, the speed is not always the key. For example, in an industrial setting, the better solution will be always pursued, as it will often have significant impact on the operation costs.

Secondly, in case of the hierarchical approach, there is also a trade-off of accuracy with respect to the solution time. When the number of epochs and the maximum number of iterations is set high, more accurate results are obtained. This is a significant difference compared to the end-to-end backpropagation, where only the learning rate indirectly determines both the speed and the accuracy of the results.

The framework, relying on sensitivity values of individual neurons relative to the overall output, is simple to implement, efficient to run and successful in producing optimized neural networks. In this case, the key advantage of the proposed approach is that it is less affected by the vanishing gradient problem, which occurs when optimizing very deep neural networks using gradient-based methods, specifically

during backpropagation [80–85]. In the case of the traditional methods, the gradients, when populated throughout the layers, become so diminished that it affects the accuracy of the optimization results.

Figure 15 presents the gradient values calculated over 200 iterations of the DNN with 20 layers. These results show that the gradient decreases very slowly over the first 140 iterations, while the solution is produced within 50 iterations, as illustrated in Table 7.

Thus, by not involving populating the gradients during the training procedure, the hierarchical approach has great potential in achieving improved solutions at lower computational effort, by focusing the modification of the parameters on the most sensitive parts of the DNN during the optimization phase, as illustrated in the case studies investigated above. With further development the proposed optimization framework can create a new paradigm shift in how professionals optimize neural networks.

8 Conclusions and outlook

In this paper, a novel hierarchical multi-scale search algorithm is introduced for the tuning of DNNs. The approach utilizes a leveled method where, within each level, the network is divided into left and right sides, and the sensitivity of each side branch is evaluated using the first- or second-order finite differences method. The algorithm then progressively selects one side to further divide the network until only one layer remains, resulting in a binary tree search. The selected layer is optimized in one iteration until a convergence criterion is achieved. The selection process repeats until an overall tolerance is reached, signaling the end of the network tuning. The proposed algorithm offers a key advantage in optimizing large DNNs with numerous layers, due to its search efficiency of $O(\log N)$.

The key innovation of the algorithm is its implementation of a binary search process that optimizes a single layer at a time. The introduction of selective tuning enables rapid convergence of the DNNs and efficient equilibration of the sensitivity values, particularly in the context of large networks, where only critical layers are optimized. If the tolerance requirements are not stringent, the method potentially achieve faster convergence than the optimization of all layers. As the sensitivity values of the scaling factor also correspond to neuron importance, there is potential for future research to explore how this method can be manipulated for the structural evolution of DNNs.

The algorithm also incorporates a crucial element of randomization through the binary selector, where a sensitivity-based probability value is computed and utilized as the likelihood to choose between left and right. This random factor contributes to the improved performance of the algorithm by reducing repetition during the optimization process of a single layer, ultimately resulting in a more efficient optimization for large-scale networks. The results of the case studies conducted to demonstrate the application of the procedure show that the hierarchical multi-scale search algorithm can generate solutions that are comparable or even better to those produces by conventional approaches such as end-to-end backpropagation.

Based on the results presented in Section 5, the newly proposed sensitivity metric facilitates effective analysis, evaluation, design, and, control of very large-to-enormous scale systems, both mathematically and abstractly. This metric can enhance the high-impact topic emphasized in Section 5 through the following aspects:

1. The first-order structural sensitivities provide a linear approximation of the investigated system from a hierarchical multi-scale perspective.
2. The second-order structural sensitivities quantitatively measure the degree of local non-linearity of the system, considering its interacting parts from the top level of abstraction down to its finest modelling scale.
3. The mixed second-order sensitivities reveal the degree of coupling within the underlying level sub-compartments of the model, as the procedure moves down the levels of the hierarchical multi-scale structure/framework.

