Topological derivative approach for deep neural network architecture adaptation

C G Krishnanunni*, Tan Bui-Thanh†, and Clint Dawson†

Abstract. This work presents a novel algorithm for progressively adapting neural network architecture along the depth. In particular, we attempt to address the following questions in a mathematically principled way: i) Where to add a new capacity (layer) during the training process? ii) How to initialize the new capacity? At the heart of our approach are two key ingredients: i) the introduction of a "shape functional" to be minimized, which depends on neural network topology, and ii) the introduction of a topological derivative of the shape functional with respect to the neural network topology. Using an optimal control viewpoint, we show that the network topological derivative exists under certain conditions, and its closed-form expression is derived. In particular, we explore, for the first time, the connection between the topological derivative from a topology optimization framework with the Hamiltonian from optimal control theory. Further, we show that the optimality condition for the shape functional leads to an eigenvalue problem for deep neural architecture adaptation. Our approach thus determines the most sensitive location along the depth where a new layer needs to be inserted during the training phase and the associated parametric initialization for the newly added layer. We also demonstrate that our layer insertion strategy can be derived from an optimal transport viewpoint as a solution to maximizing a topological derivative in p-Wasserstein space, where $p \geq 1$. Numerical investigations with fully connected network, convolutional neural network, and vision transformer on various regression and classification problems demonstrate that our proposed approach can outperform an ad-hoc baseline network and other architecture adaptation strategies. Further, we also demonstrate other applications of topological derivative in fields such as transfer learning.

Key words. Neural architecture adaptation, Topological derivative, Optimal control theory, Hamiltonian, Optimal transport theory.

AMS subject classifications. 68T07, 68T05

1. Introduction.

It has been observed that deep neural networks (DNNs) create increasingly simpler but more useful representations of the learning problem layer by layer [23, 42, 41, 67]. Furthermore, empirical evidence supports the paradigm that depth of a network is of crucial importance [49, 51, 56, 32]. Some of the problems associated with training such deep networks include: i) a possible large training set is needed to overcome the over-fitting issue; ii) the architecture adaptability problem, e.g., any amendments to a pre-trained DNN, requires retraining even with transfer learning; iii) GPU employment is almost mandatory due to massive network and data sizes. Most importantly, it is often unclear on the number of layers and number of neurons to be used in each layer while training a neural network for a specific task. Therefore, there is a critical need for rigorous adaptive principles to guide the architecture design of a neural network.

1.1. Related work.

Neural architecture adaptation algorithms can be broadly classified into two categories: i) neural architecture search algorithms and ii) principled adaptive strate-

*Dept of Aerospace Engineering & Engineering Mechanics, UT Austin (krishnanunni@utexas.edu).

†Dept of Aerospace Engineering & Engineering Mechanics, Oden Institute for Computational Engineering & Sciences, UT Austin (tanbui@utexas.edu, clint.dawson@austin.utexas.edu).

gies. Neural architecture search (NAS) algorithms rely on metaheuristic optimization, reinforcement learning strategy, or Bayesian hyperparameter optimization to arrive at a reasonable architecture [68, 53, 55, 17, 48, 4, 40, 38, 35, 26]. However, these strategies involve training and evaluating many candidate architectures (possibly deep) in the process and are thus computationally expensive. One key issue with NAS algorithms is that most of the methods report the performance of the best-found architecture, presumably resulting from a single run of the search process [35]. However, random initializations of each candidate architecture could drastically influence the choice of best architecture prompting the use of multiple runs to select the best architecture, and hence prohibitively expensive. On the other hand, sensible adaptive strategies are algorithms for growing neural networks where one starts by training a small network and progressively increasing the size of the network (width/depth) [63, 64, 62, 19, 32, 12].

In particular, existing works have considered growing the width gradually by adding neurons for a fixed depth neural network [63, 64, 39, 19]. Liu et al. [63] developed a simple criterion for deciding the best subset of neurons to split and a splitting gradient for optimally updating the off-springs. Wynne-Jones [64] considered splitting the neurons (adding neurons) based on a principal component analysis on the oscillating weight vector. Chen et al. [12] showed that replacing a network with an equivalent network that is wider (has more neurons in each hidden layer) allows the equivalent network to inherit the knowledge from the existing one and can be trained to further improve the performance. Firefly algorithm by Wu et al. [62] generates candidate neurons that either split existing neurons with noise or are completely new and selects those with the highest gradient norm.

On the other hand, many efforts have been proposed for growing neural architecture along the depth [22, 33, 60]. Layerwise training of neural networks is an approach that addresses the issue of the choice of depth of a neural network and the computational complexity involved with training [65]. Hettinger et al. [22] showed that layers can be trained one at a time and the resulting DNN can generalize better. Bengio et al. [6] proposed a greedy layerwise unsupervised learning algorithm where the initial layers of a network are supposed to represent more abstract concepts that explain the input observation, whereas subsequent layers extract low-level features. Recently, we devised a manifold regularized greedy layerwise training approach for adapting a neural architecture along its depth [32]. Net2Net algorithm [12] introduced the concept of function-preserving transformations for rapidly transferring the information stored in one neural network into another neural network. Wei et al. [60] proposed the AutoGrow algorithm to automate depth discovery in deep neural networks where new layers are inserted progressively if the accuracy improves; otherwise, stops growing and thus discovers the depth.

It is important to note that any algorithm for growing neural networks (width and/or depth) should adequately address the following questions in a mathematically principled way rather than a heuristic approach [19]: **When** to add new capacity (neurons/layers)?; **Where** to add new capacity?; **How** to initialize the new capacity? Most existing efforts to address these questions focus on growing networks in width, i.e., by adding neurons [63, 62, 64, 19, 39]. When it comes to growing neural architectures in depth, relatively little work adequately addresses these questions. Sensli [31] extends the Net2Net framework [12] by addressing not only how to initialize a new layer, but also where to insert it within the network. However, in

Sensli [31], the initialization of a new layer is independent of both the data and the location at which the new layer is inserted in the network. We hypothesize that data-dependent, location-dependent initialization of the added layer is a crucial component of such adaptation strategies, contributing to improved generalization rather than focusing solely on where the new layer is inserted.

1.2. Our contributions. In this work, we derive an algorithm for progressively increasing a neural network's depth inspired by topology optimization. In particular, we provide solutions to the following questions: i) Where to add a new layer; ii) When to add a new layer; and iii) How to initialize the new layer? We will show that our approach leads to a data-dependent, position dependent initialization of an added layer. We shall present two versions of the algorithm: i) a semi-automated version where a predefined scheduler is used to decide when a new layer needs to be added during the training phase [19]; ii) a fully automated growth process in which a validation metric is employed to automatically detect when a new layer needs to be added. Our method does not have the limitation that the loss function needs to plateau before growing such as those in [63, 28]. Our algorithm is based on deriving a closed-form expression for the topological derivative for a shape functional with respect to a neural network (Theorem 2.7). To that end, in Definition 2.2 and Proposition 2.3 we introduce the concept of admissible perturbation and suggest ways for constructing an admissible perturbation. In subsection 2.4, we show that the first-order optimality condition leads to an eigenvalue problem whose solution determines where a new layer needs to be inserted along with the associated parametric initialization for the added layer. The efficiency of our proposed algorithm is explored in section 5 by carrying out extensive numerical investigations with different architectures on prototype regression and classification problems.

2. Mathematical framework. In this work, all the matrices and vectors are represented in boldface. Consider a regression/classification task where one is provided with S training data points, input data dimension n_0, and label dimension n_T. Let the inputs $\mathbf{x}_i \in \mathbb{R}^{n_0}$ for $i \in \{1, 2, ...S\}$ be organized row-wise into a matrix $\mathbf{X} \in \mathbb{R}^{S\times n_0}$ and let the corresponding true labels be denoted as $\mathbf{c}_i \in \mathbb{R}^{n_T}$ and stacked row-wise as $\mathbf{C} \in \mathbb{R}^{S\times n_T}$. We start by considering the following empirical risk minimization problem (neural network training) with some loss function[1] Φ for a typical feed-forward neural network:

$$\min_{\boldsymbol{\theta}\in\Theta} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{S}\sum_{s=1}^{S} \Phi\left(\mathbf{x}_{s,T}\right), \quad \text{subject to: } \mathbf{x}_{s,t+1} = \mathbf{f}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1}), \quad \mathbf{x}_{s,0} = \mathbf{x}_s, \tag{2.1}$$

where $t = 0, \dots T-1$, $s \in \{1, \dots S\}$, $\mathbf{f}_{t+1} : \mathbb{R}^{n(t)} \times \Theta_{t+1} \mapsto \mathbb{R}^{n(t+1)}$ denotes the forward propagation function, $n(t)$ denotes the number of neurons in the t-th layer, the parameter set Θ_{t+1} is a subset of a Euclidean space and $\Theta = \Theta_1 \times \cdots \times \Theta_T$, $\boldsymbol{\theta}_{t+1}$ represents the network parameters corresponding to $(t+1)^{th}$ layer, $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_T)$ such that $\boldsymbol{\theta} \in \Theta$ implies $\boldsymbol{\theta}_{t+1} \in \Theta_{t+1}$, $\mathbf{x}_{s,t}$ represents the hidden states of the network, $T-1$ is the total number of hidden layers in the network. In the case of a fully connected network (FNN), one has:

$$\mathbf{f}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1}) = \sigma_{t+1}\left(\mathbf{W}_{t+1}\mathbf{x}_{s,t} + \mathbf{b}_{t+1}\right),$$

[1] More precisely the notation Φ_s may be used.

where $\mathbf{W}_{t+1} \in \mathbb{R}^{n(t+1)\times n(t)}$, $\mathbf{b}_{t+1} \in \mathbb{R}^{n(t+1)}$, and σ_{t+1} is a nonlinear activation function acting component-wise on its arguments. In the following sections, for clarity, let us limit our discussion to the case of a fully-connected neural network.

2.1. Motivation. Note that during the training of the network in (2.1), it is often desirable to dynamically add new layers (with specific initialization for the parameters), at specific locations in the architecture [12, 22]. Our objective in this work is to develop a mathematically principled criterion that guides such layer additions in a network during the training process.

Our framework is based on the concept of topological derivative which was formally introduced as the "bubble method" for the optimal design of structures [18], and further studied in detail by Sokolowski and Zochowski [52]. The topological derivative is, conceptually, a derivative of a shape functional with respect to infinitesimal changes in its topology, such as adding an infinitesimal hole or crack. The concept finds enormous application in the field of structural mechanics, image processing and inverse problems [1]. In the field of structural mechanics, the concept has been used to detect and locate cracks for a simple model problem: the steady-state heat equation with the heat flux imposed and the temperature measured on the boundary [2]. In the field of image processing, the topological derivative has been used to derive a non-iterative algorithm to perform edge detection and image restoration by studying the impact of an insulating crack in the domain [5]. In the field of inverse problems, the topological derivative approach has been applied to tomographic reconstruction problem [3].

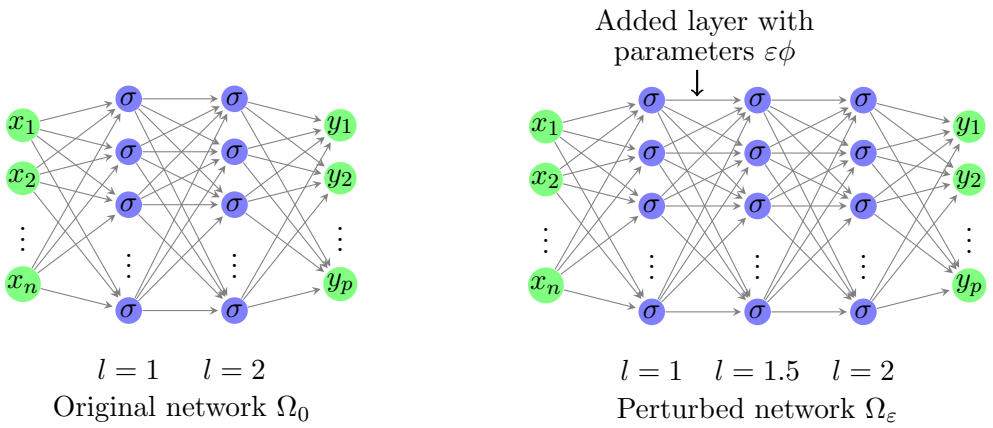


Figure 1. *Schematic view of the topological derivative approach: A new layer with parameters $\varepsilon\boldsymbol{\phi}$ is inserted between the 1^{st} and 2^{nd} layer. If one views ε as the magnitude of perturbation, then for $\varepsilon = 0$, the network Ω_ε should behave exactly the same way as Ω_0 under the standard training process (Residual connections are not shown in the figure).*

In the topological derivative approach, one often studies how the presence of a circular hole (or inclusion of new material) of radius ε at a specific location $\mathbf{z}$ in the domain $\Omega_0 \subset \mathbb{R}^2$ affects the solution of a given partial differential equation (PDE) with prescribed boundary conditions [1].

Analogously, in the context of neural networks, we will examine how the addition of a new layer initialized with parameters $\varepsilon\boldsymbol{\phi}$, at a specific location $l \in \{1, 2, \dots T-1\}$ in the neural

network Ω_0 affects the training equations. Here, $\varepsilon \in \mathbb{R}$ and $\boldsymbol{\phi}$ is the vector of parameters of the new layer. Figure 1 shows an example where a new layer with parameters $\varepsilon\boldsymbol{\phi}$ is inserted at the location between layers 1 and 2 of the network Ω_0. This raises the following question:

In what sense does adding a layer constitute a perturbation of the neural network graph?

We answer this question by drawing analogy with the concept of "topological derivative" from mechanics[2]. Since we are interested in understanding how the addition of a layer, as illustrated in Figure 1, affects the neural network training, it is important to first understand the gradient flow/backpropagation equations for training a neural network. To that end, in subsection 2.2 we present the neural network training problem from an optimal control viewpoint [36, 7]. A key quantity that arises in this framework is the Hamiltonian H_t (see (2.2)). Later in Theorem 2.7, we unveil an interesting connection between the network topological derivative, formally defined in Definition 2.5, and the Hamiltonian H_t in (2.2).

2.2. Optimal control viewpoint for neural network training. Recently, there has been an interest in formulating deep learning as a discrete-time optimal control problem [36, 7]. Note that in the context of popular architectures such as residual neural networks (ResNet), equation (2.1) can be interpreted as discretizations of an optimal control problem subject to an ordinary differential equation constraint [7]. The Hamiltonian for the t-th layer, $H_t : \mathbb{R}^{n(t)} \times \mathbb{R}^{n(t+1)} \times \Theta_{t+1} \mapsto \mathbb{R}$ corresponding to the loss in (2.1) is defined as [57]:

$$H_t\left(\mathbf{x}_{s,t};\ \mathbf{p}_{s,t+1};\ \boldsymbol{\theta}_{t+1}\right) = \mathbf{p}_{s,t+1} \cdot \mathbf{f}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1}), \tag{2.2}$$

where, $\{\mathbf{p}_{s,1},\ \mathbf{p}_{s,2}, \dots \mathbf{p}_{s,T}\}$ denote the adjoint variables computed during a particular gradient descent iteration (backpropagation) as follows [57]:

$$\begin{aligned}
&\underline{\text{Forward propagation}}\\
&\mathbf{x}_{s,t+1} = \nabla_{\mathbf{p}} H_t\left(\mathbf{x}_{s,t}; \mathbf{p}_{s,t+1}; \boldsymbol{\theta}_{t+1}\right) = \mathbf{f}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1}),\\
&\underline{\text{Computing adjoints}}\\
&\mathbf{p}_{s,T} = -\frac{1}{S}\nabla\Phi\left(\mathbf{x}_{s,T}\right),\\
&\mathbf{p}_{s,t} = \nabla_{\mathbf{x}} H_t\left(\mathbf{x}_{s,t};\ \mathbf{p}_{s,t+1};\ \boldsymbol{\theta}_{t+1}\right) = \left[\nabla_{\mathbf{x}}\mathbf{f}_{t+1}(\mathbf{x}_{s,t}; \boldsymbol{\theta}_{t+1})\right]^T\ \mathbf{p}_{s,t+1}, \quad t = 0, \dots T-1,\\
&\underline{\text{Updating parameters via gradient descent}}\\
&\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_{t+1} - \ell\ \nabla_{\boldsymbol{\theta}} H_t\left(\mathbf{x}_{s,t};\ \mathbf{p}_{s,t+1};\ \boldsymbol{\theta}_{t+1}\right) = \boldsymbol{\theta}_{t+1} - \ell\ \left[\nabla_{\boldsymbol{\theta}}\mathbf{f}_{t+1}(\mathbf{x}_{s,t}; \boldsymbol{\theta}_{t+1})\right]^T \mathbf{p}_{s,t+1},
\end{aligned} \tag{2.3}$$

where, $\nabla_{\mathbf{x}}\mathbf{f}_{t+1}$ denotes the gradient of $\mathbf{f}_{t+1}$ with respect to the first argument (i.e. the state $\mathbf{x}$) and $\nabla_{\boldsymbol{\theta}}\mathbf{f}_{t+1}$ denotes the gradient of $\mathbf{f}_{t+1}$ with respect to the second argument (i.e. the parameter $\boldsymbol{\theta}$), ℓ denotes the step-size used in the particular gradient descent iteration, notation '$\leftarrow$' means that the parameters $\boldsymbol{\theta}_{t+1}$ of the $(t+1)-$th layer are being updated. From a Lagrangian viewpoint, the Hamiltonian can be understood as a device to generate the first-order necessary conditions for optimality [57].

[2]The analysis resulting from this viewpoint differs from other works on perturbation analysis in the machine learning community, where the focus is either (i) on how perturbations to input data affect the output of a neural network [20, 47], or (ii) on how perturbations to network parameters influence the network's performance or output [13, 59, 9, 66].

With the concept of the topological derivative introduced in subsection 2.1, and back-propagation described in subsection 2.2, we are now ready to introduce key concepts such as "perturbed network" and "admissible perturbation" leading to the definition of a discrete topological derivative which we call as the "network topological derivative" in our work. The concept of an "admissible perturbation" is based on the idea that, if $\varepsilon = 0$ in Figure 1 (i.e., the perturbation magnitude), the solutions in (2.3) should correspond to those of the original network Ω_0 at any gradient descent iteration, as if the added layer were not present in the graph. This is analogous to the topological derivative approach in mechanics, where setting $\varepsilon = 0$ (the radius of the hole) recovers the solution of the PDE on the original domain Ω_0.

Definition 2.1 (Network perturbation). *Consider $\mathcal{A}$ as the set of all feed-forward neural networks with constant width 'n' ($n \in \mathbb{N}$) in each hidden layer. Let $\Omega_0 \in \mathcal{A}$ be a given neural network with $(T-1)$ hidden layers, and $\sigma(.) : \mathbb{R} \mapsto \mathbb{R}$ be a non-linear activation function. A perturbation Ω_ε from Ω_0 at $l \in \{1, 2, \dots T-1\}$ along the "direction" $\boldsymbol{\phi}$ and with magnitude ε is defined as:*

$$\Omega_\varepsilon = \Omega_0 \oplus (l,\ \varepsilon\boldsymbol{\phi},\ \sigma), \quad s.t.\ \ \Omega_\varepsilon \in \mathcal{A},$$

where $\oplus$ represents the operation of adding a layer between the l^{th} and $(l+1)^{th}$ layer of network Ω_0 initialized with vectorized parameters (weights/biases) $\varepsilon\boldsymbol{\phi}$ and activation σ (refer Figure 1). The added layer is denoted as $\left(l + \frac{1}{2}\right)$ in Figure 1. Note that Ω_ε is a also a function of $l, \boldsymbol{\phi}$ and σ, but for the simplicity of the notation, we omit them.

In the rest of the paper (and in Definition 2.1), the notations Ω_0, Ω_ε denote the neural network with the list of all parameters and activation functions. In addition, we adopt the notation $\Omega_0(\mathbf{x}_s) : \mathbb{R}^{n_0} \mapsto \mathbb{R}^{n_T}$, and $\Omega_\varepsilon(\mathbf{x}_s;\ \varepsilon\boldsymbol{\phi}) : \mathbb{R}^{n_0} \times \mathbb{R}^{n_p} \mapsto \mathbb{R}^{n_T}$ to denote the neural network functions (forward propagation constraint given in (2.1)) where the second argument $\varepsilon\boldsymbol{\phi}$ denotes the parameters of the added layer for Ω_ε. Now it is necessary that by setting the magnitude of perturbation $\varepsilon = 0$ in Definition 2.1, the network $\Omega_\varepsilon\Big|_{\varepsilon=0}$ should behave exactly the same as Ω_0 under the gradient-based training process. We formalize this notion in Definition 2.2 below.

Definition 2.2 (Admissible perturbation). *We say that Ω_ε in Definition 2.1 is an admissible perturbation if:*

$$\Omega_\varepsilon\Big|_{\varepsilon=0} = \Omega_0, \tag{2.4}$$

where '=' in (2.4) is used to denote the fact that the network $\Omega_\varepsilon|_{\varepsilon=0}$ behaves exactly the same as Ω_0 under gradient based training process. In particular, the added layer in $\Omega_\varepsilon\Big|_{\varepsilon=0}$ is redundant and only acts as a message-passing layer. Furthermore, the solutions given by (2.3) for $t = \{0, 1, \dots l, (l+1), \dots T\}$ and the loss $\mathcal{J}$ in (2.1) together with its gradient coincide for both Ω_0 and $\Omega_\varepsilon\Big|_{\varepsilon=0}$ at every gradient descent iteration.

