Engineering with Computers
https://doi.org/10.1007/s00366-022-01661-2

ORIGINAL ARTICLE

Domain adaptation based transfer learning approach for solving PDEs on complex geometries

Ayan Chakraborty[1] · Cosmin Anitescu[2] · Xiaoying Zhuang[1,3] · Timon Rabczuk[2]

Received: 28 August 2021 / Accepted: 5 April 2022

Abstract
In machine learning, if the training data is independently and identically distributed as the test data then a trained model can make an accurate predictions for new samples of data. Conventional machine learning has a strong dependence on massive amounts of training data which are domain specific to understand their latent patterns. In contrast, Domain adaptation and Transfer learning methods are sub-fields within machine learning that are concerned with solving the inescapable problem of insufficient training data by relaxing the domain dependence hypothesis. In this contribution, this issue has been addressed and by making a novel combination of both the methods we develop a computationally efficient and practical algorithm to solve boundary value problems based on nonlinear partial differential equations. We adopt a meshfree analysis framework to integrate the prevailing geometric modelling techniques based on NURBS and present an enhanced deep collocation approach that also plays an important role in the accuracy of solutions. We start with a brief introduction on how these methods expand upon this framework. We observe an excellent agreement between these methods and have shown that how fine-tuning a pre-trained network to a specialized domain may lead to an outstanding performance compare to the existing ones. As proof of concept, we illustrate the performance of our proposed model on several benchmark problems.

Keywords Transfer learning · Domain adaptation · NURBS geometry · Navier–Stokes equations

List of symbols

$\alpha \geq 0$	Learning rate
$\bar{S}$	A bounded set S that also contains its boundary
Γ_{CL}	Learning curve of the CL model
Γ_{TL}	Learning curve of the TL model
$\mathcal{C}^p(\mathbb{R}^n)$	Space of all continuously differentiable functions at least up to p times
$\zeta > 0$	Momentum
$N_{\text{bnd}} \in \mathbb{N}$	Total boundary training points
$N_{\text{int}} \in \mathbb{N}$	Total interior training points
$\text{Tol} > 0$	Tolerance

1 Introduction

Over the last few decades, Deep Neural Networks (DNNs) have perhaps witnessed the highest boom in large scale problems in various disciplines of science and engineering [1–5]. NNs have been around since the 1940s [6] and have been used in systematic applications. However, the recent success in deep learning is due to the combination of improved hardware resources such as GPUs and advanced theories starting with un-supervised pre-training and deep belief nets that have undergone rapid development from time to time in last few decades. Conventional machine learning (CL) is designed to function optimally under such assumptions that the training data and test data should belong to the same domain. It has been shown to be a versatile tool in capturing the complex pattern of different physical phenomena by using the acquired knowledge from a given set of input values. Despite its excellent performance in various domains, however, one major bottleneck lies in data acquisition which can be quite expensive, particularly in the course of analyzing complex engineering systems. In addition, there are many scenarios in the real world applications where collecting sufficient amounts of training data

✉ Timon Rabczuk
timon.rabczuk@uni-weimar.de

[1] Chair of Computational Science and Simulation Technology, Institute of Photonics, Faculty of Mathematics and Physics, Leibniz University, 30163 Hannover, Germany

[2] Institute of Structural Mechanics, Bauhaus University, 99423 Weimar, Germany

[3] Department of Geotechnical Engineering, College of Civil Engineering, Tongji University, Shanghai, China

Published online: 23 May 2022

manually and annotating from a specific domain may turn out to be intensive or laborious because of various reasons, such as the scarcity of data. Nevertheless, there may exist an abundance of similar data but with different distribution characteristics. Also intuitively, it is not always feasible to learn everything from scratch.

Transfer learning (TL) [7–10] is an inspiring methodology that utilizes the stored knowledge obtained from a source task and applies it to a new target task. The stark difference that isolates conventional learning from transfer learning is that in this approach one can leverage knowledge from pre-trained models for training newer models and also even for tackling the challenges of having limited data. Real world is messy and everytime we may not necessarily apply our model to a carefully constructed data set instead it may encounter different scenarios. Therefore, the goal is to achieve some transferable representations between source domain and target domain and build a concrete model up to an acceptable performance which in turn should be able to make proper predictions. The insight behind TL is that the model must be able to learn first how to behave in a task effectively and then generalize its gained knowledge as much as required to transfer and then apply it in a new domain.