This numerically quantitative system metric allows for:

1. The exploration of new self-adaptive local information, simulation and system optimization algorithms within the currently proposed framework.
2. A self-adaptive parallelization of nested computations that can reliably assign loosely interacting (or loosely connected) components to different parallel processors. As a result, new parallel algorithms can be implemented on existing hardware, and new dedicated computational architectures can be designed to be exploited fully by the novel computational modelling and solution framework.

To ensure that the proposed framework can truly bring about a real paradigm shift in the use of neural networks for industrial (large scale) applications, additional efforts are necessary to improve the implementation and optimize the code.

8.1 Structural sensitivity equilibration

One important feature of the novel hierarchical multi-scale optimization algorithm is its ability to equilibrate the structural sensitivities and converge to at least a local minimum. The algorithm employs a randomized

Table 12 Table of notations

θ	Scaling factor
ϕ	Objective function minimiser
L	Lagrangian function
W_{ij}	Connection weights between neuron i and neuron j
z	Output of a neuron after activation (multiplied by the scaling factor)
y	Input to a neuron, also the output from the previous layer
h	Difference between the supposed output of a neuron and product of activated neuron output and scaling factor h is bounded to be zero
λ	The set of Lagrange multipliers associated with the constraints $h(\cdot;\cdot)$
f	Activation function applied to the input to a neuron
S_{left}, S_{right}	Absolute values of the structural sensitivities at the left and right side of the branching point, respectively
ϵ	Trust region bound
δ	A small disturbance in the value of sensitivity
q	Quadratic model as a selector at the branching point
a, b, c, d, e, f	Quadratic model coefficients
r	Value of the randomized selector to determine the side of a branch to go down
w_{rj}	Weights between input neuron r and the hidden neuron j (Table 1)
v_{jk}	Connection weight between hidden neuron j and output neuron k (Table 1)
Q_{ik}	Percentage of influence of the input variable x_i and output y_k (Table 1)
WP_{ik}	Influence of the input variable x_i on the output y_k (Table 1)
$f'(net_j)$	Derivative of the activation function of the hidden neuron j (Table 1)
$f'(net_k)$	Derivative of the activation function of the output neuron k (Table 1)
R_d	Layer-wise relevance score (Section 2.2)

left-or-right non-deterministic selector criterion at the binary multi-scale partitioning tree, which is statistically designed to favor the side with the largest locally determined absolute structural sensitivity (derivative) value. This results in iterations that are expected to achieve a path not only to a local minimum, but also to exhibit equilibration of left and right structural sensitivity values at the branching points of the binary partitioning tree.

The use of compartments with arbitrary partitioning generates equilibrated values in terms of importance on the overall defined performance index criterion of the entire system being modelled or even controlled online in real-time by the novel hierarchical multi-scale optimization framework. This property enables further novel considerations, as a system that is equilibrated in the structural sensitivity sense can easily detect any small deviation at the top-level of the binary tree and identify it at the finest structure of the embedded model and underlying physical system in $\log_2$ steps of the total number of the finest structure components of the system.

The proposed framework challenges the current status quo through rounds of partial training and facilitates innovation in the most fundamental steps of the DNNs development, leading to the development of efficient automatic compression and acceleration techniques.

This work raises interesting questions related to the meaning and potential physical interpretation beyond from the mathematical definition, as well as the physical interpretation of the system being modelled within the novel hierarchical multi-scale modelling framework proposed.

To answer these questions, it is essential to apply the model to industrial scenarios and examine the implications in a real-world setting.

8.2 Inverse problem: Physical system laws analysis and discovery

With an appropriately refined mathematical model of a system, both parametric and structural analysis can be carried out starting from the chosen performance index law and investigating the partitioning obtained at that level. The structural sensitivity balance for left-and-right partition can be achieved through recursive application of the model all the way to the tuning of fixed parameters, leading to the development of a highly equilibrated hierarchical multi-scale mathematical or other abstract model that can describe and predict the behavior of the underlying physical or abstract system.