Proposition 2.3 (Construction of an admissible perturbation). *Consider Definition 2.2. The following two steps produce an admissible perturbation:*

1. *The hidden layer propagation equation in (2.1) for Ω_ε in Definition 2.1 satisfies:*

$$
\begin{aligned}
\mathbf{x}_{s,t+1} &= \mathbf{f}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1}) = \mathbf{x}_{s,t} + \mathbf{g}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1}),\ \ t = 1, .., l-1, l+1, ..T-2, \\
\mathbf{x}_{s,l+\frac{1}{2}} &= \mathbf{f}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l};\ \boldsymbol{\theta}_{l+\frac{1}{2}}\right) = \mathbf{x}_{s,l} + \mathbf{g}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l};\ \boldsymbol{\theta}_{l+\frac{1}{2}}\right), \\
\mathbf{x}_{s,l+1} &= \mathbf{f}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}};\ \boldsymbol{\theta}_{l+1}\right) = \mathbf{x}_{s,l+\frac{1}{2}} + \mathbf{g}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}};\ \boldsymbol{\theta}_{l+1}\right).
\end{aligned} \tag{2.5}
$$

2. *Choose the activation function σ such that $\mathbf{g}_t(.;\ .)$ is continuously differentiable w.r.t both the arguments and:*

$$
\mathbf{g}_t(\mathbf{x}_{s,t};\ \mathbf{0}) = \mathbf{0},\ \ \nabla_{\mathbf{x}}\mathbf{g}_t(\mathbf{x}_{s,t};\ \mathbf{0}) = \mathbf{0},\ \ \nabla_{\boldsymbol{\theta}}\mathbf{g}_t(\mathbf{x}_{s,t};\ \mathbf{0}) = \mathbf{0},\ \ t = 2, \ldots l, l+\frac{1}{2}, l+1, \ldots T-1.
$$

Proof:. Let us compute the states and adjoints in (2.3) for network Ω_ε corresponding to the first gradient descent iteration. The forward propagation of residual neural network Ω_ε from l^{th} layer to $(l+1)^{th}$ layer can be written as:

$$
\mathbf{x}_{s,l+\frac{1}{2}} = \mathbf{f}_{l+\frac{1}{2}}(\mathbf{x}_{s,l};\ \varepsilon\boldsymbol{\phi}) = \mathbf{x}_{s,l} + \mathbf{g}_{l+\frac{1}{2}}(\mathbf{x}_{s,l};\ \varepsilon\boldsymbol{\phi}), \tag{2.6a}
$$

$$
\mathbf{x}_{s,l+1} = \mathbf{f}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}};\ \boldsymbol{\theta}_{l+1}\right) = \mathbf{x}_{s,l+\frac{1}{2}} + \mathbf{g}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}};\ \boldsymbol{\theta}_{l+1}\right). \tag{2.6b}
$$

Now, for $\varepsilon = 0$ (2.6a) gives

$$
\mathbf{x}_{s,l+\frac{1}{2}} = \mathbf{x}_{s,l},\ \Longrightarrow\ \mathbf{x}_{s,l+1} = \mathbf{x}_{s,l} + \mathbf{g}_{l+1}\left(\mathbf{x}_{s,l};\ \boldsymbol{\theta}_{l+1}\right) = \mathbf{f}_{l+1}\left(\mathbf{x}_{s,l};\ \boldsymbol{\theta}_{l+1}\right),
$$

where we have used the fact that $\mathbf{g}_{l+\frac{1}{2}}(\mathbf{x}_{s,l};\ \mathbf{0}) = \mathbf{0}$. Therefore, adding a new layer recovers the forward propagation equations of the original network Ω_0 when $\varepsilon = 0$ for the first gradient descent iteration. Now, the backward propagation of the adjoint using (2.3) can be written as:

$$
\begin{aligned}
\mathbf{p}_{s,l+\frac{1}{2}} &= \left[\nabla_{\mathbf{x}}\mathbf{f}_{l+1}(\mathbf{x}_{s,l+\frac{1}{2}};\boldsymbol{\theta}_{l+1})\right]^T \mathbf{p}_{s,l+1} = \mathbf{p}_{s,l+1} + \left[\nabla_{\mathbf{x}}\mathbf{g}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}};\ \boldsymbol{\theta}_{l+1}\right)\right]^T \mathbf{p}_{s,l+1}, \\
\mathbf{p}_{s,l} &= \left[\nabla_{\mathbf{x}}\mathbf{f}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l};\ \varepsilon\boldsymbol{\phi}\right)\right]^T \mathbf{p}_{s,l+\frac{1}{2}} = \mathbf{p}_{s,l+\frac{1}{2}} + \left[\nabla_{\mathbf{x}}\mathbf{g}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l};\ \varepsilon\boldsymbol{\phi}\right)\right]^T \mathbf{p}_{s,l+\frac{1}{2}}.
\end{aligned} \tag{2.7}
$$

Again using $\nabla_{\mathbf{x}}\mathbf{g}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l};\ \mathbf{0}\right) = \mathbf{0}$, we conclude

$$
\mathbf{p}_{s,l} = \mathbf{p}_{s,l+\frac{1}{2}}.
$$

Therefore, when $\varepsilon = 0$, the adjoint of the perturbed network and the original one are the same for the first gradient descent iteration. Finally, the third condition in Item 2 for the added layer, i.e. $\nabla_{\boldsymbol{\theta}}\mathbf{g}_{l+\frac{1}{2}}(\mathbf{x}_{s,l};\ \mathbf{0}) = \mathbf{0}$ implies that $\nabla_{\boldsymbol{\theta}}\mathbf{f}_{l+\frac{1}{2}}(\mathbf{x}_{s,l};\ \mathbf{0}) = \mathbf{0}$. Therefore, for $\varepsilon = 0$, the gradients for updating the parameters of the added layer given by the last equation in (2.3) is $\mathbf{0}$, leading to the parameters of the added layer not getting updated at the end of the first gradient descent iteration. Applying the above arguments recursively for subsequent gradient descent iterations concludes the proof.

Remark 2.4.

1. Note that for typical neural networks such as FNN's and CNN's, condition 2 can be easily realized by choosing continuously differentiable activation function σ with the property that $\sigma(0) = 0$, and $\sigma'(0) = 0$.
2. The activation $\sigma(x)$ for the added layer can be constructed as a linear combination of some existing activation functions, i.e $\sigma(x) = \alpha_1\sigma_1(x)+\sigma_2(x)$, where $\sigma_1(0) = \sigma_2(0) = 0$ and $\alpha_1 = -\frac{\sigma_2'(0)}{\sigma_1'(0)}$, $\sigma_1'(0) \neq 0$. For instance, the set $\mathcal{F} = \{\sigma_1,\ \sigma_2\}$ can be chosen as follows:

$$\mathcal{F} = \{\text{Swish},\ \tanh\},\ \{\text{Mish},\ \tanh\},\ \{\text{Swish},\ \text{Mish}\},\ \ \textit{etc.} \tag{2.8}$$

Note that it can be proved that the constructed function $\sigma(x)$ based on the choices in (2.8) is universal[3] by verifying the conditions in [27] or [10].

2.3. Network topological derivative. In this section, we define the "network topological derivative" for a feed-forward neural network and derive its explicit expression. Note that the admissible perturbation (adding a layer) defined in Definition 2.2 can be viewed as an infinitesimal change in the neural network topology. In order to formally define the "network topological derivative" let us first rewrite the loss function (2.1) as follows:

$$\mathcal{J}(\Omega_\varepsilon) = \frac{1}{S}\sum_{s=1}^{S} \Phi\left(\Omega_\varepsilon\left(\mathbf{x}_{s,0};\ \varepsilon\boldsymbol{\phi}\right)\right), \tag{2.9}$$

where $\varepsilon\boldsymbol{\phi}$ is the initialization of the added layer in Ω_ε (see Definition 2.1).

Definition 2.5 (Network topological derivative). *Consider an admissible perturbation Ω_ε in Definition 2.2. We say the loss functional $\mathcal{J}$ in (2.9) admits a network topological derivative—denoted as $d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma))$—at $\Omega_0 \in \mathcal{A}$ and at the location $l \in \{1, 2 \ldots T-1\}$ along the direction $\boldsymbol{\phi}$ if there exists a function $q : \mathbb{R}^+ \mapsto \mathbb{R}^+$ with $\lim_{\varepsilon\downarrow 0} q(\varepsilon) = 0$ such that the following "topological asymptotic expansion" holds:*

$$\mathcal{J}(\Omega_\varepsilon) = \mathcal{J}(\Omega_0) - q(\varepsilon)d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) + o(q(\varepsilon)), \tag{2.10}$$

where $o(q(\varepsilon))$ is the remainder[4]. (2.10) is equivalent to the existence of the following limit:

$$d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) = -\lim_{\varepsilon\downarrow 0}\frac{\mathcal{J}(\Omega_\varepsilon) - \mathcal{J}(\Omega_0)}{q(\varepsilon)}. \tag{2.11}$$

Remark 2.6. Note that the '-' sign is used in (2.11) to ensure that a positive derivative favors layer addition and vice versa. Since for an admissible perturbation Ω_ε in Definition 2.2,

[3]Let $\mathcal{NN}^{\sigma}_{n,m,k}$ represent the class of functions $\mathbb{R}^n \to \mathbb{R}^m$ described by feedforward neural networks of arbitrary number of hidden layers, each with k neurons and activation function σ. Let $K \subseteq \mathbb{R}^n$ be compact. We say that the activation function $\sigma(x)$ is universal if $\mathcal{NN}^{\sigma}_{n,m,k}$ is dense in $C(K;\mathbb{R}^m)$ with respect to the uniform norm.

[4]The notation $p(\varepsilon) = o(q(\varepsilon))$ means that $\lim_{\varepsilon\to 0}\frac{p(\varepsilon)}{q(\varepsilon)} = 0$, where $q(\varepsilon) \neq 0$ for all $\varepsilon \neq 0$.

we have $\mathcal{J}\left(\Omega_\varepsilon\Big|_{\varepsilon=0}\right) = \mathcal{J}(\Omega_0)$, the "topological asymptotic expansion" in (2.10), can be understood as the following Taylor series expansion:

$$\mathcal{J}(\Omega_\varepsilon) = \mathcal{J}\left(\Omega_\varepsilon\Big|_{\varepsilon=0}\right) + \varepsilon\left[\frac{d}{d\varepsilon}\left(\mathcal{J}(\Omega_\varepsilon)\right)\right]_{\varepsilon=0} + \frac{\varepsilon^2}{2}\left[\frac{d^2}{d\varepsilon^2}\left(\mathcal{J}(\Omega_\varepsilon)\right)\right]_{\varepsilon=0} + \ldots, \tag{2.12}$$

where $\mathcal{J}(\Omega_\varepsilon)$ is given by (2.9). In Theorem 2.7 we will unveil a non-trivial connection between the Hamiltonian H_l introduced in (2.2) with the network topological derivative in (2.11).

Theorem 2.7 (Existence of network topological derivative). *Assume the conditions in Proposition 2.3 and the loss $\mathcal{J}$ in (2.1). Further, let $\mathcal{X}_t$ be the set of all possible states $\mathbf{x}_{s,t}$ reachable*[5] *at layer t from all possible initial sample $\mathbf{x}_{s,0}$ and trainable parameters in Θ. Assume that*

1. *Φ has bounded second and third order derivatives on $\mathcal{X}_T$;*
2. *$\mathbf{f}_{t+1}(.;\ .)$ has bounded second and third order derivatives on $\mathcal{X}_t \times \Theta_{t+1}$.*

Then, the network topological derivative given by (2.11) exists with $q(\varepsilon) = \varepsilon^2$ and is given by:

$$d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) = \frac{1}{2}\sum_{s=1}^{S} \boldsymbol{\phi}^T\ \nabla^2_{\boldsymbol{\theta}} H_l\left(\mathbf{x}_{s,l}; \mathbf{p}_{s,l}; \boldsymbol{\theta}\right)\Big|_{\boldsymbol{\theta}=\mathbf{0}}\ \boldsymbol{\phi}, \tag{2.13}$$

where H_l is the Hamiltonian defined in (2.2)[6].

Proof:. Let us represent the parameters of the perturbed network Ω_ε as $\boldsymbol{\theta}^\varepsilon$ and the parameters of the original network Ω_0 as $\boldsymbol{\theta}^0$. For an admissible perturbation Ω_ε in Definition 2.2, Ω_0 is equivalent to $\Omega_\varepsilon|_{\varepsilon=0}$. We thus identify Ω_0 with $\Omega_\varepsilon|_{\varepsilon=0}$. By Definition 2.1 $\boldsymbol{\theta}^\varepsilon_i = \boldsymbol{\theta}^0_i,\ \forall i \neq l+\frac{1}{2}$. Define

$$\delta\mathbf{x}_{s,t} = \mathbf{x}^{\boldsymbol{\theta}^\varepsilon}_{s,t} - \mathbf{x}^{\boldsymbol{\theta}^0}_{s,t}, \quad \delta\mathbf{p}_{s,t} = \mathbf{p}^{\boldsymbol{\theta}^\varepsilon}_{s,t} - \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t},$$

and we have

$$\delta\mathbf{x}_{s,t} = 0, \quad \forall t \leq l.$$

Applying definition (2.2) for both Ω_ε and Ω_0 gives

$$\sum_{t=0}^{T-1}\left(H_t\left(\mathbf{x}^{\theta^0}_{s,t} + \delta\mathbf{x}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1} + \delta\mathbf{p}_{s,t+1};\ \boldsymbol{\theta}^\varepsilon_{t+1}\right) - (\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1} + \delta\mathbf{p}_{s,t+1})^T(\mathbf{x}^{\theta^0}_{s,t+1} + \delta\mathbf{x}_{s,t+1})\right) = 0, \tag{2.14a}$$

$$\sum_{t=0}^{T-1}\left(H_t\left(\mathbf{x}^{\theta^0}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\ \boldsymbol{\theta}^0_{t+1}\right) - (\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1})^T(\mathbf{x}^{\theta^0}_{s,t+1})\right) = 0. \tag{2.14b}$$

[5] We say that a state $\mathbf{x}_{s,t+1}$ is reachable at layer $(t+1)$ if there exists $\boldsymbol{\theta} \in \Theta$ and sample $\mathbf{x}_{s,0}$ such that the constraint in (2.1) holds.

[6] Note the change in subscript for the adjoint variable $\mathbf{p}$ while evaluating the Hamiltonian.

Note that the summation from $t = 0$ to $t = T-1$ also includes $t = \left(l + \frac{1}{2}\right)$. Subtracting (2.14b) from (2.14a) yields:

$$
\begin{aligned}
&\sum_{t=0}^{T-1} H_t\left(\mathbf{x}_{s,t}^{\theta^0} + \delta\mathbf{x}_{s,t};\ \mathbf{p}_{s,t+1}^{\boldsymbol{\theta}^0} + \delta\mathbf{p}_{s,t+1};\ \boldsymbol{\theta}_{t+1}^{\varepsilon}\right) - H_t\left(\mathbf{x}_{s,t}^{\theta^0};\ \mathbf{p}_{s,t+1}^{\boldsymbol{\theta}^0};\ \boldsymbol{\theta}_{t+1}^{0}\right) = \\
&\left(\delta\mathbf{p}_{s,l+\frac{1}{2}} \cdot \delta\mathbf{x}_{s,l+\frac{1}{2}}\right) + \left(\mathbf{p}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0} \cdot \delta\mathbf{x}_{s,l+\frac{1}{2}}\right) \\
&\underbrace{+ \sum_{t=l}^{T-1} \left(\delta\mathbf{p}_{s,t+1} \cdot \delta\mathbf{x}_{s,t+1}\right)}_{I} + \underbrace{\sum_{t=l}^{T-1} \left(\mathbf{p}_{s,t+1}^{\boldsymbol{\theta}^0} \cdot \delta\mathbf{x}_{s,t+1}\right)}_{II} + \underbrace{\sum_{t=0}^{T-1} \left(\delta\mathbf{p}_{s,t+1} \cdot \mathbf{x}_{s,t+1}^{\boldsymbol{\theta}_0}\right)}_{III}.
\end{aligned} \tag{2.15}
$$

Now let's analyze each term on the right-hand side of (2.15).

Analyzing Term I in (2.15)

$$
\delta\mathbf{x}_{s,l+\frac{1}{2}} = \mathbf{f}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l}^{\boldsymbol{\theta}_\varepsilon};\ \varepsilon\boldsymbol{\phi}\right) - \mathbf{f}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l}^{\boldsymbol{\theta}_0};\ \mathbf{0}\right).
$$

Assuming that $\mathbf{f}_{l+\frac{1}{2}}(.;.)$ has bounded second order derivative w.r.t the second argument and noting that $\mathbf{x}_{s,l}^{\boldsymbol{\theta}_\varepsilon} = \mathbf{x}_{s,l}^{\boldsymbol{\theta}_0}$, Taylor series approximation about $\varepsilon = 0$ gives:

$$
\delta\mathbf{x}_{s,l+\frac{1}{2}} = \varepsilon\ \nabla_{\boldsymbol{\theta}}\mathbf{f}_{l+\frac{1}{2}}\left(\mathbf{x}_{s,l}^{\boldsymbol{\theta}_0};\ \mathbf{0}\right) \cdot \boldsymbol{\phi} + \mathcal{O}(\varepsilon^2) = 0 + \mathcal{O}(\varepsilon^2),
$$

where, $\nabla_{\boldsymbol{\theta}}\mathbf{f}_{l+\frac{1}{2}}$ denotes the derivative of $\mathbf{f}_{l+\frac{1}{2}}$ w.r.t the second argument. Note that one has $\nabla_{\boldsymbol{\theta}}\mathbf{f}_{l+\frac{1}{2}}(\mathbf{x}_{s,l};\ \mathbf{0}) = \mathbf{0}$ by condition 2 of Proposition 2.3. Therefore, we have:

$$
\delta\mathbf{x}_{s,l+1} = \mathbf{f}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0} + \mathcal{O}(\varepsilon^2);\ \boldsymbol{\theta}_{l+1}^{\varepsilon}\right) - \mathbf{f}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0};\ \boldsymbol{\theta}_{l+1}^{0}\right).
$$

Now, considering the Taylor expansion about $\mathbf{x}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0}$ and assuming that $\mathbf{f}_{l+1}(.;.)$ has bounded second order derivative w.r.t the first argument, it is easy to see that by Taylor's theorem:

$$
\begin{aligned}
\delta\mathbf{x}_{s,l+1} &= \nabla_{\mathbf{x}}\mathbf{f}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0};\ \boldsymbol{\theta}_{l+1}^{0}\right) \cdot \mathcal{O}(\varepsilon^2) + \mathcal{O}(\varepsilon^4) \\
&= \left[\mathcal{I} + \nabla_{\mathbf{x}}\mathbf{g}_{l+1}\left(\mathbf{x}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0};\ \boldsymbol{\theta}_{l+1}^{0}\right)\right] \cdot \mathcal{O}(\varepsilon^2) + \mathcal{O}(\varepsilon^4),
\end{aligned} \tag{2.16}
$$

where we have used the fact that $\boldsymbol{\theta}_{l+1}^{\varepsilon} = \boldsymbol{\theta}_{l+1}^{0}$ and used the ResNet propagation equation in (2.6). Therefore, we have:

$$
\delta\mathbf{x}_{s,l+1} = \mathcal{O}(\varepsilon^2).
$$

Applying the above arguments recursively, one can show that:

$$
\delta\mathbf{x}_{s,i} = \mathcal{O}(\varepsilon^2), \qquad i = \left(l + \frac{1}{2}\right), (l+1), \ldots T. \tag{2.17}
$$

Now from (2.3) we have:

$$\mathbf{p}_{s,T}^{\boldsymbol{\theta}^\varepsilon} = -\frac{1}{S}\nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^\varepsilon}\right), \quad \mathbf{p}_{s,T}^{\boldsymbol{\theta}^0} = -\frac{1}{S}\nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0}\right),$$

for networks Ω_ε and Ω_0 respectively. Therefore,

$$\begin{aligned}\delta\mathbf{p}_{s,T} &= \mathbf{p}_{s,T}^{\boldsymbol{\theta}^\varepsilon} - \mathbf{p}_{s,T}^{\boldsymbol{\theta}^0} \\ &= \frac{1}{S}\left[\nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0}\right) - \nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^\varepsilon}\right)\right] = \frac{1}{S}\left[\nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0}\right) - \nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0} + \delta\mathbf{x}_{s,T}\right)\right] \\ &= \frac{1}{S}\left[\nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0}\right) - \nabla\Phi\left(\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0} + \mathcal{O}(\varepsilon^2)\right)\right].\end{aligned} \tag{2.18}$$

Now assuming that Φ has bounded third order derivative, Taylor series approximation about $\mathbf{x}_{s,T}^{\boldsymbol{\theta}^0}$ gives:

$$\delta\mathbf{p}_{s,T} = \mathcal{O}(\varepsilon^2).$$

Further, from (2.3) we also have:

$$\begin{aligned}\mathbf{p}_{s,T-1}^{\boldsymbol{\theta}^\varepsilon} &= \left[\nabla_{\mathbf{x}}\mathbf{f}_T\left(\mathbf{x}_{s,T-1}^{\boldsymbol{\theta}^\varepsilon}; \boldsymbol{\theta}_T^\varepsilon\right)\right]^T \mathbf{p}_{s,T}^{\boldsymbol{\theta}^\varepsilon} = \left[\nabla_{\mathbf{x}}\mathbf{f}_T\left(\mathbf{x}_{s,T-1}^{\boldsymbol{\theta}^0} + \mathcal{O}(\varepsilon^2);\ \boldsymbol{\theta}_T^0\right)\right]^T \left(\mathbf{p}_{s,T}^{\boldsymbol{\theta}^0} + \mathcal{O}(\varepsilon^2)\right), \\ \mathbf{p}_{s,T-1}^{\boldsymbol{\theta}^0} &= \left[\nabla_{\mathbf{x}}\mathbf{f}_T\left(\mathbf{x}_{s,T-1}^{\boldsymbol{\theta}^0}; \boldsymbol{\theta}_T^0\right)\right]^T \mathbf{p}_{s,T}^{\boldsymbol{\theta}^0}.\end{aligned} \tag{2.19}$$

Again, under the assumption that $\mathbf{f}_T(.;\ .)$ has bounded third order derivative w.r.t the first argument, it is easy to show that:

$$\delta\mathbf{p}_{s,T-1} = \mathbf{p}_{s,T-1}^{\boldsymbol{\theta}^\varepsilon} - \mathbf{p}_{s,T-1}^{\boldsymbol{\theta}^0} = \mathcal{O}(\varepsilon^2),$$

where we have considered the Taylor expansion of $\nabla_{\mathbf{x}}\mathbf{f}_T$ about $\mathbf{x}_{s,T-1}^{\boldsymbol{\theta}^0}$. Now using the above arguments recursively, one can show that:

$$\delta\mathbf{p}_{s,i} = \mathcal{O}(\varepsilon^2), \quad i = 1, \ldots T. \tag{2.20}$$