The main aim of this paper is to facilitate TL through domain adaptation techniques, by extracting the common aspects between the source and target domains. If the input feature space between the source and target domains are same then this is referred to as homogeneous transfer learning and remains our focus throughout this paper. Some interesting transfer learning topics are reinforcement transfer learning [11], online transfer learning [12], lifelong transfer learning [13] and multi-task learning [14]. Domain adaptation [15, 16] is a widespread technique associated with TL seeking similar goals in machine learning paradigm and as it pertains here, the procedure is to adapt one or more source domains for the means of transferring "knowledge" to improve the performance of a target learner. This process attempts to alter a source domain in an attempt to bring the distribution of the target closer to the source. Our main objective is to learn and investigate such transformations that can map both source and target domains into a common feature space.

In this work, we explore a domain adaptation based transfer learning approach to approximate the solutions to partial differential equations (PDEs) in complex geometries. As mentioned earlier, our proposed method incorporates skills learned previously from source tasks to speed up learning on a new target task, while retaining the effectiveness of NNs. A fundamental requirement in this process is building a model that gives an excellent performance on the source tasks. The underlying models are layered architectures that learn different features at different layers. This is a more involved technique, where the initial layers are trained to capture more generic features and the high level final layers are tailored for the specific task at hand. In addition, we need to retrain (i.e., fine-tune) selectively some (or all) of the previous layers during the process. Past research [17–19] shows that the internal representations of DNNs learned from source data sets can be effectively used to solve a variety of tasks and one just needs to fine-tune the entire model with the target data set to reduce the domain shift errors. Interested readers are referred to [18, 20, 21] for a more detailed explanations on fine-tuning and how to get the work done more effectively. We note that the idea in this work is in some ways similar to that of model defeaturing used in finite element workflows, where a complex model is simplified to make it more tractable for obtaining a rough solution. However, in transfer learning, we directly use the simplified solution to more quickly perform analysis on the actual model.

We mention that in computational studies, several other techniques have also been used. For example, [22, 23] present a strong form-based meshfree point collocation method for mechanical contact between two bodies. In [24], a particle difference method for weak discontinuities in elliptic problems is presented. Moreover, in [25], a machine learning framework is developed to obtain a posteriori error estimates for multiple goal functionals employing the dual-weighted residual approach.

Our contributions in this work are as follows:

- We prove a theorem regarding the convergence of neural networks for a more general class of PDEs.
- By implementing domain adaptation techniques into transfer learning, on one hand we avoid expanding the huge resources required to train a data-hungry model, on the other hand we have developed a sophisticated algorithm that can carefully handle the singularities in the domain and achieve similar accuracy to the state-of-the-art adaptive refinement algorithms.
- We investigate the ability of optimizers with respect to their performance and successfully demonstrates how the modifications of few hyperparameters (for example learning rate) have a strong influence over the model architecture, which up to now, has not gained much attention.

The paper is structured as follows: In Sect. 2 a detailed description of the model problem, convergence theorem, discussion about the core architecture with implementation details and training algorithm is presented. Next in Sect. 3 we briefly reviews NURBS geometry and its importance to construct complex shapes and objects. In Sect. 4 several numerical tests showcasing the performance of the proposed

algorithm are provided. Finally, in Sect. 5, we draw the conclusions and present remarks on future extensions.