Moreover, the proposed novel algorithmic framework allows for transparent interpretation of all its interactions and

machinery, which facilitates the identification of modelling flaws and exploration of alternative propositions to correct or complement any existing model in a minimal number of steps. Additionally, it enables the correct identification and safe utilization of degrees of freedom that ensure the satisfaction of multiple desirable criteria in real-world applications such as safety, profitability stability project longevity and continuity.

The proposed hierarchical multi-scale modelling framework can be applied to explore various topics such as alternative mathematical game theory models and imaginative solutions methods, as well as solving currently difficult multi-objective or other multilevel mathematical optimization formulations.

The next step in the development of the framework involves using the sensitivities introduced in this work to adapt the structure and size of DNNs in an unsupervised way, leading to the deployment of highly flexible and plastic neural networks within the field of AI.

9 Table of notations

The notations are presented in Table 12.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10489-023-04745-8.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors declare that they have no conflict of interest with respect to the research, authorship and/or publication of this article.

References

1. AbdElaziz M, Dahou A, Abualigah L, Yu L, Alshinwan M, Khasawneh AM, Lu S (2021) Advanced metaheuristic optimization techniques in applications of deep neural netowrks: a review. Neural Comput Appl 33:14079–14099
2. Shrestha A, Mahmood A (2019) Review of Deep Learning algorithms and architectures. IEEE Access 7:53040–53065
3. Bhuvaneswari V, Priyadharshini M, Deepa C, Balaji D, Rajeshkumar L, Ramesh M (2021) Deep learning for material synthesis and manufacturing systems: A review. Material Today Proc 46(part 9):3263–3269
4. Kapusuzoglu B, Mahadevan S (2020) Physics-informed and hybrid machine learning in additive manufacturing: Application to fused filament fabrication. JOM 72:4695–4705
5. Gavrishchaka V, Senyukova O, Koepke M (2019) Synergy of physics-based reasoning and machine learning in biomedical applications: towards unlimited deep learning with limited data. Adv Physics X 4(1):1582361
6. Jiao Z, Hu P, Xu H, Wang Q (2020) Machine learning and deep learning in chemical health and safety: A systematic review of techniques and applications. ACS Chem Health Saf 27(6):316–334
7. Li J, Zhu X, Li Y, Tong YW, Ok YS, Wang X (2021) Multi-task prediction and optimization of hydrochar properties from high-moisture municipal solid-waste: Application of machine learning on waste-to-resource. J Clean Prod 278:123928
8. Wang S, Ren P, Takyi-Aninakwa P, Jin S, Fernandez C (2022) A critical review of improved deep convolutional neural network for multi-timescale state prediction of Lithium-ion batteries. Energies 15(14):5053
9. Wang S, Takyi-Aninakwa P, Jin S, Yu C, Fernandez C, Stroe DI (2022) An improved feedforward-long short-term memory modelling method for the whole-life-cycle state of charge prediction of lithium-ion batteries considering current-voltage-temperature variation. Energy 254(part A):124224
10. Chen ZX, Iavarone S, Ghiasi G, Kannan V, D'Alessio G, Parente A, Swaminathan N (2021) Application of machine learning for filtered density function closure in MILD combustion. Combust Flame 225:160–179
11. Ruan H, Dorneanu B, Arellano-Garcia H, Xiao P, Zhang L (2022) Deep learning-based fault prediction in wireless sensor network embedded cyber-physical system for industrial processes. IEEE Access 10:10867–10879
12. Mishra R, Gupta H (2023) Transforming large-size to lightweight deep neural networks for IoT applications. ACM Comput Surv 55(11):1–35
13. Groumpos PP (2016) Deep learning vs. wise learning: A critical and challenging overview. IFAC-PapersOnLine 49(29):180–189
14. Vasdevan S (2020) Mutual information based learning rate decay for stochastic gradient descent training of deep neural networks. Entropy 22(5):560
15. Cheridito P, Jentzen A, Rossmannek F (2021) Non-convergence of stochastic gradient descent in the training of deep neural networks. J Complex 64:101540
16. Le-Duc T, Nguyen QH, Lee J, Nguyen-Xuan H (2022) Strengthening gradient descent by sequential motion optimization for deep neural networks. IEEE Trans Evol Comput 27(3):565–579
17. Asher N (2021) Review on gradient descent algorithms in deep learning approaches. J Innov Dev Pharm Tech Sci 4(3):91–95
18. Alarfaj FK, Khan NA, Sulaiman M, Alomair AM (2022) Application of a machine learning algorithm for evaluation of stiff fractional modelling of polytropic gas spheres and electric circuits. Symmetry 14(12):2482
19. Christou V, Arjmand A, Dimopoulos D, Varvarousis D, Tsoulos I, Tzallas AT, Gogos C, Tsipouras MG, Glavas E, Ploumis A, Giannakeas N (2022) Automatic hemiplegia type detection (right or left) using the Levenberg-Marquardt backpropagation method. Information 13(2):101
20. Choudhary P, Singhai J, Yadav JS (2022) Skin lesion detection based on deep neural networks. Chemom Intell Lab Syst 230:104659
21. Al-Shargabi AA, Almhafdy A, Ibrahim DM, Alghieth M, Chiclana F (2021) Tuning deep neural networks for predicting