Therefore, from (2.17) and (2.20) we have:

$$\delta\mathbf{p}_{s,t} \cdot \delta\mathbf{x}_{s,t} = \mathcal{O}(\varepsilon^4), \quad t = \left(l + \frac{1}{2}\right), (l+1), \ldots T. \tag{2.21}$$

Analyzing Term II and III in (2.15)

$$\begin{aligned}&\sum_{t=l}^{T-1}\left(\mathbf{p}_{s,t+1}^{\boldsymbol{\theta}^0} \cdot \delta\mathbf{x}_{s,t+1}\right) + \sum_{t=0}^{T-1}\left(\delta\mathbf{p}_{s,t+1} \cdot \mathbf{x}_{s,t+1}^{\boldsymbol{\theta}_0}\right) = \\ &\sum_{t=l+1}^{T-1}\left(\mathbf{p}_{s,t}^{\boldsymbol{\theta}^0} \cdot \delta\mathbf{x}_{s,t}\right) + \mathbf{p}_{s,T}^{\boldsymbol{\theta}^0} \cdot \delta\mathbf{x}_{s,T} + \sum_{t=0}^{T-1}\left(\delta\mathbf{p}_{s,t+1} \cdot \mathbf{x}_{s,t+1}^{\boldsymbol{\theta}_0}\right).\end{aligned} \tag{2.22}$$

Now from (2.3) note that:

$$\mathbf{p}^{\boldsymbol{\theta}^0}_{s,T}\cdot\delta\mathbf{x}_{s,T} = -\frac{1}{S}\nabla\Phi\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,T}\right)\cdot\delta\mathbf{x}_{s,T} = -\frac{1}{S}\left[\Phi\left(\mathbf{x}^{\boldsymbol{\theta}^\varepsilon}_{s,T}\right) - \Phi\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,T}\right)\right] + \mathcal{O}\left(\|\delta\mathbf{x}_{s,T}\|_2^2\right).$$

where we have assumed Φ has bounded second order derivative. Therefore,

$$\sum_{s=1}^{S}\mathbf{p}^{\boldsymbol{\theta}^0}_{s,T}\cdot\delta\mathbf{x}_{s,T} = \mathcal{J}(\Omega_0) - \mathcal{J}(\Omega_\varepsilon) + \mathcal{O}(\varepsilon^4). \tag{2.23}$$

Further, the remaining terms in (2.22) can be written in terms of the Hamiltonian as,

$$\begin{aligned}&\sum_{t=l+1}^{T-1}\left(\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t}\cdot\delta\mathbf{x}_{s,t}\right) + \sum_{t=0}^{T-1}\left(\delta\mathbf{p}_{s,t+1}\cdot\mathbf{x}^{\boldsymbol{\theta}_0}_{s,t+1}\right) = \\ &\sum_{t=l+1}^{T-1}\nabla_{\mathbf{x}}H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\boldsymbol{\theta}^0_{t+1}\right)\cdot\delta\mathbf{x}_{s,t} + \sum_{t=0}^{T-1}\nabla_{\mathbf{p}}H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\boldsymbol{\theta}^0_{t+1}\right)\cdot\delta\mathbf{p}_{s,t+1}.\end{aligned} \tag{2.24}$$

Finally, substituting (2.23), (2.24), (2.21) in (2.15) and summing over all the training data points we have:

(2.25)

$$\begin{aligned}&\mathcal{J}(\Omega_0) - \mathcal{J}(\Omega_\varepsilon) + \sum_{s=1}^{S}\sum_{t=l+1}^{T-1}\nabla_{\mathbf{x}}H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\boldsymbol{\theta}^0_{t+1}\right)\cdot\delta\mathbf{x}_{s,t} \\ &+\sum_{s=1}^{S}\sum_{t=0}^{T-1}\nabla_{\mathbf{p}}H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\boldsymbol{\theta}^0_{t+1}\right)\cdot\delta\mathbf{p}_{s,t+1} + \sum_{s=1}^{S}\nabla_{\mathbf{x}}H_{l+\frac{1}{2}}\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}},\mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+1},\boldsymbol{\theta}^0_{l+1}\right)\cdot\delta\mathbf{x}_{s,l+\frac{1}{2}} + \mathcal{O}(\varepsilon^4) \\ &= \sum_{s=1}^{S}\sum_{t=0}^{T-1}\left[H_t\left(\mathbf{x}^{\theta^0}_{s,t}+\delta\mathbf{x}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1}+\delta\mathbf{p}_{s,t+1};\ \boldsymbol{\theta}^\varepsilon_{t+1}\right) - H_t\left(\mathbf{x}^{\theta^0}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\ \boldsymbol{\theta}^0_{t+1}\right)\right].\end{aligned}$$

The R.H.S of (2.25) can be simplified by considering the Taylor series expansion about $\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t},\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t},\ \boldsymbol{\theta}^\varepsilon_{t+1}\right)$as:

$$\begin{aligned}&H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t}+\delta\mathbf{x}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1}+\delta\mathbf{p}_{s,t+1};\ \boldsymbol{\theta}^\varepsilon_{t+1}\right) - H_t\left(\mathbf{x}^{\theta^0}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\ \boldsymbol{\theta}^0_{t+1}\right) = \\ &H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\ \boldsymbol{\theta}^\varepsilon_{t+1}\right) - H_t\left(\mathbf{x}^{\theta^0}_{s,t};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\ \boldsymbol{\theta}^0_{t+1}\right) + \nabla_{\mathbf{x}}H_t(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\boldsymbol{\theta}^\varepsilon_{t+1})\cdot\delta\mathbf{x}_{s,t}+ \\ &\nabla_{\mathbf{p}}H_t\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,t};\mathbf{p}^{\boldsymbol{\theta}^0}_{s,t+1};\boldsymbol{\theta}^\varepsilon_{t+1}\right)\cdot\delta\mathbf{p}_{s,t+1} + \mathcal{O}(\varepsilon^4).\end{aligned}$$

Substituting the above in (2.25) and simplifying (note that $\boldsymbol{\theta}^0_t = \boldsymbol{\theta}^\varepsilon_t,\ \forall t \neq l+\frac{1}{2}$) we have:

$$\begin{aligned}\mathcal{J}(\Omega_0) - \mathcal{J}(\Omega_\varepsilon) = &\sum_{s=1}^{S}H_l\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,l};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}};\ \varepsilon\boldsymbol{\phi}\right) - H_l\left(\mathbf{x}^{\theta^0}_{s,l};\ \mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}};\ \mathbf{0}\right) + \mathcal{O}(\varepsilon^4) \\ &+\left[\nabla_{\mathbf{x}}H_l\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,l},\mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}},\varepsilon\boldsymbol{\phi}\right) - \nabla_{\mathbf{x}}H_l\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,l},\mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}},\mathbf{0}\right)\right]\cdot\delta\mathbf{x}_{s,l} \\ &+\left[\nabla_{\mathbf{p}}H_l\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,l},\mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}},\varepsilon\boldsymbol{\phi}\right) - \nabla_{\mathbf{p}}H_l\left(\mathbf{x}^{\boldsymbol{\theta}^0}_{s,l},\mathbf{p}^{\boldsymbol{\theta}^0}_{s,l+\frac{1}{2}},\mathbf{0}\right)\right]\cdot\delta\mathbf{p}_{s,l}.\end{aligned} \tag{2.26}$$

Now considering the Taylor expansion about $\varepsilon = 0$ for the last two terms in (2.26) and noting that $\nabla_{\mathbf{x}\boldsymbol{\theta}} H_l\left(\mathbf{x}_{s,l}^{\boldsymbol{\theta}^0}, \mathbf{p}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0}, \mathbf{0}\right) = \mathbf{0}$ and $\nabla_{\mathbf{p}\boldsymbol{\theta}} H_l\left(\mathbf{x}_{s,l}^{\boldsymbol{\theta}^0}, \mathbf{p}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}^0}, \mathbf{0}\right) = \mathbf{0}$ due to assumption 2, it follows that the last two terms in (2.26) are of order $\mathcal{O}(\varepsilon^4)$. Therefore, we finally have:

$$\mathcal{J}(\Omega_0) - \mathcal{J}(\Omega_\varepsilon) = \sum_{s=1}^{S} H_l\left(\mathbf{x}_{s,l}^{\boldsymbol{\theta}^0};\ \mathbf{p}_{s,l}^{\boldsymbol{\theta}^0};\ \varepsilon\boldsymbol{\phi}\right) - H_l\left(\mathbf{x}_{s,l}^{\theta^0};\ \mathbf{p}_{s,l}^{\boldsymbol{\theta}^0};\ \mathbf{0}\right) + \mathcal{O}(\varepsilon^4).$$

where we have also used the fact $\mathbf{p}_{s,l+\frac{1}{2}}^{\boldsymbol{\theta}_0} = \mathbf{p}_{s,l}^{\boldsymbol{\theta}_0}$ since the added layer with zero weights and biases acts as a message-passing layer due to Proposition 2.3. Finally, applying Taylor series expansion about $\varepsilon = 0$ and assuming that $\mathbf{f}_t(.;\ .)$ has bounded third order derivative w.r.t second argument we have:

$$\mathcal{J}(\Omega_0) - \mathcal{J}(\Omega_\varepsilon) = \sum_{s=1}^{S} \left(\varepsilon \nabla_{\boldsymbol{\theta}} H_l\Big|_{\boldsymbol{\theta}=0} \cdot \boldsymbol{\phi} + \frac{\varepsilon^2}{2} \boldsymbol{\phi}^T\ \nabla_{\boldsymbol{\theta}}^2 H_l\Big|_{\boldsymbol{\theta}=0}\ \boldsymbol{\phi} + \mathcal{O}(\varepsilon^3) \right). \tag{2.27}$$

Note that $\nabla_{\boldsymbol{\theta}} H_l\Big|_{\boldsymbol{\theta}=0} = \mathbf{0}$ by condition 2 of Proposition 2.3. Therefore, the topological derivative is computed as:

$$d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) = -\lim_{\varepsilon \downarrow 0} \frac{\mathcal{J}(\Omega_\varepsilon) - \mathcal{J}(\Omega_0)}{\varepsilon^2} = \frac{1}{2} \sum_{s=1}^{S} \boldsymbol{\phi}^T\ \nabla_{\boldsymbol{\theta}}^2 H_l\Big|_{\boldsymbol{\theta}=0}\ \boldsymbol{\phi},$$

thereby concluding the proof.

Corollary 2.8. *Assume the conditions in Theorem 2.7. For the expansion (2.12) there hold:*

- $\left[\frac{d}{d\varepsilon}\left(\mathcal{J}\left(\Omega_\varepsilon\right)\right)\right]_{\varepsilon=0} = \nabla_{\boldsymbol{\theta}} \mathcal{J}\left(\Omega_\varepsilon\right)\Big|_{\varepsilon=0} \cdot \boldsymbol{\phi} = -\sum_{s=1}^{S} \nabla_{\boldsymbol{\theta}} H_l\Big|_{\boldsymbol{\theta}=0} \cdot \boldsymbol{\phi} = 0,$
- $\frac{1}{2}\left[\frac{d^2}{d\varepsilon^2}\left(\mathcal{J}\left(\Omega_\varepsilon\right)\right)\right]_{\varepsilon=0} = \frac{1}{2}\boldsymbol{\phi}^T \nabla_{\boldsymbol{\theta}}^2 \mathcal{J}\left(\Omega_\varepsilon\right)\Big|_{\varepsilon=0} \boldsymbol{\phi} = -\frac{1}{2}\sum_{s=1}^{S} \boldsymbol{\phi}^T\ \nabla_{\boldsymbol{\theta}}^2 H_l\Big|_{\boldsymbol{\theta}=0}\ \boldsymbol{\phi}$, *and*
- $d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) = -\frac{1}{2}\boldsymbol{\phi}^T \nabla_{\boldsymbol{\theta}}^2 \mathcal{J}\left(\Omega_\varepsilon\right)\Big|_{\varepsilon=0} \boldsymbol{\phi} = \frac{1}{2}\sum_{s=1}^{S} \boldsymbol{\phi}^T\ \nabla_{\boldsymbol{\theta}}^2 H_l\Big|_{\boldsymbol{\theta}=0}\ \boldsymbol{\phi}.$

Proof:. Comparing equations (2.27) and (2.12) and noting that $\mathcal{J}\left(\Omega_\varepsilon\Big|_{\varepsilon=0}\right) = \mathcal{J}\left(\Omega_0\right)$ concludes the proof.

As can be seen from Corollary 2.8, the directional derivative along the direction $\boldsymbol{\phi}$ of the loss function $\mathcal{J}$ at $\mathbf{0}$ vanishes. That is, when the newly added layer is a message passing layer, it does not affect the loss function, and this is consistent with Proposition 2.3. Ignoring the factor $1/2$, we can also see that that network derivative is the negative of the Hessian of the loss function at $\mathbf{0}$ acting on $\boldsymbol{\phi}$ both on the left and the right. Corollary 2.8 was also the reason we introduced the generalized notion of network derivative in (2.11) that allows us to deploy the Hessian as the measure of sensitivity when the gradient vanishes.

2.4. Derivation of our adaptation algorithm. We now exploit the derived closed-form expression for the network topological derivative (2.13) to devise a greedy algorithm for architecture adaptation. Given the network derivative in (2.13), the criterion is obvious: **we**

add a new layer at the depth l^* if the network derivative there is the largest as that will incur the greatest decrease of the loss function according to (2.10). In fact, (2.13) helps us determine not only the best location l^* to add a new layer but also an appropriate initialization for the training process when the new layer is added. To begin, we note that the maximal network derivative at a location l and the corresponding direction Φ_l (and hence the most appropriate initialization of the newly added layer, has it been added) are determined by

$$\begin{aligned} \Lambda_l &= \max_{\boldsymbol{\phi}} \left(d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) \right), \quad s.t.\ \|\boldsymbol{\phi}\|_2^2 = 1, \\ \Phi_l &= \operatorname*{argmax}_{\boldsymbol{\phi}} \left(d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma)) \right), \quad s.t.\ \|\boldsymbol{\phi}\|_2^2 = 1, \end{aligned} \tag{2.28}$$

where we have considered initializing the new layer with unit direction $\boldsymbol{\phi}$ to avoid an arbitrary scaling in (2.28). *A new layer is added at the location* $l^* = \operatorname*{argmax}_l \{\Lambda_l\}_{l=1}^{T-1}$. It is clear that Λ_l is the maximum eigenvalue and Φ_l is the corresponding eigenvector to the following eigenvalue problem:

$$Q_l \boldsymbol{\phi} = \lambda \boldsymbol{\phi}, \quad Q_l = \frac{1}{2} \sum_{s=1}^{S} \nabla_{\boldsymbol{\theta}}^2 H_l \left(\mathbf{x}_{s,l}; \mathbf{p}_{s,l}; \boldsymbol{\theta} \right) \Big|_{\boldsymbol{\theta}=0} = -\frac{1}{2} \nabla_{\boldsymbol{\theta}}^2 \mathcal{J} \left(\Omega_\varepsilon \right) \Big|_{\varepsilon=0}. \tag{2.29}$$

Theorem 2.9. *Consider adding a new layer with weights/biases $\varepsilon\boldsymbol{\phi}$ at depth l. Suppose Λ_l in (2.28) is positive, and the Hessian $\nabla_{\boldsymbol{\theta}}^2 \mathcal{J}(\Omega_\varepsilon)$ is locally Lipschitz at $\varepsilon = 0$ in the neighborhood of radius $\frac{2\Lambda_l}{L}$ with Lipschitz constant $L > 0$. If $0 < \varepsilon < \frac{2\Lambda_l}{L}$, then initializing the newly added layer with $\varepsilon\Phi_l$ decreases the loss function, i.e., $\mathcal{J}(\Omega_\varepsilon) < \mathcal{J}(\Omega_0)$.*

Proof:. Terminating the Taylor expansion (2.12) at the second order term, we have

$$\mathcal{J}(\Omega_\varepsilon) = \mathcal{J}(\Omega_0) + \frac{\varepsilon^2}{2} {\Phi_l}^T \nabla_{\boldsymbol{\theta}}^2 \mathcal{J}(\Omega_\gamma) \Phi_l,$$

for some $0 \leq \gamma \leq \varepsilon$. It follows that

$$\begin{aligned} \mathcal{J}(\Omega_\varepsilon) = \mathcal{J}(\Omega_0) - \Lambda_l \varepsilon^2 + \frac{\varepsilon^2}{2} {\Phi_l}^T \left[\nabla_{\boldsymbol{\theta}}^2 \mathcal{J}(\Omega_\gamma) - \nabla_{\boldsymbol{\theta}}^2 \mathcal{J}(\Omega_0) \right] \Phi_l \\ \leq \mathcal{J}(\Omega_0) - \Lambda_l \varepsilon^2 + \frac{\gamma L \varepsilon^2}{2} < \mathcal{J}(\Omega_0). \end{aligned}$$

Remark 2.10. We have shown that if Λ_l, given by (2.28), is greater than 0, then adding a new layer at depth l guarantees a decrease in the loss for sufficiently small ε. The question is when $\Lambda_l > 0$? Corollary 2.8 shows that Λ_l is half of the negative of the smallest eigenvalue value of the Hessian $\nabla_{\boldsymbol{\theta}}^2 \mathcal{J}(\Omega_\varepsilon) \Big|_{\varepsilon=0}$. Thus, the smallest eigenvalue needs to be negative. In that case, initializing the newly added layer with $\varepsilon\Phi_l$ corresponds to moving along a negative curvature direction, and thus decreasing the loss. If the smallest eigenvalue is non-negative, and thus Λ_L is non-positive, we do not add a new layer at the depth l, as it would not improve the loss function.

Remark 2.11. Note that if the geometric multiplicity of Λ_l (the maximum eigenvalue in (2.28)) is greater than 1, then one may choose Φ_l (eigenvector in (2.28)) corresponding to any eigenvalue as initialization for the new layer since all these initialization gives the same topological derivative.

Our algorithm starts by training a small network for some E_e epochs. Then, we compute the quantities in (2.28) for each layer l. At the l^* location where the topological derivative is highest and positive, we insert a new layer there. That is, we add a new layer at the depth at which the loss function is most topologically sensitive. Our formulation (2.29) also suggests that the initial value of the parameters of the newly added layer should be $\varepsilon\Phi_l$. The new network Ω_ε is trained again for E_e epochs and the procedure is repeated until a suitable termination criteria is satisfied (see Algorithm 2.1). The procedure in the case of a fully-connected network is provided in Algorithm 2.1.

2.5. Application to fully-connected neural network. In this section, we derive the expression for the topological derivative $d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma))$ for a fully connected network. The Hamiltonian H_l for the l^{th} layer is written as (refer equation (2.2)):

$$H_l\left(\mathbf{x}_{s,l}; \mathbf{p}_{s,l}; \mathbf{W}_{l+1}; \mathbf{b}_{l+1}\right) = \mathbf{p}_{s,l} \cdot \left[\mathbf{x}_{s,l} + \sigma(\mathbf{W}_{l+1}^T \mathbf{x}_{s,l} + \mathbf{b}_{l+1})\right], \tag{2.30}$$

where, $\mathbf{W}_{l+1}$ and $\mathbf{b}_{l+1}$ denote the weights and biases. Further, without loss of generality let's assume $\mathbf{b}_{l+1} = \mathbf{0}$ for the hidden layers and vectorized parameters (weights) as $\boldsymbol{\theta}_{l+1} = \left[\mathbf{W}_{l+1}^{11}, \dots \mathbf{W}_{l+1}^{n1}, \mathbf{W}_{l+1}^{12}, \dots \mathbf{W}_{l+1}^{n2}, \ \mathbf{W}_{l+1}^{1N_h}, \dots \mathbf{W}_{l+1}^{nn}\right]$. Then, the matrix $\mathbf{Q}_l$ in (2.29) can be explicitly written as

$$\mathbf{Q}_l = \frac{1}{2}\begin{bmatrix} \sum_{s=1}^{S} \mathbf{p}_{s,l}^{(1)} \mathbf{x}_{s,l}\mathbf{x}_{s,l}^T \sigma''(0) & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{0} & \sum_{s=1}^{S} \mathbf{p}_{s,l}^{(2)} \mathbf{x}_{s,l}\mathbf{x}_{s,l}^T \sigma''(0) & \mathbf{0} & \dots \\ \vdots & & \ddots & \\ \mathbf{0} & \dots & \mathbf{0} & \sum_{s=1}^{S} \mathbf{p}_{s,l}^{(n)} \mathbf{x}_{s,l}\mathbf{x}_{s,l}^T \sigma''(0) \end{bmatrix}, \tag{2.31}$$

where $\mathbf{p}_{s,l+1}^{(r)}$ denotes the r^{th} component of the vector. Therefore, in the case of a fully connected network, $\mathbf{Q}_l$ turns out to be a block-diagonal matrix. This structure can be further exploited to devise a strategy to choose the best subset of neurons in the new layer to activate in subsection 2.5.1.

Remark 2.12 (Assumptions on activation functions and their practical implications). Note that Item 1 in Remark 2.4 gives conditions to be satisfied by the activation function $\sigma(x)$ in our framework. (2.31) shows that $\sigma''(0) \neq 0$ is an additional necessary condition to be satisfied for ensuring that $\mathbf{Q}_l \neq \mathbf{0}$. There are several choices for $\sigma(x)$ that satisfies this condition. In particular, the choice of constructed activation functions mentioned in Item 2 of Remark 2.4 automatically satisfies this constraint. In subsection 5.2.1, Table 1, we conducted a neural architecture search (NAS) experiment comparing the performance of our newly designed activation function (linear combination of Swish and tanh) with popular activation functions such as ReLU and tanh. Our results summarized in Table 1 shows that our activation function performs on par with, and in some cases slightly better than, ReLU and tanh activations.

Further, when applying our approach to fine-tuning or transfer learning, as demonstrated in subsection 5.3, it is sufficient for the activation function in the newly added layer to satisfy the above constraints. Additionally, the new layer should include residual connections, while no constraints are imposed on the pretrained network. This demonstrates the broader utility of the technique beyond neural architecture adaptation.

Remark 2.13 (Computational cost associated with solving the eigenvalue problem (2.29)). Let $\mathbf{Q}_l^{(i)}$ be the i^{th} block (along the diagonal) of $\mathbf{Q}_l$ given by (2.31). Note that the eigenvalues of $\mathbf{Q}_l$ are just the list of eigenvalues of each $\mathbf{Q}_l^{(i)}$. [7] Therefore, one only needs to solve the eigenvalue problem for each block $\mathbf{Q}_l^{(i)}$ separately thereby significantly reducing the computational cost. In particular, for a network with k hidden layers of width n, using the QR algorithm [58] for eigenvalue decomposition results in kn embarrasingly parallel $\mathcal{O}(n^3)$ floating-point operations. Furthermore, since only the largest eigenvalue of $\mathbf{Q}_l^{(i)}$ is of interest, specialized methods (such as power method [58]) may be employed that require fewer floating-point operations than the full QR algorithm.

An important consequence of the block-diagonal structure of $\mathbf{Q}_l$ is that it enables an extension of our algorithm that activates only the most sensitive neurons in a newly added layer, rather than all n neurons. The details of this extension are discused in subsection 2.5.1.

2.5.1. Activating the most sensitive neurons in a newly added layer. Let $\mathbf{Q}_l^{(r)} := \sum_{s=1}^{S} \mathbf{p}_{s,l}^{(r)} \mathbf{x}_{s,l} \mathbf{x}_{s,l}^T \sigma''(0) \in \mathbb{R}^{n\times n}$ be the rth diagonal block in $\mathbf{Q}_l$ corresponding to the rth neuron. The following result is obvious.

Proposition 2.14. *Let $\boldsymbol{v}$ be a vector in $\mathbb{R}^n$. Define*

$$\Phi_r := [\underbrace{0, \ \dots}_{(r-1)\times n.}, \boldsymbol{v}, \ 0 \dots 0]^T. \tag{2.32}$$

Then $\boldsymbol{v}$ is an eigenvector of $\mathbf{Q}_l^{(r)}$ if and only if Φ_r is an eigenvector of $\mathbf{Q}_l$. Furthermore, the eigenvalues corresponding to $\boldsymbol{v}$ and Φ_r are the same.

As a direct consequence of Proposition 2.14, if $\boldsymbol{v}$ is an eigenvector corresponding to the largest eigenvalue (among all eigenvalues of all diagonal blocks), then the corresponding Φ_r is associated with the largest eigenvalue of $\mathbf{Q}_l$.

As a result, the shape functional is most sensitive to the rth neuron, and the rth neuron should be activated (i.e. initialized with non-zero weights/biases). Alternatively, given an integer m, we can identify "the best" m neurons such that when they are activated, the induced change in the shape functional is largest.

Proposition 2.15. *Consider the following constrained optimization problem:*

$$\max_{\boldsymbol{\phi}_i, \boldsymbol{\phi}_j \in \{\Phi_k\}_{k=1}^n, \boldsymbol{\phi}_i \neq \boldsymbol{\phi}_j} \left(d\mathcal{J}(\Omega_0; \ (l, \ (\boldsymbol{\phi}_1 + \dots \boldsymbol{\phi}_m), \ \sigma))\right), \quad s.t \, \|\boldsymbol{\phi}_i\|_2^2 = 1, \, i = 1, \dots, m. \tag{2.33}$$

[7] This follows from the fact that the characteristic polynomial of $\mathbf{Q}_l$ is the product of the characteristic polynomials of all $\mathbf{Q}_l^{(i)}$ [21]

Then, the optimal value is the sum of the first m largest eigenvalues of m diagonal sub-blocks of $\mathbf{Q}_l$. The optimal solutions $\{\boldsymbol{\phi}_i\}_{i=1}^m$ are the corresponding eigenvectors of these diagonal sub-blocks written in the form given by (2.32).

Proof:. The first-order optimality condition can be obtained by first formulating the Lagrangian as:

$$\begin{aligned} L\left(\boldsymbol{\phi}_1;\ldots,\boldsymbol{\phi}_m;\lambda_1;\ldots\lambda_m\right) &= d\mathcal{J}(\Omega_0;\ (l,\ \varphi,\ \sigma)) + \sum_{i=1}^{m}\lambda_i(1-\|\boldsymbol{\phi}_i\|_2^2) \\ &= \varphi^T\mathbf{Q}_l\varphi + \sum_{i=1}^{m}\lambda_i(1-\|\boldsymbol{\phi}_i\|_2^2), \end{aligned} \tag{2.34}$$

where $\varphi = \boldsymbol{\phi}_1 + \ldots \boldsymbol{\phi}_m$. Now notice that using the constraint $\boldsymbol{\phi}_i \neq \boldsymbol{\phi}_j$ in (2.33) and $\boldsymbol{\phi}_i, \boldsymbol{\phi}_j \in \{\Phi_k\}_{k=1}^n$, the Lagrangian can be rewritten in terms of the new variables $\{\hat{\boldsymbol{\phi}}_1, \ldots \hat{\boldsymbol{\phi}}_m\}$ as:

$$L\left(\hat{\boldsymbol{\phi}}_1;\ldots;\hat{\boldsymbol{\phi}}_m;\lambda_1;\ldots\lambda_m\right) = \sum_{i=1}^{m}\left[\hat{\boldsymbol{\phi}}_i\right]^T\mathbf{Q}_l^{\iota(i)}\hat{\boldsymbol{\phi}}_i + \sum_{i=1}^{m}\lambda_i\left(1-\left\|\hat{\boldsymbol{\phi}}_i\right\|_2^2\right),$$

where $\iota(\cdot) : \{1,\ldots,m\} \ni i \mapsto \iota(i) \in \{1,\ldots,n\}$ is the mapping from the subscript of $\hat{\boldsymbol{\phi}}_i$ to the diagonal block number $\iota(i)$ in $\mathbf{Q}_l$, and λ_i, $i = 1,\ldots,m$ are the Lagrangian multiplier. Using the Lagrangian, the first-order optimality condition of (2.33) reads

$$\mathbf{Q}_l^{\iota(i)}\hat{\boldsymbol{\phi}}_i = \lambda_i\hat{\boldsymbol{\phi}}_i, \quad \forall i,$$

that is, the pair $\left\{\lambda_i, \hat{\boldsymbol{\phi}}_i\right\}$ is an eigenpair of the diagonal block $\iota(i)$ of $\mathbf{Q}_l$.

As a result, the optimal objective function in (2.33) is the sum of the first m largest eigenvalues of m diagonal sub-blocks of $\mathbf{Q}_l$. The optimal solutions $\{\boldsymbol{\phi}_i\}_{i=1}^m$ are the corresponding eigenvectors of these diagonal sub-blocks written in the form (2.32). If we denote the m largest eigenvalues (from highest to lowest) as $\{\lambda_i\}_{i=1}^m$, the solution is written as:

$$\Lambda_l = \sum_{i=1}^{m}\lambda_i, \quad \Phi_l = \boldsymbol{\phi}_i + \ldots \boldsymbol{\phi}_m. \tag{2.35}$$

Based on the above-computed quantities, our algorithm for adapting the depth of a fully connected feed-forward architecture is given in Algorithm 2.1.

Remark 2.16. Note that our framework can also be applied to a convolutional neural network (CNN) architecture and the details are provided in Appendix A. We also provide a numerical demonstration in Appendix D.

3. Fully automated network growing Algorithm 3.1. Note that Algorithm 2.1 activates the "the best" m neurons while inserting a new layer as described in subsection 2.5.1. Note that m is a user-defined hyperparameter in Algorithm 2.1. In this section, we extend our algorithm to automatically select the parameter m, i.e. automatically determine the number of neurons (width) of the added hidden layer. To that end let us revisit Proposition 2.15

Algorithm 2.1 Fully connected architecture adaptation algorithm for a regression task

Input: Training data $\mathbf{X}$, labels $\mathbf{C}$, validation data $\mathbf{X}_1$, validation labels $\mathbf{C}_1$, number of neurons in each hidden layer n, loss function Φ, number of neurons to activate in each hidden layer m, number of iterations N_n, parameter ε, ε^t, parameter T_b, hyperparameters and predefined scheduler for optimizer (Appendix E).

Initialize: Initialize network $\mathcal{Q}_1$ with T_b hidden layers.

Train network $\mathcal{Q}_1$ and store the validation loss $(\varepsilon_v)^1$.
set $i = 1$, $(\varepsilon_v)^0 >> (\varepsilon_v)^1$, $\Lambda_l^m \geq \varepsilon^t$
while $i \leq N_n$ **and** $\left[(\varepsilon_v)^i \leq (\varepsilon_v)^{i-1}\right]$ **and** $\Lambda_l^m \geq \varepsilon^t$ **do**
    Compute the topological derivative for each layer l using (2.35) and store as $\{\Lambda_l\}$, also store $\Lambda_l^m = \max_l\{\Lambda_l\}$
    Store the corresponding eigenvectors for each layer as Φ_l given by (2.35).
    Obtain the new network $\mathcal{Q}_{i+1}$ by adding a new layer at position $l^* = \underset{l}{\text{argmax}}\{\Lambda_l\}$ with initialization $\varepsilon\Phi_{l^*}$.
    Perform a backtracking line search to update ε as outlined in Algorithm C.1.
    Update the parameters for optimizer if required (refer Appendix E for scheduler details).
    Train network $\mathcal{Q}_{i+1}$ and store the best validation loss $(\varepsilon_v)^{i+1}$ and the best network $\mathcal{Q}_{i+1}$.
    $i = i + 1$
end while

Output: Network $\mathcal{Q}_{i-1}$

where m is now unknown. The idea is to look at how the optimal loss in (2.33) changes as one increases m. In particular, for each layer l we choose the largest value of m such that:

$$\frac{\lambda_1 - \lambda_m}{\lambda_1} \leq \varepsilon_s, \quad s.t \;\; \lambda_1,\ \lambda_m > 0, \tag{3.1}$$

where λ_i is defined in (2.35) and ε_s is a user-defined hyperparameter. Since now m could be different for each layer l, let us denote it as $m(l)$. Criterion (3.1) simply compares the sensitivity of each neuron relative to the most sensitive neuron in the added layer. If $\varepsilon_s = 0$, then it is clear that only the most sensitive neuron gets activated. On the other hand, when $\varepsilon_s = 1$ all the neurons (with $\lambda_m > 0$) in the hidden layer gets activated. In practice, we choose $\varepsilon_s = 0.5$ such that all neurons which are at least 50 % as sensitive as the most sensitive neuron gets activated in the added layer. Further, note that the derivative Λ_l in (2.35) is computed with respect to the operation of activating exactly m neurons in an added layer. However, since in this case $m = m(l)$ can be different for each layer l, one needs to normalize the derivative in (2.35) as $\Lambda_l = \frac{\Lambda_l}{m(l)}$ while making decision on where to add a new layer. The full algorithm is provided in Algorithm 3.1.

Further, in the fully automated network growing (Proposed II), we do not use a predefined scheduler (such as in Algorithm 2.1) to decide when a new layer needs to be added during

the training process. Instead, a validation data set is employed to decide when a new layer needs to be added. The key difference of fully-automated growing from Algorithm 2.1 is the training loop without using a predefined scheduler (lines 9-17 in Algorithm 3.1). Note that one stops training the current network $\mathcal{Q}_{i+1}$ if the validation loss does not decrease for N_k consecutive training epochs. Then, a new layer is added. The extension of Algorithm 3.1 to the case of a CNN architecture is trivial (in this case the parameter m corresponds to the number of channels in the hidden layer).

Algorithm 3.1 Fully automated network growing algorithm for fully connected architecture

Input: Training data $\mathbf{X}$, labels $\mathbf{C}$, validation data $\mathbf{X}_1$, validation labels $\mathbf{C}_1$, number of neurons in each hidden layer n, loss function Φ, number of iterations N_n, parameter ε, ε^t, parameter T_b, N_k, hyperparameters for optimizer (Appendix E).

Initialize: Initialize network $\mathcal{Q}_1$ with T_b hidden layers.

1: Train network $\mathcal{Q}_1$ and store the best validation loss $(\varepsilon_v)^1$.
2: set $i = 1$, $(\varepsilon_v)^0 >> (\varepsilon_v)^1$, $\Lambda_l^m \geq \varepsilon^t$
3: **while** $i \leq N_n$ **and** $\left[(\varepsilon_v)^i \leq (\varepsilon_v)^{i-1}\right]$ **and** $\Lambda_l^m \geq \varepsilon^t$ **do**
4: Compute $m(l)$ for each layer l using (3.1) as outlined in section 3.
5: Compute the derivative for each layer l using (2.35) and store as $\{\frac{\Lambda_l}{m(l)}\}$, also store $\Lambda_l^m = \max_l\{\frac{\Lambda_l}{m(l)}\}$
6: Store the corresponding eigenvectors for each layer as Φ_l given by (2.35).
7: Obtain the new network $\mathcal{Q}_{i+1}$ by adding a new layer at position $l^* = \underset{l}{\operatorname{argmax}}\{\frac{\Lambda_l}{m(l)}\}$ with initialization $\varepsilon\Phi_{l^*}$.
8: Perform a backtracking line search to update ε as outlined in Algorithm C.1.
9: Set epoch $j = 1$, $(\varepsilon_v)_j^{i+1}$ as the validation loss for $\mathcal{Q}_{i+1}$, and set $k = 1$.
10: **while** $k \leq N_k$ **do** ▷ Training loop without using a scheduler
11: Update parameters of network $\mathcal{Q}_{i+1}$ and store the current best validation loss $(\varepsilon_v)_{j+1}^{i+1}$.
12: **if** $(\varepsilon_v)_{j+1}^{i+1} \geq (\varepsilon_v)_j^{i+1}$ **then**
13: k=k+1
14: **else**
15: set $k = 1$
16: **end if**
17: $j = j + 1$
18: **end while**
19: Store the best validation loss for $\mathcal{Q}_{i+1}$ as $(\varepsilon_v)^{i+1}$=$(\varepsilon_v)_{j+1}^{i+1}$ and the corresponding best network $\mathcal{Q}_{i+1}$.
20: $i = i + 1$
21: **end while**

Output: Network $\mathcal{Q}_{i-1}$

4. Topological derivative approach through the lens of an optimal transport problem. In this section, we show that our layer insertion strategy can be derived as a solution to maximizing a topological derivative in p-Wasserstein space, where $p \geq 1$. Optimization in $\infty-$Wasserstein space was earlier considered by Liu et al. [63] to explain their neuron splitting strategy for growing neural networks along the width. The standard optimal transport (Kantorovich formulation [50]) aims at computing a minimal cost transportation plan γ between a source probability measure μ, and target probability measure ν. A common formulation of the cost function is based on the Wasserstein distance $W_p(\mu;\ \nu)$.

Definition 4.1 (p-Wasserstein metric). *The p-Wasserstein distance between two probability measures μ and ν is given as:*

$$W_p\left(\mu;\ \nu\right) = \inf_{\gamma \in \Gamma(\mu;\ \nu)} \left(\mathop{\mathbb{E}}_{(\boldsymbol{\phi},\ \boldsymbol{\phi}') \sim \gamma} \left\| \boldsymbol{\phi} - \boldsymbol{\phi}' \right\|_2^p \right)^{1/p}, \tag{4.1}$$

where $\Gamma(\mu;\ \nu)$ is the set of couplings of μ and ν. A coupling γ is a joint probability measure whose marginals are μ and ν on the first and second factors, respectively.

In this work, *we set up an optimization problem to determine the unknown target measure ν given the source measure μ.* Let us consider the scenario depicted in Figure 2 in which we add a message-passing layer with zero weights/biases, and then ask the question: what is the initialization values $\overline{T}(\mathbf{0})$ (where $\overline{T}$ is the transport map) for the weights/biases of the newly added layer such that the loss function is most sensitive too? In this section, we answer this question from an optimal transport point of view. For the clarity of the exposition (and without loss of generality), let us assume that the message-passing layer (new layer) is added in front of the output layer of Ω_0 as depicted in Figure 2.

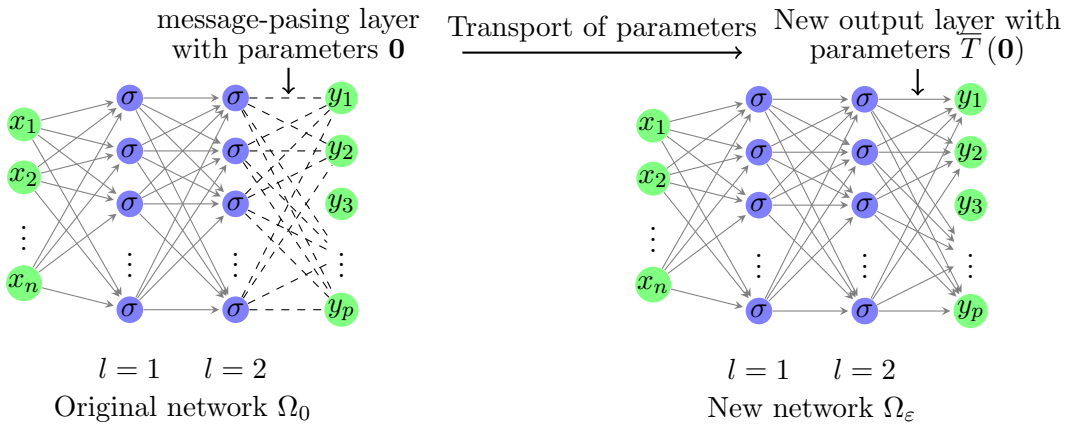


Figure 2. *Optimal transport interpretation of our proposed approach: We wish to optimally transport (in some sense) the parameters from network Ω_0 (left figure) to the new network Ω_ε (right figure).*

For the message-passing layer with zero weights/biases, the functional representation of the loss (2.9) can be written as:

$$\mathcal{J}(\mu_{\mathbf{0}}) = \frac{1}{S}\sum_{s=1}^{S}\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\Omega_{\varepsilon}\left(\mathbf{x}_{s,0};\ \boldsymbol{\phi}\right)\right]\right), \qquad \mu_{\mathbf{0}} = \delta_0 \times \delta_0 \times \ldots (n_p\ times) \cdots \times \delta_0, \tag{4.2}$$

where the second argument in $\Omega_{\varepsilon}\left(\mathbf{x}_{s,0};\ \boldsymbol{\phi}\right)$ denotes the parameters $\boldsymbol{\phi}$ of the added layer, δ_0 denotes the Dirac measure on $\mathbb{R}$, n_p denotes the total number of parameters in the added layer, $\mu_{\mathbf{0}}$ is the product measure. Similarly, for some non-zero initialization values $\varepsilon\boldsymbol{\phi}$ (which we seek to determine) for the weights/biases of the newly added layer, the functional representation of the loss (2.9) can be written as:

$$\mathcal{J}(\nu_{\varepsilon\boldsymbol{\phi}}) = \frac{1}{S}\sum_{s=1}^{S}\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\nu_{\varepsilon\boldsymbol{\phi}}}\left[\Omega_{\varepsilon}\left(\mathbf{x}_{s,0};\ \boldsymbol{\phi}\right)\right]\right), \qquad \nu_{\varepsilon\boldsymbol{\phi}} = \delta_{\varepsilon\boldsymbol{\phi}_1} \times \delta_{\varepsilon\boldsymbol{\phi}_2} \times \cdots \times \delta_{\varepsilon\boldsymbol{\phi}_{n_p}}, \tag{4.3}$$

Theorem 4.2 (Topological derivative in $p-$Wasserstein space). *Let $\mu_{\mathbf{0}}$ be a given probability measure (4.2) and assume that conditions Item 1 and, Item 2 in Proposition 2.3 are satisfied for the neural network. Let $\nu_{\varepsilon\boldsymbol{\phi}}$, be another (unknown) perturbed measure (4.3) such that $W_p\left(\mu_{\mathbf{0}};\ \nu_{\varepsilon\boldsymbol{\phi}}\right) \leq \varepsilon$. Further, assume that the conditions 1, 2 of Theorem 2.7 are satisfied. Then, the following results holds:*

1. *The probability measure $\nu_{\varepsilon\boldsymbol{\phi}}$ has the form $\nu_{\varepsilon\boldsymbol{\phi}} = \overline{T}_{\#}\mu$ where the transport map $\overline{T}$ satisfies:*

$$\overline{T}(\mathbf{0}) = \varepsilon\boldsymbol{\phi}, \quad \text{where } \|\boldsymbol{\phi}\|_2 \leq 1, \tag{4.4}$$

2. *With $W_p\left(\mu_{\mathbf{0}};\ \nu_{\varepsilon\boldsymbol{\phi}}\right) \leq \varepsilon$, the Topological derivative in $p-$Wasserstein space at $\mu_{\mathbf{0}}$ along the direction $\boldsymbol{\phi}$ is given by:*

$$d\mathcal{J}(\mu_{\mathbf{0}};\ \boldsymbol{\phi}) = -\lim_{\varepsilon\downarrow 0}\frac{\mathcal{J}(\nu_{\varepsilon\boldsymbol{\phi}}) - \mathcal{J}(\mu_{\mathbf{0}})}{\varepsilon^2} = \frac{\boldsymbol{\phi}^T Q(\mathbf{0})\boldsymbol{\phi}}{2}, \quad \|\boldsymbol{\phi}\|_2 \leq 1, \tag{4.5}$$

where

$$Q(\boldsymbol{\phi}) = \sum_{s=1}^{S}\nabla^2_{\boldsymbol{\phi}} H_2\left(\mathbf{x}_{s,2};\ \mathbf{p}_{s,2};\ \boldsymbol{\phi}\right),$$

and H_2 is the Hamiltonian given by (2.2) corresponding to the output layer.

3. *Consider the problem of maximizing the Topological derivative in $p-$ Wasserstein space:*

$$\max_{\boldsymbol{\phi}}\left(d\mathcal{J}(\mu_{\mathbf{0}};\ \boldsymbol{\phi})\right), \quad \textit{subject to: } \|\boldsymbol{\phi}\|_2 \leq 1.$$

Then the optimal transport map in (4.4) satisfies the following:

$$\overline{T}(\mathbf{0}) = \varepsilon\ v_{max}(\mathbf{0}),$$

where $v_{max}(\mathbf{0})$ is the eigenvector of $Q(\mathbf{0})$ corresponding to the maximum eigenvalue.