energy consumption in arid climate based on building characteristics. Sustainability 13(22):12442
22. Choudhary T, Mishra V, Goswami A, Sarangapani J (2020) A comprehensive survey on model compression and acceleration. Artif Intell Rev 53:5113–5155
23. Zhang Z, Kouzani AZ (2020) Implementation of DNNs on IoT devices. Neural Comput Appl 32:1327–1356
24. Mittal S (2020) A survey on modelling and improving reliability of DNN algorithms and accelerators. J Syst Architect 104:101689
25 Dhouibi M, Ben Salem AK, Saidi A, Saoud SB (2021) Accelerating deep neural networks: A survey. IET Comput Digit Tech 15(2):79–96
26. Armeniakos G, Zervakis G, Soudris D, Henkel J (2022) Hardware approximate techniques for deep neural network accelerators: A survey. ACM Comput Surv 55(4):1–36
27. Liu D, Kong H, Luo X, Liu W, Subramaniam R (2022) Bringing AI to edge: From deep learning's perspective. Neurocomputing 485:297–320
28. Hussain H, Tamizharasan PS, Rahul CS (2022) Design possibilities and challenges of DNN models: a review on the perspective end devices. Artif Intell Rev 55:5109–5167
29. Zhang Y, Tiňo P, Leonardis A, Tang K (2021) A survey on neural network interpretability. IEEE Trans Emerg Topics Computat Intell 5(5):726–741
30. Montavon G, Samek W, Müller K-R (2018) Methods for interpreting and understanding deep neural networks. Digit Signal Process 73:1–15
31. Gevrey M, Dimopoulos I, Lek S (2003) Review and comparison of methods to study the contribution of variables in artificial neural network models. Ecol Model 160(3):249–264
32. Montaño J, Palmer A (2003) Numeric sensitivity analysis applied to feedforward neural networks. Neural Comput Appl 12(2):119–125
33. Lek S, Delacoste M, Baran P, Dimopoulos I, Lauga J, Aulagnier S (1996) Application of neural networks to modelling nonlinear relationships in ecology. Ecol Model 90(1):39–52
34. Fawzi A, Moosavi-Dezfooli SM, Frossard P (2017) The robustness of deep networks: A geometrical perspective. IEEE Signal Process Mag 34(6):50–62
35. Shu H, Zhu H (2019) Sensitivity analysis of deep neural networks, in Proceedings of the AAAI Conference on Artificial Intelligence 33: 4943–4950
36. Mrzygłód B, Hawryluk M, Janik M, Olejarczyk-Wożeńska I (2020) Sensitivity analysis of the artificial neural networks in a system for durability prediction of forging tools to forgings made of C45 steel. Int J Adv Manuf Technol 109:1385–1395
37. Zhang S (2021) Design of deep neural networks formulated as optimisation problems," Doctoral thesis, University of Cambridge. https://doi.org/10.17863/CAM.82337
38. Tchaban T, Taylor M, Griffin J (1998) Establishing impacts of the inputs in a feedforward neural network. Neural Comput Appl 7(4):309–317
39. Garson DG (1991) Interpreting neural network connection weights. AI EXPERT 6(4): 47–51
40. Oparaji U, Sheu R-J, Bankhead M, Austin J, Patelli E (2017) Robust artificial neural network for reliability and sensitivity analyses of complex non-linear systems. Neural Netw 96:80–90
41. May Tzuc O, Bassam A, Ricalde LJ, Cruz May E (2019) Sensitivity analysis with artificial neural networks for operation of photovoltaic systems. Artif Neural Netw Eng Appl 10:127–138
42. Zhang X, Xie Q, Song M (2021) Measuring the impact of novelty, bibliometric, and academic-network factors on citation count using a neural network. J Inform 15(2):101140
43. Xie Q, Wang J, Kim G, Lee S, Song M (2021) A sensitivity analysis of factors influential to the popularity of shared data in repositories. J Inform 15(3):101142