Proof:. *a*) The first part of the proof follows from the fact that for Dirac measures $\mu_{\mathbf{0}},\ \nu_{\varepsilon\boldsymbol{\phi}}$, $p-$Wasserstein distance (4.1) simplifies as:

$$W_p\left(\mu_{\mathbf{0}};\ \nu_{\varepsilon\boldsymbol{\phi}}\right) = \left\|\overline{T}(\mathbf{0}) - \mathbf{0}\right\|_2.$$

Therefore,

$$W_p\left(\mu_{\mathbf{0}};\ \nu_{\varepsilon\boldsymbol{\phi}}\right) \leq \varepsilon \implies \left\|\overline{T}(\mathbf{0}) - \mathbf{0}\right\|_2 \leq \varepsilon \implies \overline{T}(\mathbf{0}) = \varepsilon\boldsymbol{\phi}, \quad \|\boldsymbol{\phi}\|_2 \leq 1.$$

b) To derive the Topological derivative in $p-$Wasserstein space, we look at the Taylor series expansion of the functional $\mathcal{J}$ about the measure $\mu_{\mathbf{0}}$ as follows:

$$\mathcal{J}(\nu_{\varepsilon\boldsymbol{\phi}}) - \mathcal{J}(\mu_{\mathbf{0}}) = \mathbf{D}J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] + \frac{1}{2}\mathbf{D}^2J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] + \frac{1}{6}\left[\frac{d^3}{d\eta^3}\mathcal{J}(\mu_\eta)\right]_{\eta=\zeta}, \tag{4.6}$$

where ζ is a number between 0 and 1, $\mathbf{D}^kJ(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}]$ denotes the k^{th} variation of the functional $J(\mu_{\mathbf{0}})$ computed based on the displacement interpolation [50, 11, 63] given by:

$$\mathbf{D}^kJ(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] = \left[\frac{d^k}{d\eta^k}\mathcal{J}(\mu_\eta)\right]_{\eta=0}, \quad \mu_\eta = (\pi_\eta)_{\#}\mu_{\mathbf{0}},$$

where $\pi_\eta = (1-\eta)\boldsymbol{\phi} + \eta\overline{T}(\boldsymbol{\phi})$ with $\eta = [0,\ 1]$ (note that $\mu_0 = \mu_{\mathbf{0}}$ and $\mu_1 = \nu_{\varepsilon\boldsymbol{\phi}}$). Analyzing the first-order term in (4.6) yields:

$$\begin{aligned}
\mathbf{D}J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] &= \frac{d}{d\eta}\left(\frac{1}{S}\sum_{s=1}^{S}\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}_\eta\sim\mu_\eta}\left[\Omega_\varepsilon\left(\mathbf{x}_{s,0};\ \boldsymbol{\phi}_\eta\right)\right]\right)\right)\Bigg|_{\eta=0} \\
&= \frac{d}{d\eta}\left(\frac{1}{S}\sum_{s=1}^{S}\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\Omega_\varepsilon\left(\mathbf{x}_{s,0};\ \eta\overline{T}(\boldsymbol{\phi}) + (1-\eta)\boldsymbol{\phi}\right)\right]\right)\right)\Bigg|_{\eta=0} \\
&= \frac{1}{S}\sum_{s=1}^{S}\left[\nabla\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})\right]\right)\cdot\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left(\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})\right)\left(\overline{T}(\boldsymbol{\phi}) - \boldsymbol{\phi}\right)\right]\right],
\end{aligned} \tag{4.7}$$

where $\nabla_{\boldsymbol{\phi}}$ denotes the gradient w.r.t the second argument of Ω_ε. Further note that the second term in the last line of (4.7) can be simplified as:

$$\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left(\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})\right)\left(\overline{T}(\boldsymbol{\phi}) - \boldsymbol{\phi}\right)\right] = \nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \mathbf{0})\overline{T}(\mathbf{0}) - (\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \mathbf{0}))\mathbf{0} = \mathbf{0},$$

where we have used the property of the Dirac-measure as defined in (4.2) and also used $\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \mathbf{0}) = \mathbf{0}$ due to to assumption (Item 2) in Theorem 4.2. Proceeding similarly for

computing the second-order term $\mathbf{D}^2 J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}]$ in (4.6), we have:

(4.8)

$$\mathbf{D}^2 J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] = \frac{d}{d\eta}\left[\frac{1}{S}\sum_{i=1}^{S}\left[\nabla\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\Omega_\varepsilon\left(\mathbf{x}_{s,0};\ \pi_\eta\right)\right]\right)\cdot\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left(\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \pi_\eta)\right)g(\boldsymbol{\phi})\right]\right]\right]\Bigg|_{\eta=0} =$$

$$\frac{1}{S}\sum_{s=1}^{S}\left[\left(\nabla^2\Phi\left[\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})\right]\right]\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left(\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})\right)\left(g(\boldsymbol{\phi})\right)\right]\right)\cdot\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})g(\boldsymbol{\phi})\right]\right]$$

$$+\sum_{s=1}^{S}\left[\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[g(\boldsymbol{\phi})^T\ \nabla^2_{\boldsymbol{\phi}}\ \left(-\mathbf{p}_{s,T}\cdot\Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi})\right)\ g(\boldsymbol{\phi})\right]\right],$$

where $g(\boldsymbol{\phi}) = \overline{T}(\boldsymbol{\phi}) - \boldsymbol{\phi}$ and we have used (2.3) in the second term of (4.8) while applying the chain rule, i.e,

$$\mathbf{p}_{s,T} = -\frac{1}{S}\nabla\Phi\left(\mathbf{x}_{s,T}\right) = -\frac{1}{S}\nabla\Phi\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\Omega_\varepsilon\left(\mathbf{x}_{s,0};\ \boldsymbol{\phi}\right)\right]\right).$$

Now note that the first term in (4.8) vanishes due to the property of the Dirac-measure as defined in (4.2) and also used $\nabla_{\boldsymbol{\phi}}\ \Omega_\varepsilon(\mathbf{x}_{s,0};\ \mathbf{0}) = \mathbf{0}$ due to assumptions (Item 2) in Theorem 4.2. Further, note that from (2.2) we have:

$$H_2\left(\mathbf{x}_{s,2};\ \mathbf{p}_{s,2};\ \boldsymbol{\phi}\right) = \mathbf{p}_{s,T}\cdot\Omega_\varepsilon(\mathbf{x}_{s,0};\ \boldsymbol{\phi}) = \mathbf{p}_{s,T}\cdot\mathbf{f}_3\left(\mathbf{x}_{s,2};\ \boldsymbol{\phi}\right).$$

where we have used $\mathbf{p}_{s,2} = \mathbf{p}_{s,3} = \mathbf{p}_{s,T}$ since the added layer with zero weights and biases acts as a message-passing layer due to Proposition 2.3. Using the above result and the definition of $Q(\boldsymbol{\phi})$, (4.8) can be simplified as:

$$\mathbf{D}^2 J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] = -\sum_{s=1}^{S}\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left(\overline{T}(\boldsymbol{\phi})-\boldsymbol{\phi}\right)^T\ \nabla^2_{\boldsymbol{\phi}}H_2\left(\mathbf{x}_{s,2};\mathbf{p}_{s,2};\boldsymbol{\phi}\right)\left(\overline{T}(\boldsymbol{\phi})-\boldsymbol{\phi}\right)\right]$$

$$= -\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left(\overline{T}(\boldsymbol{\phi})-\boldsymbol{\phi}\right)^T\ Q(\boldsymbol{\phi})\ \left(\overline{T}(\boldsymbol{\phi})-\boldsymbol{\phi}\right)\right] = -\overline{T}(\mathbf{0})^T\ Q(\mathbf{0})\ \overline{T}(\mathbf{0}),$$

where we used the property of Dirac measure in the last line. Now using (4.4), the expression for the second variation above can be further simplified as:

$$\mathbf{D}^2 J(\mu_{\mathbf{0}})[\nu_{\varepsilon\boldsymbol{\phi}}] = -\varepsilon^2\boldsymbol{\phi}^T Q(\mathbf{0})\boldsymbol{\phi}, \quad \text{where } \|\boldsymbol{\phi}\|_2 \leq 1. \tag{4.9}$$

Proceeding similarly for computing the third-order term in (4.6), it can be shown after some algebraic manipulations that [63]:

$$\left[\frac{d^3}{d\eta^3}\mathcal{J}(\mu_\eta)\right]_{\eta=\zeta} = \mathcal{O}\left(\mathop{\mathbb{E}}_{\boldsymbol{\phi}\sim\mu_{\mathbf{0}}}\left[\left\|\boldsymbol{\phi}-\overline{T}(\boldsymbol{\phi})\right\|_2^3\right]\right) = \mathcal{O}\left(\left\|\mathbf{0}-\overline{T}(\mathbf{0})\right\|_2^3\right) = \mathcal{O}\left(\varepsilon^3\right), \tag{4.10}$$

where we have used the assumptions in Theorem 2.7 while simplifying and used the form of transport map (4.4). Substituting results (4.7), (4.9), (4.10) in (4.6) we have:

$$\mathcal{J}(\nu_{\varepsilon\boldsymbol{\phi}}) - \mathcal{J}(\mu_{\mathbf{0}}) = -\frac{\varepsilon^2\boldsymbol{\phi}^T Q(\mathbf{0})\boldsymbol{\phi}}{2} + \frac{1}{6}\mathcal{O}\left(\varepsilon^3\right), \quad \text{where } \|\boldsymbol{\phi}\|_2 \leq 1.$$

Therefore, the Topological derivative in $p-$Wasserstein space at $\mu_{\mathbf{0}}$ along the direction $\boldsymbol{\phi}$ is given by:

$$d\mathcal{J}(\mu_{\mathbf{0}};\ \boldsymbol{\phi}) = -\lim_{\varepsilon\downarrow 0}\frac{\mathcal{J}(\nu_{\varepsilon\boldsymbol{\phi}})-\mathcal{J}(\mu_{\mathbf{0}})}{\varepsilon^2} = \frac{\boldsymbol{\phi}^T Q(\mathbf{0})\boldsymbol{\phi}}{2}, \quad \text{where } \|\boldsymbol{\phi}\|_2 \leq 1.$$

c) Finally maximizing the Topological derivative (4.5) yields:

$$\max_{\boldsymbol{\phi}}\left(d\mathcal{J}(\mu_{\mathbf{0}};\ \boldsymbol{\phi})\right) = \max_{\boldsymbol{\phi}}\left(\frac{\boldsymbol{\phi}^T Q(\mathbf{0})\boldsymbol{\phi}}{2}\right), \quad \text{subject to } \|\boldsymbol{\phi}\|_2 \leq 1. \tag{4.11}$$

The optimal solution $\boldsymbol{\phi}^*$ to (4.11) is clearly given by:

$$\boldsymbol{\phi}^* = v_{max}(\mathbf{0}),$$

where $v_{max}(\mathbf{0})$ is the eigenvector of $Q(\mathbf{0})$ corresponding to the maximum eigenvalue. Therefore, from (4.4) the optimal transport map satisfies:

$$\overline{T}(\mathbf{0}) = \varepsilon\boldsymbol{\phi}^* = \varepsilon v_{max}(\mathbf{0}),$$

thereby concluding the proof.

5. Numerical experiments. In this section, we numerically demonstrate the proposed approach using different types of architecture such as a) Radial basis function neural network; b) Fully connected neural network; and c) vision transformer (ViT). In all our numerical experiments, we choose a linear combination of $Swish$ and $tanh$ as our activation function; that is, we take $\mathcal{F} = \{Swish,\ tanh\}$ in (2.8). Our supplementary file contains additional numerical examples where we considered both simulated and real-world data sets and also demonstrated the approach for a convolutional neural network architecture (CNN). General experimental settings for all the problems and descriptions of methods adopted for comparison are detailed in Appendix B. Note that in our numerical results, Proposed (I) refers to Algorithm 2.1 and proposed (II) refers to "fully automated growing" algorithm mentioned in section 3.

5.1. Proof of concept example: Radial basis function (RBF) neural network. As a proof of concept of our proposed approach, we first consider the problem of learning a 1-dimensional function using a radial basis function (RBF) neural network. In particular, we consider the multi-layered RBF neural network [15, 8] with residual connections where $\mathbf{g}_{t+1}(.;\ .)$ in (2.5) is given by:

$$\mathbf{g}_{i+1}(x_{s,i};\ \boldsymbol{\theta}_{i+1}) = \boldsymbol{\theta}_{i+1}^{(3)} \times \exp\left(-\frac{1}{2}\left(\boldsymbol{\theta}_{i+1}^{(1)}x_{s,i} + \boldsymbol{\theta}_{i+1}^{(2)} - c\right)^2\right) - \boldsymbol{\theta}_{i+1}^{(3)} \times \exp\left(-\frac{1}{2}\left(c\right)^2\right), \tag{5.1}$$

where $\boldsymbol{\theta}_i = \left[\theta_i^{(1)},\ \theta_i^{(2)},\ \theta_i^{(3)}\right]^T \in \mathbb{R}^3$ for $i = 0, \ldots T-1$, and c is a non-zero parameter of the activation function. Note that, the second term in (5.1) is introduced to satisfy condition 2 of Proposition 2.3 and therefore in this case we have a modified radial basis function. Further, note that (5.1) corresponds to using a single neuron in each hidden layer.

5.1.1. Topological derivative for the modified RBF network (5.1). The topological derivative is computed based on solving the eigenvalue problem (2.29) where the matrix $\mathbf{Q}_l$ (equation (2.29)) is given as:

$$\mathbf{Q}_l = \frac{1}{2}\sum_{s=1}^{S}\left(\begin{bmatrix} 0 & 0 & c_1\ p_{s,l}\ x_{s,l} \\ 0 & 0 & c_1\ p_{s,l} \\ c_1\ p_{s,l}\ x_{s,l} & c_1\ p_{s,l} & 0 \end{bmatrix}\right), \tag{5.2}$$

where $c_1 = c\ \exp\left(-\frac{1}{2}(c)^2\right)$ and c is a parameter in (5.1).

5.1.2. Data generation and numerical results. For generating the training data set, we set $T = 15$ in (2.5) and sample a true set of parameters $\{\boldsymbol{\theta}_i\}_{i=1}^{T}$ from normal distribution $\mathcal{N}(\mathbf{0}, 3\mathbf{I})$, where $\mathbf{I}$ is the 3×3 identity matrix. Further, we set $c = 0.1$ in (5.1). The training data set $D = \{x_i,\ c_i\}_{i=1}^{5000}$ is then generated by drawing x_i uniformly from $[-2,\ 2]$ and computing the corresponding labels as $c_i = \Omega^*(x_i)$, where Ω^* denote the true map to be learnt. In addition, we consider an additional 500 data points for generating the validation data set and 1000 data points for the testing data set. The generated true function is shown in Figure 3 (rightmost figure).

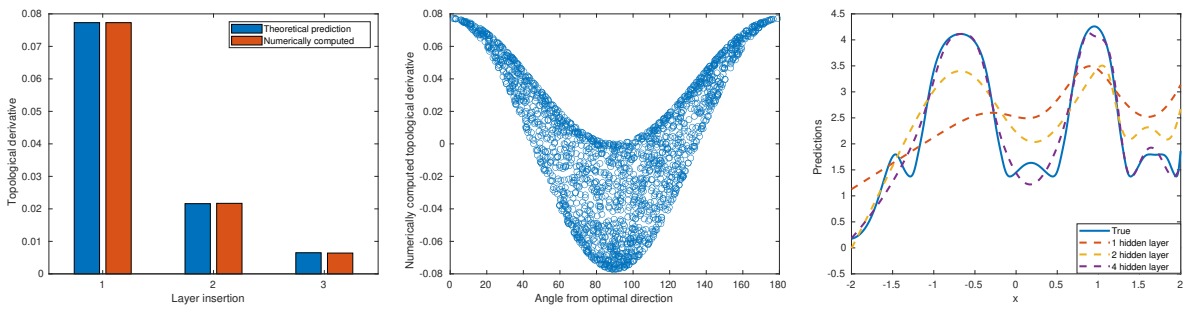
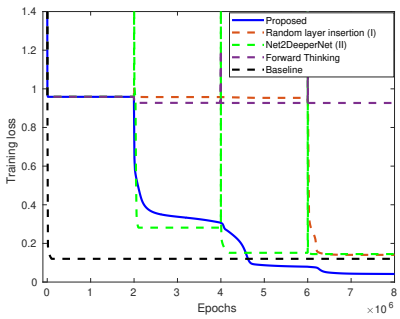


Figure 3. *Validating Theorem 2.7.* **Left subfigure***: comparison of the theoretically computed topological derivative (equation (2.13)) with the numerically computed derivative for layer 'l' with the largest eigenvalue and initialization Φ_l given by (2.28);* **Middle subfigure***: effect of initialization $\boldsymbol{\phi}$ on the numerically computed derivative $d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma))$ in (2.11) for $l = 1$ and at the end of 1^{st} iteration;* **Right subfigure***: learned function using the proposed approach at different iterations of the algorithm.*

5.1.3. Discussion of results. As outlined in subsection 2.4, we begin the adaptive training process by starting with a one-hidden layer network and progressively adding new layers. Each layer is trained to a (approximately) local minimum before adding a new layer. To validate our proposed theory, we compare the topological derivative predicted by our Theorem 2.7 with that computed numerically at each iteration of the algorithm and the results are shown in Figure 3 (left). Note that the derivatives are computed for the layer l with the largest eigenvalue at each iteration and initialization $\boldsymbol{\phi} = \Phi_l$ given by (2.28). We use a step size of $\varepsilon = 1 \times 10^{-4}$ to compute the numerical derivative $d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma))$ in (2.11). Figure 3 (left) shows that the numerical derivative is in close agreement with the theoretical derivative. Figure 3 (middle) investigates how the choice of initialization $\boldsymbol{\phi}$ influences the derivative

$d\mathcal{J}(\Omega_0;\ (l,\ \boldsymbol{\phi},\ \sigma))$ in (2.11). For this, we set $l = 1$ (at the end of training the initial one hidden layer network) and generated different unit vectors (each sampled vector is represented as a blue circle in Figure 3 (middle)). The x-axis denotes the angle that each sampled vector makes with the optimal direction Φ_l given by (2.28) and the y-axis denotes the numerically computed derivative for each sampled $\boldsymbol{\phi}$. It is clear from Figure 3 (middle) that the maximum topological derivative is indeed observed for the optimal eigenvector Φ_l predicted by (2.28). Figure 3 (rightmost figure) shows the learned function (by our proposed approach) at each iteration of the algorithm.



Method	Test loss ($\mu \pm \sigma$)
Proposed	0.051 ± 0.027
Random layer insertion (I)	0.099 ± 0.0514
Net2DeeperNet (II) [12]	0.069 ± 0.0335
Baseline network	0.0897 ± 0.038
Forward Thinking [22]	0.23 ± 0.150

Figure 4. Left subfigure: *typical training loss curves for different approaches.* **Right subfigure**: *summary of results.*

Typical training curves for different adaptation strategies are shown in Figure 4 (left). The sharp dip in the loss in Figure 4 (left) corresponds to a decrease in loss on adding a new layer. It is clear from Figure 4 (left) that our proposed framework outperforms all other adaptation strategies while also exhibiting superior performance in comparison to a baseline network trained from scratch. It is also interesting to note that while Net2DeeperNet (II) seems to exhibit superior performance in the beginning stages in Figure 4 (left), it is clear that there is no guarantee that Net2DeeperNet (II) will escape saddle points as evident from the later stages of adaptation. In contrast, our layer initialization strategy ensures a decrease in loss for sufficiently small ε thereby escaping saddle points. The decision-making process at each iteration on where to add a new layer is shown in Figure 5 where the blocks represent the relative magnitude of the derivative for different layers. Note that a new layer is inserted at the location with maximum derivative. Further, the effectiveness of the approach is quantified by

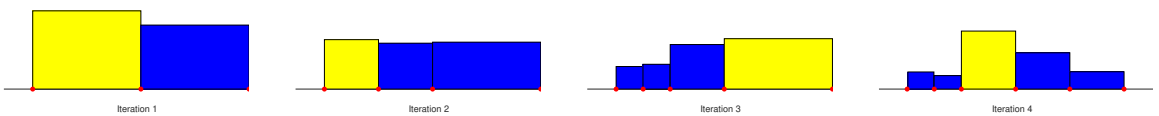


Figure 5. *Relative magnitude of the topological derivative for different hidden layers during the adaptation procedure. A new layer is inserted at the position of the maximal topological derivative (yellow block), and the red dots denote the location of layers.*

performing an uncertainty analysis where the algorithm is run considering different random

initializations for the initial single hidden layer network and the results are tabulated in Figure 4 (right). Figure 4 (right) shows that on average our proposed method outperformed all other adaptation strategies while also exhibiting lower uncertainty. Note that in contrast to other approaches, the only source of uncertainty in our approach is the random initialization of initial network $\mathcal{Q}_1$ in Algorithm 2.1. Further, the results in Figure 4 (right) can be further improved (lower testing loss) if one considers an RBF network with multiple neurons in each hidden layer. *It is also important to note that in practice it is not essential to train the network to a local minima at each iteration of the algorithm as demonstrated in Figure 4 (left) (note that the topological derivative in Theorem (2.7) is valid even when the current network Ω_0 is not in a local minima). This is a significant advantage over some other methods that require the training loss to plateau before growing the network [63, 28].* One may consider more efficient training approaches where the network is trained for a fixed user-defined number of epochs (scheduler) at each iteration as suggested in [19]. Our Algorithm 2.1 considers the use of a predefined scheduler to decide when a new layer needs to be added during the training phase, whereas Algorithm 3.1 employs the use of a validation metric to automatically detect when a new layer needs to be added.

In the following sections, we consider experiments with more complex datasets and assess the performance of both Algorithm 2.1 and Algorithm 3.1.

5.2. Experiments with fully connected neural network. In this section, we consider experiments with fully connected neural networks (see Algorithm 2.1).

5.2.1. Learning the observable to parameter map for 2D heat equation. In this section we consider solving the 2D heat conductivity inversion problem [43]. The heat equation we consider is the following:

$$\begin{aligned} -\nabla \cdot (e^u \nabla y) &= 20 \quad \text{in } \Omega = (0,1)^2, \\ y &= 0 \quad \text{on } \Gamma^{\text{ext}}, \\ \mathbf{n} \cdot (e^u \nabla y) &= 0 \quad \text{on } \Gamma^{\text{root}}, \end{aligned} \tag{5.3}$$

where the conductivity u is the parameter of interest (PoI), y is the temperature field, and $\mathbf{n}$ is the unit outward normal vector on Neumann boundary part Γ^{root}. The objective is to infer the parameter field $u(\mathbf{x})$ given the observation field $y(\mathbf{x})$. We construct the input vector for the neural network as $\mathbf{y} = [y(\mathbf{x}_1), \ \dots y(\mathbf{x}_{n_0})]$, where $\mathbf{x}_i$ are fixed locations on Ω. The domain Ω along with the fixed locations is provided in Figure 10. For the present experiment, we set the input dimension $n_0 = 10$.

Data generation and numerical results

Note that in order to generate the observation vector $\mathbf{y}$, one needs to assume a parameter field $u(\mathbf{x})$ and solve (5.3). The parameter field $u(\mathbf{x})$ is generated as follows:

$$u(\mathbf{x}) = \sum_{i=1}^{n_T} \sqrt{\lambda_i}\ \phi_i(\mathbf{x})\ c_i, \quad \mathbf{x} \in [0,\ 1]^2, \tag{5.4}$$

where $(\lambda_i,\ \phi_i)$ is the eigenpair of an exponential two-point correlation function from [14] and $\mathbf{c} = (c_1, \dots, c_{n_T}) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is a standard Gaussian random vector. Note that we consider $\mathbf{c}$

as the network output with dimension $n_T = 12$. In addition, 5% additive Gaussian noise is added to the observations $\mathbf{y}$ to represent the actual field condition. Therefore, the training data points can be denoted as $\{\mathbf{y}_i,\ \mathbf{c}_i\}_{i=1}^{S}$. Given a new observation data $\mathbf{y}$, the network outputs the vector $\mathbf{c}$ which can then be used to reconstruct $u(\mathbf{x})$ using (5.4). To quantify the performance of our algorithm, we compute the average relative errors on the test dataset as follows:

$$\text{Err} = \frac{1}{M}\sum_{i=1}^{M}\frac{\left\|\mathbf{u}_i^{pred} - \mathbf{u}_i^{true}\right\|^2}{\|\mathbf{u}_i^{true}\|^2},$$

where M denotes the number of test data samples, $\mathbf{u}_i^{pred}$ denotes the neural network prediction for the parameter field $u(\mathbf{x})$ for the i^{th} sample and $\mathbf{u}_i^{true}$ denotes the synthetic ground truth parameters. Here, $\mathbf{u}$ is a vector containing the solutions on a 16×16 grid shown in Figure 10 (right).

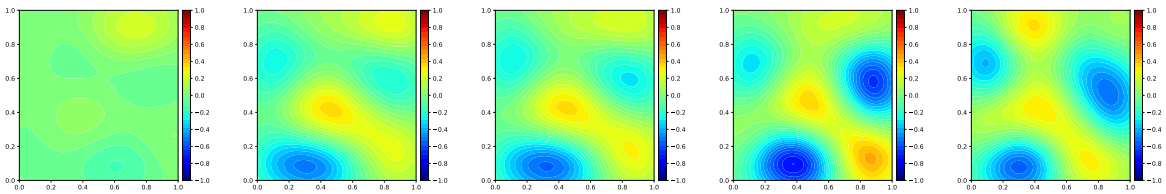


Figure 6. *Evolution of parameter field $u(\mathbf{x})$ for a particular test observation upon adding new layers for $S = 1000$ (Left to Right): inverse solution after the 1^{st} iteration; inverse solution after the 3^{rd} iteration; inverse solution after the 5^{th} iteration; inverse solution after the 7^{th} iteration; and the groundtruth parameter distribution.*

Further, we consider experiments with training data set of size $S = 1000$ and $S = 1500$. We consider an additional 200 data points for generating the validation data set and 500 data points for the testing data set. Other details on the hyperparameter settings are provided in Table 9. Note that for each adaptation approach, the best network (in terms of least average relative error on the validation dataset) out of 100 random initialization is retained for comparison purposes (such as for generating Figure 6 and Figure 7).

Figure 6 shows the parameter field (for a particular observational data input) predicted by our proposed approach at different iterations of our algorithm. It is interesting to see that while the initial networks with a smaller number of parameters learn the low-frequency components of the solution, later iterations of the algorithm capture the more complex features in the solution (as more parameters are added). In other words, our algorithm identifies missing high-level (high-frequency) features and inserts new layers to capture them. Further, the final parameter field is visibly close to the ground truth solution as evident from Figure 6. Further, in an attempt to estimate the accuracy of each adaptation strategy (for a graphical representation of the error by each approach), we define the pointwise average relative error as:

$$\text{Err}_j = \frac{1}{M}\sum_{i=1}^{M}\frac{\left(\mathbf{u}_{i,j}^{pred} - \mathbf{u}_{i,j}^{true}\right)^2}{\|\mathbf{u}_i^{true}\|^2/|\mathbf{u}_i|}, \tag{5.5}$$

where subscript j denotes the j^{th} component of $\mathbf{u}_i$ and $|\mathbf{u}_i|$ denotes the number of elements in the vector. Figure 7 shows the error Err_j plotted for the main adaptation strategies. It is quite clear from Figure 7 that our approach outperforms all other adaptation strategies by a good margin.

Table 1

Statistics ($\mu \pm \sigma$) of the relative error (rel. error) for different methods (2D heat equation)

Method	rel. error ($S = 1000$)	rel. error ($S = 1500$)	Best rel. error $S = 1000$	Best rel. error $S = 1500$	Training time for $S = 1000$
Proposed (II)	$\mathbf{0.400} \pm 0.032$	$\mathbf{0.391} \pm 0.033$	**0.327**	**0.321**	11.2 min
Proposed (I)	0.434 ± 0.022	0.429 ± 0.023	0.351	0.364	15.1 min
Random layer insertion (I)	0.457 ± 0.031	0.447 ± 0.029	0.392	0.360	13.7 min
Net2DeeperNet (II) [12]	0.456 ± 0.027	0.451 ± 0.022	0.400	0.362	12.4 min
Baseline network	0.50 ± 0.028	0.489 ± 0.029	0.446	0.420	16.4 min
Forward Thinking [22]	0.66 ± 0.033	0.65 ± 0.034	0.570	0.555	12 min
NAS (our activation) [34, 35]	$- - \pm - -$	$- - \pm - -$	0.349	0.344	376 min
NAS (ReLU activation) [34, 35]	$- - \pm - -$	$- - \pm - -$	0.355	0.347	380 min
NAS (tanh activation) [34, 35]	$- - \pm - -$	$- - \pm - -$	0.350	0.342	370 min

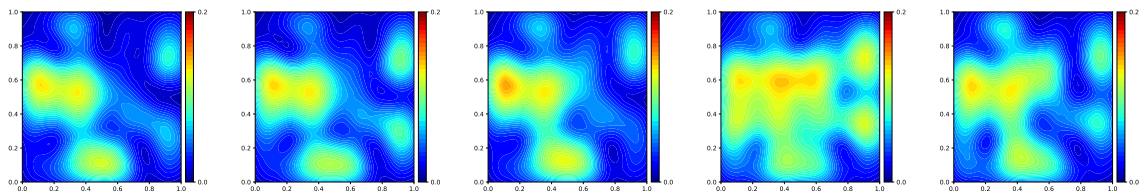


Figure 7. *Average error in the predicted parameter field over the spatial domain (for the full test data set, $S = 1000$). Left to right: Proposed method (II); Random layer insertion (I); Net2DeeperNet (II) [12]; Forward Thinking [22]; Baseline.*

The uncertainty associated with each method is also investigated and the results are provided in Table 1 where it is clear that our proposed method outperformed all other strategies (details of other methods are provided in Appendix B). Note that the relative error reported is with respect to the test dataset. Further, we have also conducted experiments with a larger data set ($S = 1500$) and the results are shown in Table 1. When considering the best relative error achieved, we observe from Table 1 that the effect of our proposed strategy (I) over other methods decreases as the number of data points increases. However, the fully-automated network growing presented in section 3 (Proposed (II)) seems to outperform all the other methods both in the low and high training data regime. More theoretical analysis is necessary to characterize our algorithm's performance in low and high data regimes (see the discussion in section 6 for more details). We hypothesize that the data-dependent initialization of the added layers in our approach plays a key role in guiding the optimization toward good local minima, leading to better generalization. Note that the baseline network presented in Table 1, although having the same number of layers as Proposed (I), uses randomly initialized parameters leading to poor generalization performance.

Further, a comparison of training times for different algorithms is provided in Table 1. As

shown, our fully automated algorithm, Proposed (II), achieves the best relative error in the shortest computational time. Proposed (I) requires more time because each layer is trained for a larger number of epochs (due to the use of a scheduler) compared to Proposed (II). The baseline network takes even longer because it trains a larger number of parameters from the start. While Table 1 shows that neural architecture search (NAS) can produce good results, it is the most computationally expensive approach among all the methods.

Since our algorithm and all methods in Table 1 uses a new activation function $\sigma(x) = \alpha_1 \text{ Swish(x)} + \alpha_2 \tanh(\text{x})$, where the coefficients $\alpha_1,\ \alpha_2$ are defined in Item 2, it is necessary to examine what happens when popular activations such as tanh(x) or ReLU(x) are used instead in the hidden layers. We conducted a neural architecture search (NAS) using various activation functions, and the results are shown in Table 1. As illustrated in Table 1, our activation function performs on par with, and in some cases slightly better than, ReLU and tanh.

5.2.2. Learning the observable to parameter map for 2D Navier-Stokes equation. In this section, we consider another inverse problem concerning the 2D Navier-Stokes equation for viscous and incompressible fluid [37] written as:

$$
\begin{aligned}
\partial_t u(\mathbf{x},\ t) + \mathbf{v}(\mathbf{x},\ t) \cdot \nabla u(\mathbf{x},\ t) &= \nu \Delta u(\mathbf{x},\ t) + f(\mathbf{x}), && \mathbf{x} \in (0,\ 1)^2,\ t \in (0,\ T] \\
\nabla \cdot \mathbf{v}(\mathbf{x},\ t) &= 0, && \mathbf{x} \in (0,\ 1)^2,\ t \in (0, T] \\
u(\mathbf{x},\ 0) &= u_0(\mathbf{x}), && \mathbf{x} \in (0,\ 1)^2,
\end{aligned} \tag{5.6}
$$

where $\mathbf{v}(\mathbf{x},\ t)$ is the velocity field, $u(\mathbf{x},\ t)$ is the vorticity, $u_0(\mathbf{x})$ is the initial vorticity, $f(\mathbf{x}) = 0.1\left(\sin\left(2\pi(x_1 + x_2)\right) + \cos\left(2\pi(x_1 + x_2)\right)\right)$ is the forcing function, and $\nu = 10^{-3}$ is the viscosity coefficient. The spatial domain is discretized with 32×32 uniform mesh, while the time horizon $t \in (0, 10)$ is subdivided into 1000 time steps with $\Delta t = 10^{-2}$. Our objective is to reconstruct the initial vorticity $u_0(\mathbf{x})$ from the measurements of vorticity at 20 observed points at the final time $T = 10$ (denoted as $\mathbf{y}$), i.e, in this case, we have input dimension $n_0 = 20$. Additional details on dataset generation is provided in Appendix D.2.

We consider experiments with training data set of size $S = 250$ and $S = 500$. We consider an additional 50 data points for validation data set and 200 data points for testing data set. Other details on the hyperparameter settings are provided in Table 9. Figure 8 shows the error

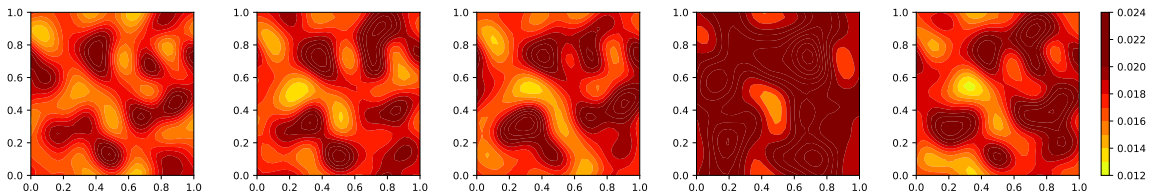


Figure 8. *Average error in the predicted parameter field over the spatial domain (for the full test data set, $S = 250$). Left to right: Proposed method (II) (best rel. error: 0.295); Random layer insertion (I) (best rel. error: 0.299); Net2DeeperNet (II) [12] (best rel. error: 0.309); Forward Thinking [22] (best rel. error: 0.362); Baseline (best rel. error: 0.305).*

Err_j (equation (5.5)) plotted for each approach (for S=250) where it is clear that our proposed

approach provided the best results. Additional results on the performance (statistics of relative error) for each method is also provided in Table 4, where we observed that in the low data regime, when one considers the average relative error, our proposed approach outperformed all other strategies, performing on par with a neural architecture search algorithm. However, as the dataset size increased, we observed that random layer insertion (I) strategy performed equally well to our approach (see Table 4). We hypothesize that our initialization strategy based on local sensitivity analysis matters more (in terms of generalization) in the low-data regime. Similar observations have been made in subsection 5.2.1.

5.3. Topological derivative informed transfer learning approach. Transfer learning is a machine learning technique in which knowledge gained through one task or dataset is used to improve the model's performance on a new dataset by fine-tuning the model on the new dataset. In this section, we demonstrate how the results in Theorem 2.7 can be used for adapting a pre-trained model in the context of transfer learning.

5.3.1. Improving performance of pre-trained state-of-the-art machine learning models. Using models pre-trained on large datasets and transferring to tasks with fewer data points is a commonly adopted strategy in machine learning literature [16]. In this section, we consider a tiny vision transformer model (ViT) pre-trained on ImageNet dataset (more details on the specific architecture employed is provided in Appendix D.6). Our objective is to fine-tune this model to achieve the best performance on the CIFAR-10 dataset. The traditional approach is to modify the output layer, i.e the MLP (multilayer perception) head of the ViT to have an output dimension of 10 and retrain the whole network to achieve the best performance on CIFAR-10 dataset. Let's call the best-performing model obtained this way as "ViT baseline" (as denoted in Table 2) Our objective in this section is to use the topological derivative approach as a post-processing stage to improve the performance of "ViT baseline" by further adapting the MLP Head using Algorithm 2.1.

Table 2
Accuracy achieved by different adaptation strategies

ViT baseline	Proposed (II)	Proposed (I)	Random layer insertion (I)	Net2DeepNet [12]	Forward Thinking [22]
90.9%	91.37%	**91.52**%	91.11 ± 0.027	91.11 ± 0.0137	90.9 ± 0.0

The best accuracy achieved by different adaptation strategies is shown in Table 2 and it is clear that our approach produces the best results. Note that for this task there is no source of randomness in our proposed approach and hence the statistics is not reported. However, other adaptation strategies has an inherent randomness due to random initialization of an added layer. Further, Forward Thinking [22] was unable to improve the performance over ViT baseline due to the freezing of previous layers at each adaptation step. Other details of hyperparameters used for the problem are provided in Appendix E. Numerical experiments on fine-tuning internal encoder blocks, rather than the MLP head, are beyond the scope of the present work. In such a setting, our approach could be used to address the following questions: (i) which encoder block should be adapted; (ii) for the selected encoder block, where to add a new layer in the MLP; and (iii) how to initialize the added layer. State-of-the-art fine-tuning

methods, such as LoRA [24], while effective, do not address these questions in a principled manner. This type of multi-level adaptation framework is beyond the scope of the present work and will be explored in future research.

5.3.2. Application in parameter-efficient fine tuning. Parameter-efficient fine tuning is a transfer learning approach to address catastrophic forgetting [30] by freezing many parameters/layers in the pre-trained network thereby preventing overfitting on the new data [46]. The procedure adopted in parameter-efficient transfer learning may be summarized as follows: a) Train a neural network from scratch on the large dataset $\mathcal{D}_s$; b) Fine tune a small part of the network (by freezing parameters in rest of the network) to fit the sparse data-set $\mathcal{D}_e$. However, in this procedure, it is often unclear on which hidden layers of the network needs retraining to fit the new data $\mathcal{D}_e$. Subel et al. [54] pointed out that common wisdom guided transfer learning wherein one trains the last few layers is not necessarily the best option. Often in literature one finds that the layers that need retraining are treated as a hyperparameter and a random search is conducted to arrive at the best decision [44].

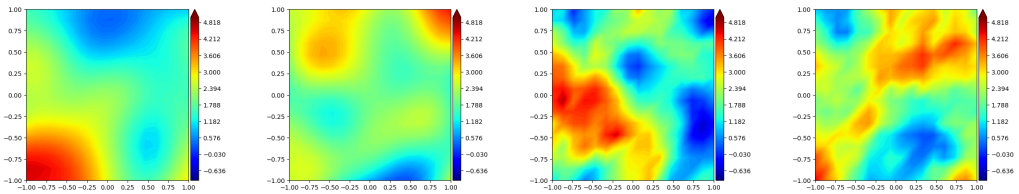


Figure 9. *Samples of conductivity field drawn from different distributions. Left to right: First two are samples from autocorrelation prior; Last two are samples from a BiLaplacian prior.*

In this section, we demonstrate how the topological derivative in Theorem 2.7 could be used to inform which layers need retraining in parameter-efficient transfer learning. In particular, we seek to determine the most sensitive part of the network with respect to the new data set $\mathcal{D}_e$. This is accomplished by defining the loss functional in (2.11) based on the new data set $\mathcal{D}_e$ and computing the topological derivative which informs where to add a new layer as described in subsection 2.4. For demonstration, we revisit the heat equation (5.3) where the objective now is to learn a surrogate for the parameter to observable map (PtO). In particular, the network takes in the conductivity field $\mathbf{u} \in \mathbb{R}^{256}$ as input and predicts the corresponding temperature $\mathbf{y} \in \mathbb{R}^{50}$ at fixed locations on the domain. To train the initial network, the data-set $\mathcal{D}_s$ consists of $10,000$ data points where the conductivity field $\mathbf{u}$ is drawn from a BiLaplacian prior [61] with mean $\mu = 2$ and covariance $\mathcal{C}$ given in (5.7) with parameters $(\gamma,\ \delta) = (0.1,\ 0.5)$. It is expected that with the passage of time the distribution of conductivity field $\mathbf{u}$ changes due to changes in field conditions. To simulate this shift in distribution, the new data-set $\mathcal{D}_e$ consists of 50 samples where $\mathbf{u}$ is drawn from a Gaussian autocorrelation smoothness prior [25] with mean $\mu = \mathbf{2}$ and covariance $\mathbf{\Gamma}$ given in (5.7) with with $\sigma^2 = 2,\ \rho = 0.5$.

$$\mathcal{C} = (\delta I + \gamma \nabla \cdot (\nabla))^{-2}, \quad \mathbf{\Gamma}_{ij} = \sigma^2 \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{2\rho^2}\right). \tag{5.7}$$

Samples of generated conductivity fields from the two distributions are shown in Figure 9.

Table 3
Mean squared error achieved by different transfer learning approaches

Complete retraining	Traditional transfer learning	Proposed approach	Exhaustive search	
			Best case	Worst case
3.572	5.040	**2.576**	2.604	3.240

Performance of different transfer learning approaches on a test data set of size 1000 is shown in Table 3 where it is clear that our proposed approach provides the best results. In Table 3, "Complete retraining" refers to the case when the entire pre-trained network is trained to fit the new data; "Traditional transfer learning" refers to retraining only the last layer in the network; "Proposed approach" refers to the topological derivative approach where a new layer is added and retrained along with the first and last layers; and "Exhaustive search" refers to the case of retraining the last layer, first layer and a randomly chosen hidden layer. In addition, we also observed that while our proposed approach and traditional transfer learning takes approximately 3 min to retrain the network, an exhaustive search procedure conducted serially took around 20 min to find the best retrained network. Additional justification (with numerical results) on why topological derivative serves as a good indicator for determining where to add a new layer is provided in Appendix D.7.

6. Conclusion. In this work, we derived an expression for the "topological derivative" for a neural network and demonstrated how the concept can be used for progressively adapting a neural network along the depth. In section 4, we also showed that our layer insertion strategy can be viewed through the lens of an optimal transport problem. Numerically, we observed that our proposed method outperforms other adaptation strategies for most datasets considered in the study. In particular, our method (especially Proposed (I)) exhibits superior performance compared to other approaches in the low training data regime. Numerical results further suggest that employing a scheduler (Proposed I) is in general effective leading to superior performance in comparison to other strategies although occasionally our Proposed (II) (fully automated growing in Remark 2.16) exhibited superior performance. Note that when the training dataset is very large, it is sufficient to ensure a good fit to the training data to achieve strong generalization. In this setting, the problem of architecture adaptation—namely, how to distribute parameters within a network—becomes less relevant. Instead, it is often sufficient to consider a very deep network and train it to fit the data well, as demonstrated by the success of large language models.

Note that the topological derivative approach is a greedy approach where one chooses the best decision (locally) on where to add a layer along with the initialization at each step to reach at a globally optimal architecture. Here we define globally optimal architecture as the one that gives the lowest generalization error out of all possible architecture sampled from a predefined search space (see Appendix B for the search space we consider) and considering all random initializations from the normal distribution $\mathcal{N}\left(\mathbf{0}, \sigma_n^2 \mathbf{I}\right)$. In spite of our approach being greedy, we observe that our approach achieves comparable performance to that of a neural architecture search algorithm [35] for the datasets considered in this study. Deriving bounds on the distance between the best architecture from our algorithm and the globally

optimal architecture is beyond the scope of present work and will be investigated in the future (previous works have developed distance metric in the space of neural network architectures [26]). We anticipate that this analysis would provide insight on why our method (especially Proposed (I)) work particularly well in the low training data regime in comparison to other methods.