44. Mazidi MH, Eshghi M, Raoufy MR (2022) Premature ventricular contraction (PVC) detection system based on tunable Q-factor wavelet transform. J Biomed Phys Eng 12(1):61–74
45. Liu X, Qiao S, Han G, Hang J, Ma Y (2022) Highly sensitive HF detection based on absorption enhanced light-induced thermoelastic spectroscopy with a quartz tuning fork of receive and shallow neural network fitting. Photoacustics 28:100422
46. Ivanovs M, Kadikis R, Ozols K (2021) Perturbation-based methods for explaining deep neural networks: A survey. Pattern Recogn Lett 150:228–234
47. Teodoro G, Kurç TM, Taveira LFR, Melo ACMA, Gao Y, Kong J, Saltz JH (2017) Algorithm sensitivity analysis and parameter tuning for tissue image segmentation pipelines. Bioinformatics 33(7):1064–1072
48. Akenbrand MJ, Shainberg L, Hock M, Lohr D, Schreiber LM (2021) Sensitivity analysis for interpretation of machine learning based segmentation models in cardiac MRI. BMC Med Imaging 21:27
49. Jeczmionek E, Kowalski PA (2022) Input reduction of convolutional neural networks with global sensitivity analysis as a data-centric approach. Neurocomputing 506:196–205
50. Kim MK, Cha J, Lee E, Pham VH, Lee S, Theera-Umpon N (2019) Simplified neural network model design with sensitivity analysis and electricity consumption prediction in a commercial building. Energies 12(7):1201
51. Kowalski PA, Kusy M (2018) Determining significance of input neurons for probabilistic neural network by sensitivity analysis procedure. Comput Intell 34(3):895–916
52. Samek W, Binder A, Montavon G, Lapuschkin S, Müller K-R (2016) Evaluating the visualization of what a deep neural network has learned. IEEE Trans Neural Netw Learn Syst 28(11):2660–2673
53. Buhrmester V, Münch D, Arens M (2021) Analysis of explainers of black box deep neural networks for computer vision: A survey. Mach Learn Knowl Extraction 3(4):966–989
54. Meister S, Wermes M, Stüve J, Groves RM (2021) Cross-evaluation of a parallel operating SVM-CNN classifier for reliable internal decision-making processes in composite inspection. J Manuf Syst 60:620–639
55. Li Z, Li H, Meng L (2023) Model compression for deep neural networks: A survey. Computers 12(3):60
56. Shin E, Park J, Yu J, Patra C (2018) Prediction of grouting efficiency by injection of cement milk into sandy soil using an artificial neural network. Soil Mech Found Eng 55(5):305–311
57. Mozumder RA, Laskar AI, Hussain M (2018) Penetrability prediction of microfine cement grout in granular soil using artificial intelligence techniques. Tunn Undergr Space Technol 72:131–144
58. Chaurasia RC, Sahu D, Suresh N (2021) Prediction of ash content and yield percent of clean coal in multi gravity separator using artificial neural networks. Int J Coal Prep Util 41(5):362–369
59. Bach S, Binder A, Montavon G, Klauschen F, Müller K-R, Samek W (2015) On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. PLoS ONE 10(7):e0130140
60. Böhle M, Eitel F, Weygandt M, Ritter K (2019) Layer-wise relevance propagation for explaining deep neural network decisions in MRI-based Alzheimer's disease classification. Front Aging Neurosci 11:194
61. Grezmak J, Zhang J, Wang P, Loparo KA, Gao RX (2019) Interpretable convolutional neural network through layer-wise relevance propagation for machine fault diagnosis. IEEE Sens J 20(6):3172–3181
62. Montavon G, Binder A, Lapuschkin S, Samek W, Müller K-R (2019) "Layer-wise relevance propagation: An overview," Explainable AI: interpreting, explaining and visualizing deep learning, SpringerCham, pp. 193–209