Acknowledgment. The authors would like to thank Hai Van Nguyen for fruitful discussions and helping to generate some of the data sets for computational experiments. The authors also acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC, visualization, database, or grid resources that have contributed to some of the results reported within this paper. URL: http://www.tacc.utexas.edu.

REFERENCES

[1] S. Amstutz, *An introduction to the topological derivative*, Engineering Computations, 39 (2022), pp. 3–33.
[2] S. Amstutz, I. Horchani, and M. Masmoudi, *Crack detection by the topological gradient method*, Control and cybernetics, 34 (2005), pp. 81–101.
[3] D. Auroux, L. Jaafar-Belaid, and B. Rjaibi, *Application of the topological gradient method to tomography*, Revue Africaine de Recherche en Informatique et Mathématiques Appliquées, 13 (2010).
[4] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, *Deephyper: Asynchronous hyperparameter search for deep neural networks*, in 2018 IEEE 25th international conference on high performance computing (HiPC), IEEE, 2018, pp. 42–51.
[5] L. J. Belaid, M. Jaoua, M. Masmoudi, and L. Siala, *Image restoration and edge detection by topological asymptotic expansion*, Comptes Rendus Mathematique, 342 (2006), pp. 313–318.
[6] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, et al., *Greedy layer-wise training of deep networks*, Advances in neural information processing systems, 19 (2007), p. 153.
[7] M. Benning, E. Celledoni, M. J. Ehrhardt, B. Owren, and C.-B. Schönlieb, *Deep learning as optimal control problems: Models and numerical methods*, arXiv preprint arXiv:1904.05657, (2019).
[8] Y. Bodyanskiy, A. Pirus, and A. Deineko, *Multilayer radial-basis function network and its learning*, in 2020 IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT), vol. 1, IEEE, 2020, pp. 92–95.
[9] J. Büchel, F. Faber, and D. R. Muir, *Network insensitivity to parameter noise via adversarial regularization*, arXiv preprint arXiv:2106.05009, (2021).
[10] T. Bui-Thanh, *A unified and constructive framework for the universality of neural networks*, IMA Journal of Applied Mathematics, 89 (2024), pp. 197–230.
[11] R. Carmona, F. Delarue, et al., *Probabilistic theory of mean field games with applications I-II*, Springer, 2018.
[12] T. Chen, I. Goodfellow, and J. Shlens, *Net2net: Accelerating learning via knowledge transfer*, arXiv preprint arXiv:1511.05641, (2015).
[13] N. Cheney, M. Schrimpf, and G. Kreiman, *On the robustness of convolutional neural networks to internal architecture and weight perturbations*, arXiv preprint arXiv:1703.08245, (2017).
[14] P. G. Constantine, C. Kent, and T. Bui-Thanh, *Accelerating markov chain monte carlo with active subspaces*, SIAM Journal on Scientific Computing, 38 (2016), pp. A2779–A2805.
[15] R. Craddock and K. Warwick, *Multi-layer radial basis function networks. an extension to the radial basis function*, in Proceedings of International Conference on Neural Networks (ICNN'96), vol. 2, IEEE, 1996, pp. 700–705.
[16] A. Dosovitskiy, *An image is worth 16x16 words: Transformers for image recognition at scale*, arXiv preprint arXiv:2010.11929, (2020).
[17] T. Elsken, J. H. Metzen, and F. Hutter, *Efficient multi-objective neural architecture search via lamarckian evolution*, arXiv preprint arXiv:1804.09081, (2018).

[18] H. A. Eschenauer, V. V. Kobelev, and A. Schumacher, *Bubble method for topology and shape optimization of structures*, Structural optimization, 8 (1994), pp. 42–51.
[19] U. Evci, B. van Merrienboer, T. Unterthiner, M. Vladymyrov, and F. Pedregosa, *Gradmax: Growing neural networks using gradient information*, arXiv preprint arXiv:2201.05125, (2022).
[20] A. Fawzi, S.-M. Moosavi-Dezfooli, and P. Frossard, *Robustness of classifiers: from adversarial to random noise*, Advances in neural information processing systems, 29 (2016).
[21] S. R. Garcia and R. A. Horn, *Block matrices in linear algebra*, PRIMUS, 30 (2020), pp. 285–306.
[22] C. Hettinger, T. Christensen, B. Ehlert, J. Humpherys, T. Jarvis, and S. Wade, *Forward thinking: Building and training neural networks one layer at a time*, arXiv preprint arXiv:1706.02480, (2017).
[23] G. E. Hinton, *Learning multiple layers of representation*, Trends in cognitive sciences, 11 (2007), pp. 428–434.
[24] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, et al., *Lora: Low-rank adaptation of large language models.*, ICLR, 1 (2022), p. 3.
[25] J. Kaipio and E. Somersalo, *Statistical and computational inverse problems*, vol. 160, Springer Science & Business Media, 2006.
[26] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, *Neural architecture search with bayesian optimisation and optimal transport*, Advances in neural information processing systems, 31 (2018).
[27] P. Kidger and T. Lyons, *Universal approximation with deep narrow networks*, in Conference on learning theory, PMLR, 2020, pp. 2306–2327.
[28] Y. Kilcher, G. Bécigneul, and T. Hofmann, *Escaping flat areas via function-preserving structural network modifications*, (2018).
[29] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).
[30] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al., *Overcoming catastrophic forgetting in neural networks*, Proceedings of the national academy of sciences, 114 (2017), pp. 3521–3526.
[31] L. Kreis, E. Herberg, F. Köhne, A. Schiela, and R. Herzog, *Sensli: Sensitivity-based layer insertion for neural networks*, arXiv preprint arXiv:2311.15995, (2023).
[32] C. Krishnanunni and T. Bui-Thanh, *Layerwise sparsifying training and sequential learning strategy for neural architecture adaptation*, arXiv preprint arXiv:2211.06860, (2022).
[33] M. Kulkarni and S. Karande, *Layer-wise training of deep networks using kernel similarity*, arXiv preprint arXiv:1703.07115, (2017).
[34] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, B. Recht, and A. Talwalkar, *A system for massively parallel hyperparameter tuning*, Proceedings of Machine Learning and Systems, 2 (2020), pp. 230–246.
[35] L. Li and A. Talwalkar, *Random search and reproducibility for neural architecture search*, in Uncertainty in artificial intelligence, PMLR, 2020, pp. 367–377.
[36] Q. Li and S. Hao, *An optimal control approach to deep learning and applications to discrete-weight neural networks*, in International Conference on Machine Learning, PMLR, 2018, pp. 2985–2994.
[37] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Fourier neural operator for parametric partial differential equations*, arXiv preprint arXiv:2010.08895, (2020).
[38] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, *A survey on evolutionary neural architecture search*, IEEE transactions on neural networks and learning systems, (2021).
[39] K. Maile, E. Rachelson, H. Luga, and D. G. Wilson, *When, where, and how to add new neurons to anns*, in International Conference on Automated Machine Learning, PMLR, 2022, pp. 18–1.
[40] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al., *Evolving deep neural networks*, in Artificial intelligence in the age of neural networks and brain computing, Elsevier, 2019, pp. 293–312.
[41] G. Montavon, M. L. Braun, and K.-R. Müller, *Kernel analysis of deep networks*, Journal of Machine Learning Research, 12 (2011), pp. 2563–2581.
[42] G. Montavon, K.-R. Muller, and M. Braun, *Layer-wise analysis of deep networks with gaussian*

kernels, Advances in neural information processing systems, 23 (2010), pp. 1678–1686.
[43] H. V. Nguyen and T. Bui-Thanh, *Tnet: A model-constrained tikhonov network approach for inverse problems*, SIAM Journal on Scientific Computing, 46 (2024), pp. C77–C100.
[44] M. L. Olson, S. Liu, J. J. Thiagarajan, B. Kustowski, W.-K. Wong, and R. Anirudh, *Transformer-powered surrogates close the icf simulation-experiment gap with extremely limited data*, Machine Learning: Science and Technology, 5 (2024), p. 025054.
[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research, 12 (2011), pp. 2825–2830.
[46] C. Poth, H. Sterz, I. Paul, S. Purkayastha, L. Engländer, T. Imhof, I. Vulić, S. Ruder, I. Gurevych, and J. Pfeiffer, *Adapters: A unified library for parameter-efficient and modular transfer learning*, arXiv preprint arXiv:2311.11077, (2023).
[47] A. Rahnama, A. T. Nguyen, and E. Raff, *Robust design of deep neural networks against adversarial attacks based on lyapunov theory*, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 8178–8187.
[48] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, *Regularized evolution for image classifier architecture search*, in Proceedings of the aaai conference on artificial intelligence, vol. 33, 2019, pp. 4780–4789.
[49] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al., *Imagenet large scale visual recognition challenge*, International journal of computer vision, 115 (2015), pp. 211–252.
[50] F. Santambrogio, *Optimal transport for applied mathematicians*, Birkäuser, NY, 55 (2015), p. 94.
[51] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, arXiv preprint arXiv:1409.1556, (2014).
[52] J. Sokolowski and A. Zochowski, *On the topological derivative in shape optimization*, SIAM journal on control and optimization, 37 (1999), pp. 1251–1272.
[53] K. O. Stanley and R. Miikkulainen, *Evolving neural networks through augmenting topologies*, Evolutionary computation, 10 (2002), pp. 99–127.
[54] A. Subel, Y. Guan, A. Chattopadhyay, and P. Hassanzadeh, *Explaining the physics of transfer learning a data-driven subgrid-scale closure to a different turbulent flow*, arXiv preprint arXiv:2206.03198, (2022).
[55] M. Suganuma, S. Shirakawa, and T. Nagao, *A genetic programming approach to designing convolutional neural network architectures*, in Proceedings of the genetic and evolutionary computation conference, 2017, pp. 497–504.
[56] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 1–9.
[57] E. Todorov et al., *Optimal control theory*, Bayesian brain: probabilistic approaches to neural coding, (2006), pp. 268–298.
[58] L. N. Trefethen and D. Bau, *Numerical linear algebra*, SIAM, 2022.
[59] Y.-L. Tsai, C.-Y. Hsu, C.-M. Yu, and P.-Y. Chen, *Formalizing generalization and robustness of neural networks to weight perturbations*, arXiv preprint arXiv:2103.02200, (2021).
[60] W. Wen, F. Yan, Y. Chen, and H. Li, *Autogrow: Automatic layer growing in deep convolutional networks*, in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 833–841.
[61] J. Wittmer, C. Krishnanunni, H. V. Nguyen, and T. Bui-Thanh, *On unifying randomized methods for inverse problems*, Inverse Problems, 39 (2023), p. 075010.
[62] L. Wu, B. Liu, P. Stone, and Q. Liu, *Firefly neural architecture descent: a general approach for growing neural networks*, Advances in neural information processing systems, 33 (2020), pp. 22373–22383.
[63] L. Wu, D. Wang, and Q. Liu, *Splitting steepest descent for growing neural architectures*, Advances in Neural Information Processing Systems, 32 (2019).
[64] M. Wynne-Jones, *Node splitting: A constructive algorithm for feed-forward neural networks*, Neural Computing & Applications, 1 (1993), pp. 17–22.
[65] D. Xu and J. C. Principe, *Training mlps layer-by-layer with the information potential*, in IJCNN'99.

International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339), vol. 3, IEEE, 1999, pp. 1716–1720.
[66] L. Yu, Y. Wang, and X.-S. Gao, *Adversarial parameter attack on deep neural networks*, in International Conference on Machine Learning, PMLR, 2023, pp. 40354–40372.
[67] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, in European conference on computer vision, Springer, 2014, pp. 818–833.
[68] B. Zoph and Q. V. Le, *Neural architecture search with reinforcement learning*, arXiv preprint arXiv:1611.01578, (2016).

SUPPLEMENTARY MATERIAL: TOPOLOGICAL DERIVATIVE APPROACH FOR DEEP NEURAL NETWORK ARCHITECTURE ADAPTATION.

Appendix A. Architecture adaptation for convolutional neural network (CNN). Note that though subsection 2.2 and subsection 2.3 present the approach in the context of a fully connected network, the framework can also be applied to a CNN architecture where the input is a tensor of shape: (input height) × (input width) × (input channels). Note that in this case the function $\mathbf{f}_{t+1}(\mathbf{x}_{s,t};\ \boldsymbol{\theta}_{t+1})$ in (2.1) simply represents the 2D convolutional layer where $\mathbf{x}_{s,t}$ represents the vectorized input and $\boldsymbol{\theta}_{t+1}$ represents the vectorized filter parameters. Note that $\mathbf{f}_t(.;\ .)$ satisfies the assumptions in Theorem 2.7 when employing differentiable activation functions.

Algorithm A.1 CNN architecture adaptation algorithm

Input: Training data $\mathbf{X}$, labels $\mathbf{C}$, validation data $\mathbf{X}_1$, validation labels $\mathbf{C}_1$, number of channels in each hidden layer n, loss function Φ, filter size $f \times f \times n$, stride s, number of channels to activate in each hidden layer m, number of iterations N_n, parameter ε, ε^t, parameter T_b, hyperparameters and predefined scheduler for optimizer (Appendix E).

Initialize: Initialize network $\mathcal{Q}_1$ with T_b hidden layers.

Train network $\mathcal{Q}_1$ and store the validation loss $(\varepsilon_v)^1$.
set $i = 1$, $(\varepsilon_v)^0 >> (\varepsilon_v)^1$, $\Lambda_l^m \geq \varepsilon^t$
while $i \leq N_n$ **and** $\left[(\varepsilon_v)^i \leq (\varepsilon_v)^{i-1}\right]$ **and** $\Lambda_l^m \geq \varepsilon^t$ **do**
    Compute the topological derivative for each layer l using (2.35) and store as $\{\Lambda_l\}$, also store $\Lambda_l^m = \max_l\{\Lambda_l\}$.
    Store the corresponding eigenvectors for each layer as Φ_l representing n filters of size $f \times f \times n$.
    Obtain the new network $\mathcal{Q}_{i+1}$ by adding a new layer at position $l^* = \underset{l}{\operatorname{argmax}}\{\Lambda_l\}$ with filter parameters $\varepsilon\Phi_{l^*}$.
    Perform a backtracking line search to update ε as outlined in Algorithm C.1.
    Update the parameters for optimizer if required (refer Appendix E for scheduler details).
    Train network $\mathcal{Q}_{i+1}$ and store the best validation loss $(\varepsilon_v)^{i+1}$ and the best network $\mathcal{Q}_{i+1}$.
    $i = i + 1$
end while

Output: Network $\mathcal{Q}_{i-1}$

In order to satisfy condition 1 in Proposition 2.3 (ResNet propagation), zero-padding is employed to preserve the original input size. Note that in this case Φ_l in (2.35) denotes the filter parameters for the added layer (vectorized). Our algorithm for a CNN architecture is given in Algorithm A.1 and is self-explanatory. Note that our proposed approach can be applied to any architecture as long as conditions in Proposition 2.3 and Theorem 2.7 are satisfied.

Appendix B. General setting for numerical experiments. All codes were written in PyTorch. Throughout the study, we have employed the Adam optimizer [29] for minimizing the loss function. Our proposed approach is compared with a number of different approaches as given below:

Proposed (I) : Semi-automated architecture adaptation based on the topological derivative approach as described in Algorithm 2.1, Algorithm A.1.

Proposed (II) : Automated architecture adaptation based on the topological derivative approach as described in section 3 and Algorithm 3.1.

Random layer insertion (I) : Adaptation strategy based on inserting a new layer at random position initialized with $\varepsilon\Phi$ where Φ is a random unit vector.

Net2DeeperNet (II) : Increasing depth of network based on based on function preserving transformations. A layer is inserted at random position with a small Gaussian noise added to the parameters to break symmetry [12].

Baseline network (B) : Training a randomly initialized network with the same final architecture as obtained by our proposed approach.

Forward Thinking (H) : Algorithm for layerwise adaptation proposed by Hettinger et al. [22].

Neural Architecture Search (NAS) : Random search with early stopping proposed in [35, 34].

We maintain the same activation functions and hyperparameters for all the adaptation strategies in order to make a fair comparison (except Proposed (II) which does not have a predefined scheduler (see section 3)). Note that the strategies Proposed (I), Random layer insertion (I), and Net2DeeperNet (II) differ only in the way a newly added layer is initialized and where a new layer is inserted. The Baseline network (B) is trained for the same total number of epochs as the proposed method (I). Note that, for all methods, the reported numerical results correspond to the model that achieves the best validation loss. Further, the uncertainty in each approach is quantified (presented in the numerical results) by running the algorithm for different random initializations (100 in this case). For all problems we consider $\mathcal{F} = \{Swish, \ tanh\}$

(see Remark 2.4) for constructing the activation function $\sigma(x)$. Further, for neural architecture search [35, 34], the search space is defined by considering architectures with a maximum of N_n layers with each layer having a maximum of n neurons (refer to Appendix E for the values of N_n and n for each problem). We randomly sample a total of $50,000$ architectures in the beginning and follow the procedure in [35, 34] to arrive at the best architecture.

Appendix C. Backtracking algorithm. The backtracking line search for choosing parameter ε is provided in Algorithm C.1. Note that lines 1-4 in Algorithm C.1 try to find the ε that leads to a maximum decrease in loss $\mathcal{J}$. The values of τ_1 and ε used is provided in Appendix E.

Algorithm C.1 Backtracking line search

Input: Loss function $\mathcal{J}$, initialization of the added layer Φ_l, parameter τ_1, ε.

1: **while** $\mathcal{J}((\varepsilon+\tau_1)\Phi_l) \leq \mathcal{J}(\varepsilon\Phi_l)$ **do**
2: $\quad \varepsilon = \varepsilon + \tau_1$
3: **end while**

Output: Return ε as the solution.

Appendix D. Additional numerical results.

D.1. Learning the observable to parameter map for 2D heat equation. Figure 10 describes the domain Ω along with the fixed locations $\mathbf{x}_i$ in subsection 5.2.1.

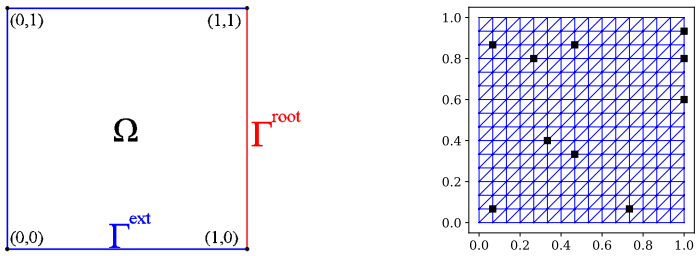


Figure 10. *2D heat conductivity inversion problem (Left to Right): The domain and the boundaries; A* 16×16 *finite element mesh and* 10 *observational locations.*

D.2. Learning the observable to parameter map for 2D Navier-Stokes equation.

Data generation and numerical results. For dataset generation in subsection 5.2.2, we draw samples of $u(\mathbf{x},\ 0)$ based on the truncated Karhunen-Loève expansion as:

$$u(x,\ 0) = \sum_{i=1}^{n_T} \sqrt{\lambda_i}\omega_i(x)c_i, \tag{D.1}$$

where $\mathbf{c} = (c_1, \ldots, c_{n_T}) \sim \mathcal{N}(\mathbf{0}, I)$ is a standard Gaussian random vector, $(\lambda_i,\ \omega_i)$ are eigenpairs obtained by the eigendecomposition of the covariance operator $7^{\frac{3}{2}}\left(-\Delta + 49\mathbf{I}\right)^{-2.5}$ with

periodic boundary conditions. For demonstration, we choose $n_T = 15$. For a given $u_0(\mathbf{x})$, we solve the Navier-Stokes equation by the stream-function formulation with a pseudospectral method [37] to compute $u(\mathbf{x},\ T)$ on the grid, which is then used to generate the observation vector $\mathbf{y}$. Similar to subsection 5.2.1 for a new observation data $\mathbf{y}$, the network outputs the vector $\mathbf{c}$ which can then be used to reconstruct $u_0(\mathbf{x})$ using (D.1). Figure 11 shows the pa-

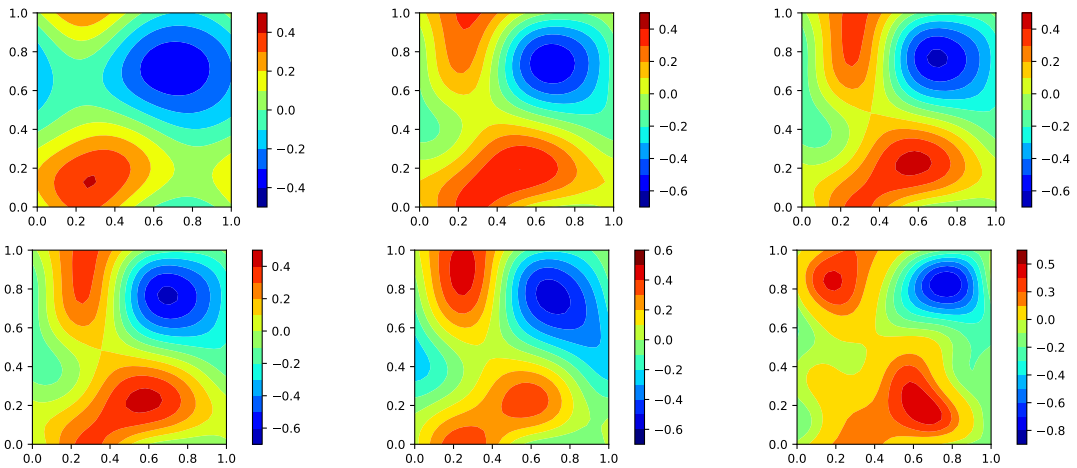


Figure 11. *Evolution of parameter field $u(\mathbf{x},\ T)$ for a particular test observation upon adding new layers for $S = 250$ (Left to Right): inverse solution after the 1^{st} iteration; inverse solution after the 2^{nd} iteration; inverse solution after the 3^{rd} iteration; inverse solution after the 4^{th} iteration; inverse solution after the 6^{th} iteration; and the groundtruth parameter distribution.*

rameter field (for a particular observational data input) predicted by our proposed approach at different iterations of our algorithm where we see that the predicted vorticity field improves as one adds more parameters (weights/biases).

Further, the performance (statistics of relative error) for each method is also provided in Table 4. Results in Table 4 are explained in subsection 5.2.2.

Table 4

Statistics ($\mu \pm \sigma$) of the relative error (rel. error) different methods (Navier-Stokes equation)

Method	rel. error ($S = 250$)	rel. error ($S = 500$)	Best error	
			$S = 250$	$S = 500$
Proposed (II)	0.328 ± 0.0227	0.283 ± 0.0039	**0.295**	0.274
Proposed (I)	**0.320** ± 0.0198	**0.277** ± 0.0034	0.301	0.271
Random layer insertion (I)	0.326 ± 0.0219	**0.277** ± 0.0036	0.299	**0.267**
Net2DeeperNet (II) [12]	0.347 ± 0.050	0.285 ± 0.006	0.309	0.275
Baseline network	0.350 ± 0.038	0.280 ± 0.0037	0.305	0.273
Forward Thinking [22]	0.429 ± 0.0286	0.313 ± 0.017	0.362	0.284
NAS [34, 35]	$- - \pm - -$	$- - \pm - -$	**0.295**	0.273

D.3. Performance on a real-world regression data set: The California housing dataset. In this section, we consider experiments with a real-world data set, i.e the California housing dataset (ML data set in scikit-learn [45]) where the task is to predict the housing prices given a set of 8 input features, i.e we have $n_0 = 8$ and $n_T = 1$ for this problem. We consider experiments with training data sets of sizes $S = 500$ and $S = 1000$, and considered a validation data set of size 100 and a testing data set of size 6192. Other details on the hyperparameter settings are provided in Table 9. Table 5 provides the summary of the result (we use mean squared error (mse) to quantify the performance) where it is clear that our proposed method (I) outperformed other strategies. However, the fully-automated network growing Algorithm 3.1 (i.e proposed (II)) does not seem to be very effective for this dataset in comparison to other adaptation strategies which could be due to overfitting in the early iterations of the algorithm.

Table 5

Statistics ($\mu \pm \sigma$) of the mean squared error (MSE) for different methods (California housing dataset)

Method	MSE ($S = 500$)	MSE ($S = 1000$)	Best error $S = 500$	Best error $S = 1000$
Proposed (II)	0.455 ± 0.015	0.407 ± 0.020	0.398	0.350
Proposed (I)	$\mathbf{0.448} \pm 0.020$	$\mathbf{0.384} \pm 0.0192$	**0.392**	**0.346**
Random layer insertion (I)	0.453 ± 0.016	0.386 ± 0.0185	0.403	0.350
Net2DeeperNet (II) [12]	0.455 ± 0.017	0.391 ± 0.0197	0.407	0.346
Baseline network	0.481 ± 0.032	0.391 ± 0.0207	0.405	0.351
Forward Thinking [22]	0.541 ± 0.028	0.430 ± 0.033	0.440	0.380
NAS [34, 35]	$--\pm---$	$--\pm---$	0.395	0.347

D.4. Wind velocity reconstruction problem. This example considers the wind velocity reconstruction problem where the objective is to predict the magnitude of wind velocity on a uniform 3D grid based on sparse measurement data. The observational dataset consists of (x, y, z) position components and the corresponding velocity $(u(x, y, z),\ v(x, y, z),\ w(x, y, z))$ over North America. We train a network that takes in the $(x,\ y,\ z)$ components as inputs and predicts the magnitude of wind velocity $\sqrt{u^2 + v^2 + z^2}$ as the output. We consider a training data set of sizes $S = 1000$ and $S = 5000$, a validation data set of size 2000 and a testing data set of size 21525. The performance of each method is provided in Table 6. It is clear that both our proposed method (I) and (II) outperformed all other strategies. Further, the improvement in solution (3D air current profile) on adding new layers is shown in Figure 12 where one clearly sees that the algorithm progressively picks up complex features in the solution as evident from more contour lines appearing in later stages of the algorithm. Further, Figure 13 shows the relative error in predictions w.r.t truth (equation (5.5)) over the 3D domain for different methods. It is clear from Figure 13 that our proposed approach provides the most accurate estimates.

D.5. Image classification with convolutional neural network. Finally, we test the performance of our adaptation scheme on a CNN architecture for an image classification task. In particular, we consider experiments on the MNIST handwritten dataset. Our architecture

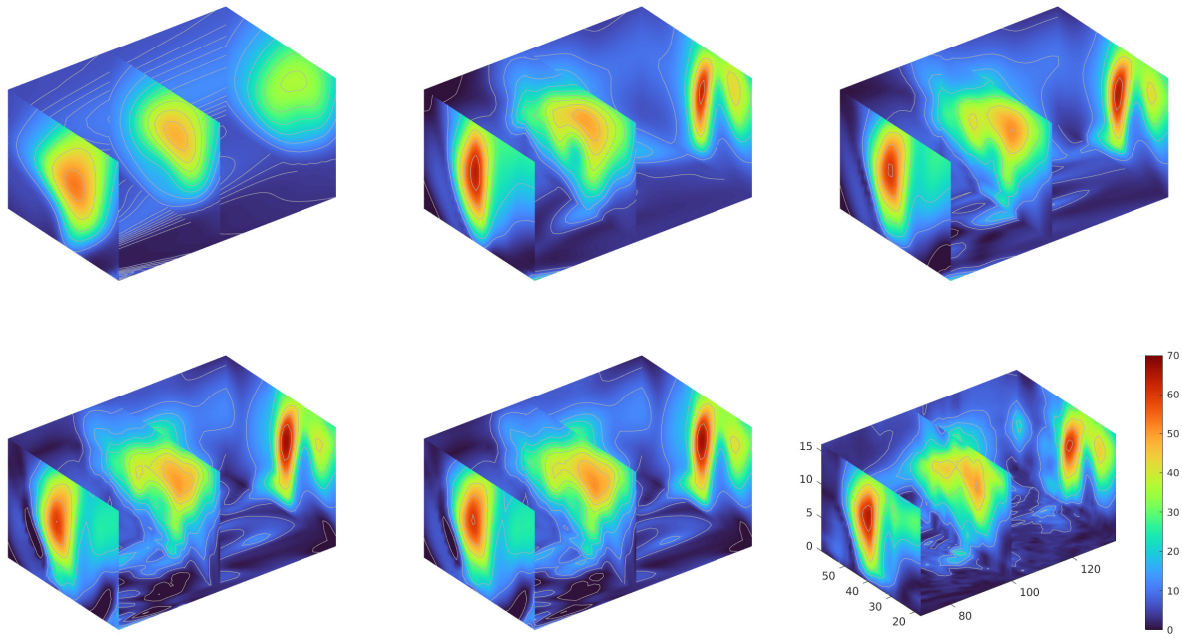


Figure 12. *Evolution of solution (3D air current profile) upon adding new hidden layers for $S = 1000$ (Left to right): Solution after the 4^{th} iteration; Solution after the 5^{th} iteration; Solution after the 6^{th} iteration; Solution after the 8^{th} iteration; Solution after the 10^{th} iteration; Ground truth solution.*

Table 6

Statistics ($\mu \pm \sigma$) of the mean squared error (MSE) for different methods (Wind velocity reconstruction)

Method	MSE ($S = 1000$)	MSE ($S = 5000$)	Best error	
			$S = 1000$	$S = 5000$
Proposed (II)	16.57 ± 2.26	5.32 ± 0.97	13.01	3.97
Proposed (I)	$\mathbf{16.01} \pm 2.90$	$\mathbf{4.89} \pm 1.23$	**12.40**	**3.77**
Random layer insertion (I)	17.85 ± 4.17	5.84 ± 1.41	13.29	4.68
Net2DeeperNet (II) [12]	19.10 ± 4.15	5.70 ± 1.86	13.28	4.60
Baseline network	18.96 ± 4.62	5.53 ± 2.02	13.24	4.53
Forward Thinking [22]	87.90 ± 17.64	20.59 ± 10.23	48.15	15.23
NAS [34, 35]	$- - \pm - -$	$- - \pm - -$	14.12	4.35

consists of an upsampling layer that maps the input image to m channels, a sequence of residual layers that we adapt, a downsampling layer that maps m channels to one channel, and a fully connected layer that outputs the probability vector to classify digits. Other details on the architecture are provided in Table 9. A summary of results for MNIST dataset (for two different sizes) is provided in Table 7. Even in this case, we observe that our algorithm significantly outperforms other methods in the low data regime as evident from the maximum accuracy achieved in Table 7. However, when looking at the performance achieved by training

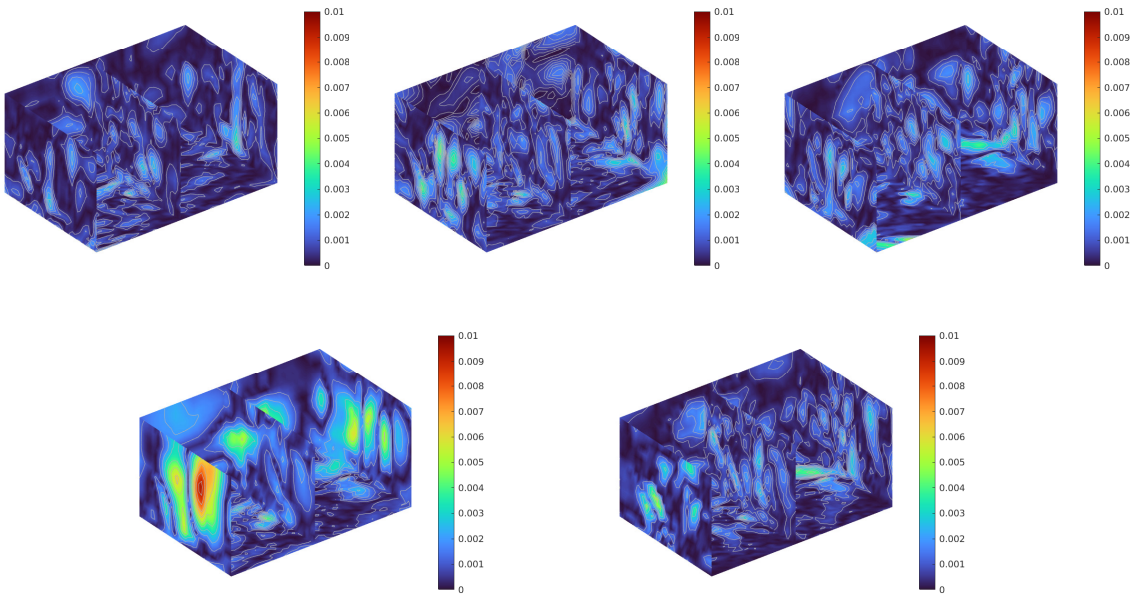


Figure 13. *Relative error in the predicted 3D air current profile over the spatial domain ($S = 1000$). Left to right: Proposed method (I); Random layer insertion (I); Net2DeeperNet (II) [12]; Forward Thinking [22]; Baseline.*

on the full dataset we do not see considerable advantages over other approaches.

Table 7
Statistics ($\mu \pm \sigma$) of the classification accuracy for different methods (MNIST Classification)

Method	Accuracy in % ($S = 600$)	Accuracy in % ($S = 60000$)	Best accuracy $S = 600$	Best accuracy $S = 60000$
Proposed (II)	86.52 ± 0.53	98.90 ± 0.07	87.40	98.95
Proposed (I)	$\mathbf{86.98} \pm 0.73$	98.92 ± 0.06	**88.44**	98.99
Random layer insertion (I)	86.78 ± 0.62	98.90 ± 0.06	88.00	98.95
Net2DeeperNet (II) [12]	86.76 ± 0.71	98.92 ± 0.07	87.64	98.99
Baseline network	86.78 ± 0.64	$\mathbf{98.98} \pm 0.06$	87.81	99.05
Forward Thinking [22]	86.60 ± 0.85	98.90 ± 0.08	87.66	98.95
NAS [34, 35]	$- - \pm - -$	$- - \pm - -$	88.00	**99.07**

D.6. Improving performance of pre-trained state-of-the-art machine learning models.

For discussion in subsection 5.3.1, we consider a ViT model by Google Brain Team with 5.8 million parameters (vit_tiny_patch16_384), a schematic of which is shown in Figure 14. The model is trained on ImageNet-21k dataset and fine-tuned on ImageNet-1k. Note that in subsection 5.3.1, we have only considered adapting the MLP head in Figure 14 since it is reasonable to assume that the transformer Encoder block in Figure 14 has learnt a good representation of the CIFAR-10 input data and it is only necessary to increase the capacity

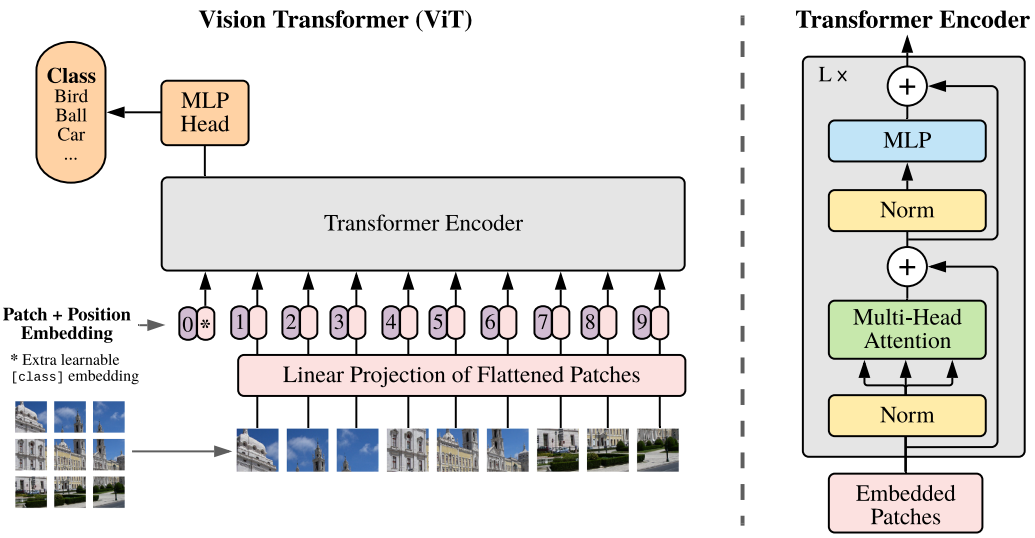


Figure 14. *Schematic of a vision transformer (ViT) image classification model (Dosovitskiy et al. [16])*

of the classifier (MLP Head) to further improve the classification accuracy. However, it is noteworthy that our approach may be extended to adapting MLP in transformer encoder blocks instead of the MLP Head. In this case one needs to make decisions on the following aspects: (i) out of different encoder blocks, which encoder block should be adapted; (ii) for the selected encoder block, where to add a new layer within the MLP; and (iii) how to initialize the added layer. This multi-level adaptation strategy is beyond the scope of the present work and will be investigated in the future.

Possibility of adapting the attention module in a transformer architecture. The attention mechanism in transformers relies on computing the Query ($\boldsymbol{Q}$), Key ($\boldsymbol{K}$), and Value ($\boldsymbol{V}$) matrices by applying three separate linear projections to an input sequence $\boldsymbol{X} \in \mathbb{R}^{N\times D}$, where N is the number of vectors in the sequence and D is the embedded dimension of the vector. Specifically, one computes $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_q$, $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_k$, $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_v$, where $\boldsymbol{W}_q$, $\boldsymbol{W}_k$, $\boldsymbol{W}_v \in \mathbb{R}^{D\times D}$ are the learnable parameters of the attention module. The output representation is then computed as: $\text{softmax}\left(\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{D}\right)\boldsymbol{V}$, where softmax(.) operation is applied row-wise, normalizing each row of the resulting matrix so that it sums to 1.

Since our approach is, after all, a sensitivity method, it can be applied to any architecture, including transformer, as long as a user-specific update strategy needs sensitivity. For transformer, one can derive the sensitivy of the loss function with respect to ($\boldsymbol{Q}$), ($\boldsymbol{K}$), and ($\boldsymbol{V}$). Together with the user-specific update choice, we can adapt ($\boldsymbol{Q}$), ($\boldsymbol{K}$), and ($\boldsymbol{V}$), respectively.

On the other hand, if one considers a nonlinear modification to the attention module such as modifying the Query as $\tilde{\mathbf{Q}} = f_{\boldsymbol{\theta}}(\mathbf{Q})$, where $f_{\boldsymbol{\theta}}(.)$ is an MLP with residual connections (with all layers having the same dimension), our proposed approach could potentially be used

to add additional layers to the MLP during the adaptation process. The procedure requires detailed investigation with user-specific choice of adaptivity.

Possibility of layers that change the dimensionality of the network features. Note that for the MLP in each encoder block of a ViT (see Figure 14), one could consider a propagation of the form $\boldsymbol{y}_1 = \boldsymbol{P}_1(\boldsymbol{Q}_1\boldsymbol{x} + \boldsymbol{g}(\boldsymbol{Q}_1\boldsymbol{x}))$, where $\boldsymbol{Q}_1$ is the up-sampling/downsampling in the input layer of MLP, $\boldsymbol{g}(.)$ is the nonlinear function acting componentwise on the input $\boldsymbol{Q}_1\boldsymbol{x} \in \mathbb{R}^k$, and $\boldsymbol{P}_1$ denotes the upsampling/downsampling layer (the output layer of MLP). For this two hidden layer MLP, it is clear that the hidden dimension is k. Now the output $\boldsymbol{y}_1$ can be used as an input to a new MLP block with hidden dimension l whose output is given as $\boldsymbol{y}_2 = \boldsymbol{P}_2(\boldsymbol{Q}_2\boldsymbol{y}_1 + \boldsymbol{g}(\boldsymbol{Q}_2\boldsymbol{y}_1))$, where $\boldsymbol{Q}_2\boldsymbol{x} \in \mathbb{R}^l$. Consequently, we obtain two MLP blocks with different hidden dimensions. Our approach could then be used to address the following questions: (i) which MLP block should be adapted; (ii) for the selected MLP block, where to add a new layer; and (iii) how to initialize the added layer. A detailed investigation of this multi-level adaptation strategy is left for future work.

D.7. Topological derivative informed transfer learning approach: Application in parameter efficient fine tuning. In order to further back our claim that topological derivative is indeed a good indicator for determining where to add a new layer (refer subsection 5.3.2), we consider adding a new layer at different locations and the results are tabulated in Table 8. We found that the computed topological derivative correlates well with the observed mean squared error and the best performance is achieved by adding a layer at the location of the highest topological derivative. Interestingly, we observe that retraining the first hidden layer yields the best results, contrary to traditional transfer learning, where the last few layers are typically retrained. Other details of hyperparameters used for the problem are provided in Appendix E.

Table 8
Correlation of topological derivative with mean squared error achieved by proposed method

Hidden layer number (Added layer)	Topological derivative	Mean squared error
1	**0.974**	**2.576**
3	0.941	2.716
5	0.907	3.120
7	0.851	3.280
8	**0.821**	**3.524**

Appendix E. Details of hyperparameter values for different problems. Details of hyperparameters used in Algorithm 2.1 and Algorithm A.1 is provided in Table 9. Table 9 additionally provides details on the hyperparameters used for the optimizer. The description of each problem is also provided below. In Table 9, b_s denotes the batch size, ℓ_r denotes the learning rate, and $\sigma_n = 0.01$ is chosen as the standard deviation of Gaussian noise for initializing the weights/biases. In order to reduce the number of parameters in the first layer, we introduce the sparsity parameter denoted as i_s. In our experiments, $i_s\%$ parameters in the first layer are initialized as zeros and the corresponding connections are removed (or those

Table 9
Details of hyperparameters for Algorithm 2.1

Problem	n	m	N_n	T_b	i_s (%)	E_e	κ_e	b_s	ℓ_r	ε^t	ε	τ_1
I (S=1000)	10	5	7	2	30 %	2000	1000	1000	0.001	0.01	0.001	0.001
I (S=1500)	10	5	7	2	30 %	2000	1000	1500	0.001	0.01	0.001	0.001
II (S=250)	20	10	6	2	25 %	2000	1000	125	0.001	0.01	0.001	0.001
II (S=500)	20	10	6	2	25 %	2000	1000	250	0.001	0.01	0.001	0.001
III (S=500)	20	10	6	2	25 %	500	500	500	0.01	0.01	0.001	0.001
III (S=1000)	20	10	6	2	25 %	500	500	1000	0.01	0.01	0.001	0.001
IV (S=1000)	20	10	6	2	10 %	500	500	1000	0.001	0.01	0.001	0.001
IV (S=5000)	20	10	6	2	10 %	500	500	5000	0.001	0.01	0.001	0.001
VI	10	5	20	0	0 %	1	1	512	0.001	0.01	0.001	0.1
VII	50	50	1	8	0 %	2000	0	20	0.001	0.01	0.001	0.001

parameters are untrainable/frozen throughout the procedure). Further, as more layers are added to capture more complex features in the solution, it is also natural to consider training the bigger network for a larger number of epochs. We consider the following scheduler for choosing the training epochs at each iteration, $E(i) = E_e + \kappa_e \times (i-1)$, where i denotes the i^{th} iteration of the algorithm. For Algorithm 3.1, we use the same hyperparameters in Table 9, Table 10 except that no scheduler is employed (i.e E_e, κ_e is not employed) and we do not fix the parameter m. Further, the parameter N_k in Algorithm 3.1 is chosen as 100 for all problems. For MNIST classification problem we fix N_k as 20.

Table 10
Details of hyperparameters for Algorithm A.1

Problem	n	m	N_n	T_b	f	s	E_e	κ_e	b_s	ℓ_r	ε^t	ε	τ_1
V (S=600)	10	5	6	2	5	1	15	0	600	0.001	0.01	0.0001	0.00001
V (S=60000)	10	10	7	6	5	1	5	0	600	0.001	0.01	0.0001	0.00001

I : Learning the observable to parameter map for 2D heat equation.
II : Learning the observable to parameter map for Navier Stokes equation.
III : Regression on California housing dataset.
IV : Wind velocity reconstruction problem.
V : Image classification on MNIST dataset with CNN architecture.
VI : Transfer learning for vision transformer (subsection 5.3.1).
VII : Transfer learning involving scientific data-set (subsection 5.3.2).