63. Yeom SK, Seegerer P, Lapushkin S, Binder A, Wiedemann S, Müller KR, Samek W (2021) Pruning by explaining: A novel criterion for deep neural network pruning. Pattern Recogn 115:107899
64. Baydin AG, Pearlmutter BA, Radul AA, Siskind JM (2018) Automatic differentiation in machine learning: A survey. J Mach Learn Res 18:1–43
65. Margossian CC (2019) A review of automatic differentiation and its efficient implementation. Wiley Interdiscip Rev Data Min Knowl Disc 9(4):e1305
66. Cai S, Wang Z, Wang S, Perdikaris P, Karniadakis GE (2021) Physics-informed neural networks for heat transfer problems. ASME J Heat Transf 143(6):060801
67. Conejeros R, Vassiliadis VS (2000) Dynamic biochemical reaction process analysis and pathway modification predictions. Biotechnol Bioeng 68(3):285–297
68. Haghighat E, Raissi M, Moure A, Gomez H, Juanes R (2021) A physics-informed deep learning framework for inversion and surrogate modelling in solid mechanics. Comput Methods Appl Mech Eng 379:113741
69. Abdolrasol MGM, Hussain SMS, Ustun TS, Sarker MR, Hannan MA, Mohamed R, Abd Ali J, Mekhilef S, Milad A (2021) Artificial neural networks based optimization techniques: A review. Electronics 10(21):2689
70. Aszemi NM, Dominic PDD (2019) Hyperparameter optimization in convolutional neural network using genetic algorithms. Int J Adv Comput Sci Appl 10(6):269–278
71. Lillicrap TP, Santoro A, Marris L, Ackerman CJ, Hinton G (2020) Backpropagation and the brain. Nat Rev Neurosci 21:335–346
72. Sutton RS, Barto AG (2018) Reinforcement learning: An introduction. MIT Press
73. Hariharan N, Paavai PA (2022) A brief study of deep reinforcement learning with epsilon-greedy exploration. Int J Comput Digit Syst 11(1):541–551
74. Yang T, Zhang S, Li C (2021) A multi-objective hyper-heuristic algorithm based on adaptive epsilon-greedy selection. Complex & Intelligent Systems 7:765–780
75. Gong M, Liu J, Qin AK, Zhao K, Tan KC (2021) Evolving deep neural networks via cooperative coevolution with backpropagation. IEEE Trans Neural Netw Learn Syst 32(1):420–434
76. Gambella C, Ghaddar B, Naoum-Sawaya J (2021) Optimization problems for machine learning: A survey. Eur J Oper Res 290(3):807–828
77. Wright LG, Onodera T, Stein MM, Wang T, Schachter DT, Hu Z, McMahon PL (2022) Deep physical neural networks trained with backpropagation. Nature 601:549–555
78. Zaras A, Passalis N, Tefas A (2022) Neural networks and backpropagation. Deep Learning for Robot Perception and Cognition 2:17–34
79. Dua D, Graff C (2019) UCI machine learning repository, Irvine, CA: University of California, School of Information and Computer Science. [Online]. Available: http://archive.ics.uci.edu/ml. Accessed Dec 2022
80. Lillicrap TP, Santoro A (2019) Backpropagation through time and the brain. Curr Opin Neurobiol 55:82–89
81. Basodi HZS, Ji C, Pan Y (2020) Gradient amplification: An efficient way to train deep neural networks. Big Data Min Analytics 3:196–207
82. Scardapane S, Scarpinti M, Baccarelli E, Uncini A (2020) Why should we add early exits to neural networks? Cogn Comput 12:954–966
83. Van Houdt G, Mosquera C, Nápoles G (2020) A review on the long short-term memory model. Artif Intell Rev 53:5929–5955
84. Mishra RK, Sandesh Reddy GY, Pathak H (2021) The understanding of deep learning: A comprehensive review. Math Probl Eng 2021:5548884
85. Alzubaidi L, Zhang J, Humaidi AJ, Al-Dujaili A, Duan Y, Al-Shamma O, Santamaría J, Fadhel MA, Al-Amidie M, Fahran L (2021) Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. J Big Data 8:53

Sushen Zhang is a Chemical Engineering graduate, at the MEng level, from the University of Cambridge. Following her graduation she pursued successfully a PhD degree in Chemical Engineering at the University of Cambridge, working in the Process Systems Engineering research group at the Department of Chemical Engineering and Biotechnology. Her PhD research focused on novel formulations and modeling of systems via Artificial Neural Networks and Artificial Intelligence more broadly. Following completion of her studies, she has returned to China where she presently pursues an independent career in the commercial sector.

Vassilios S. Vassiliadis received the Diploma degree in Chemical Engineering (Masters of Engineering) from the School of Chemical Engineering, National Technical University of Athens, Athens, Greece, in 1989 and the Ph.D. degree in Process Systems Engineering from the Department of Chemical Engineering and Chemical Technology, Imperial College London, London, U.K., in 1993. At Imperial College, he was supervised for his Ph.D. studies by Professor Roger W. H. Sargent, the founder of the PSE research area internationally, and by Professor Costas C. Pantelides, a leading figure in the area of Dynamic Simulation. He then spent a year working as a Postdoctoral Associate with the Department of Chemical Engineering, Princeton University, Princeton, NJ, USA. He is a retired Senior Lecturer with the University of Cambridge, Cambridge, U.K. His research interests lie in the development and application of optimization and simulation tools in engineering and scientific domains.

Bogdan Dorneanu studied Chemical Engineering at the Politehnica University of Bucharest in Romania and received his doctorate from the Technical University of Delft, The Netherlands. He is currently a Postdoctoral Researcher with the Department of Process and Plant Technology, Brandenburg University of Technology (BTU), leading the Digitalization Group. His research interests include process and product modeling and simulation, machine learning, model predictive control, and optimization and design of systems and processes, including models and decision-making tools for distributed energy resource networks.

Harvey Arellano-Garcia studied Energy and Process Engineering at the TU Berlin in Germany. He then did his doctorate at the same University in the field of Process Systems Engineering with a focus on Process Intensification and Optimization. Prof. Dr.-Ing. Arellano-Garcia founded and headed the "Process and Energy Systems Engineering" research group at the TU Berlin from 2007 to 2012. During this time he carried out research stays at Imperial College London and MIT in Boston. His work has been awarded various prizes, including the EFCE (European Federation of Chemical Engineering) Excellence Award in recognition of his outstanding PhD in CAPE (Computer-Aided Process Engineering). Between 2012 and 2019 he was appointed University Professor in the UK and lately Research Director and Professor of Chemical Engineering at the University of Surrey. Prof. Dr.-Ing. Arellano-Garcia head now the Department of Process and Plant Engineering at the Brandenburg University of Technology in Germany.