2 Neural network (NN) approximation for PDEs

2.1 Problem statement and methodology

The idea to approximate PDEs using NNs was first proposed by Dissanayake et al [26]. In this paper we will also provide a strong theoretical formulation regarding the approximation power of NN for some specific family of PDEs. Consider the family,

$$\left.\begin{aligned}\mathcal{L}[u(\boldsymbol{x})] &= h(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega \\ \mathcal{B}[u(\boldsymbol{x})] &= \psi(\boldsymbol{x}) \quad \boldsymbol{x} \in \gamma \subseteq \partial\Omega\end{aligned}\right\} \tag{1}$$

where $\mathcal{L}$ and $\mathcal{B}$ is a differential and boundary operator (quasi-linear or non-linear), respectively, domain of interest $\Omega \subset \mathbb{R}^n$ is bounded , boundary $\partial\Omega$ is at least Lipschitz continuous unless mentioned otherwise and γ is its subset on which the boundary conditions (BCs) are imposed. We intend to train the DNN approximate solution $U_\theta := U(\boldsymbol{x}, \boldsymbol{\theta})$ We proceed by approximating u and $\mathcal{L}$ using Deep Neural Networks (DNNs) $U_\theta = U(\mathbf{x}, \theta)$. Any prior knowledge on the exact analytical solution u is redundant. As previously mentioned, our approach adopts the deep collocation method, which assumes a certain discretization of the domain Ω and the boundary γ into a collection of points π_Ω and π_γ, respectively. The goal is to learn the parameters of NNs. These along with parameters of the operator $\mathcal{L}$ are learned by minimizing the error function under mean squared error (MSE) norm. For $\boldsymbol{x}_i \in \pi_\Omega$ and $\mathbf{s}_i \in \pi_\gamma$, we define:

$$\begin{aligned}\mathcal{E}[\theta] &:= \frac{1}{N_{\text{int}}} \sum_{i=1}^{N_{\text{int}}} |(\mathcal{L}\,[U_\theta] - h)(\boldsymbol{x}_i)|^2 \\ &\quad + \frac{1}{N_{\text{bnd}}} \sum_{i=1}^{N_{\text{bnd}}} |\mathcal{B}[u] - \psi(\mathbf{s}_i)|^2 \\ &= ||(\mathcal{L}\,[U_\theta] - h)(\boldsymbol{x})||^2_{\pi_\Omega} \quad + ||\mathcal{B}[u] - \psi(\boldsymbol{s})||^2_{\pi_\gamma}\end{aligned} \tag{2}$$

The points are uniformly sampled from the domain and the given boundary. Basically this equation is a Monte Carlo approximation of

$$\begin{aligned}\text{Err}[\theta] &:= \mathop{\mathbb{E}}_{x\sim U(\Omega)}\left[((\mathcal{L}\,[U_\theta] - h)(\boldsymbol{x}))^2\right] \\ &\quad + \mathop{\mathbb{E}}_{s\sim U(\gamma)}\left[(\mathcal{B}[u] - \psi(\mathbf{s}))^2\right]\end{aligned}$$

where $U(\Omega)$ and $U(\gamma)$ represent the uniform distribution over the domain and given boundary. This error function measures how accurately our model approximates the original solution and satisfies the boundary value problem (BVP) with respect to the defined norm. Our main objective is to construct such a neural network function U_θ by fine tuning the entire model so that $\mathcal{E}$ is as close to 0 as possible.

2.2 Model architecture: transfer Learning setup

DNN is comprised of multiple hidden layers and single input–output layers, where each layer is consisting of numerous neurons. For example if we have N neurons in the input layer, and M neurons in the output layer then the neural network is simply a mapping: $\mathbb{R}^N \mapsto \mathbb{R}^M$. The neurons in each layer are connected by weights-bias parameters and they are used to compute a weighted sum of the input neurons from the preceding layers to which they are connected. For example two adjacent layers are coupled as follows :

$$y_\ell(\boldsymbol{x}) := \sigma_\ell(\mathcal{W}_\ell y_{\ell-1}(\boldsymbol{x}) + \beta_\ell) \quad ; \ \ell = 1, 2, 3 \ldots$$

where y_ℓ is the output of layer ℓ, $\mathcal{W}_\ell$ are affine mappings and σ_ℓ is a fixed element wise activation function. The connection strength between neurons are solely dependent upon the weights and bias terms associated with these. The entire signal is now summed and used as an input for the layers activation function. The role of activation function is to introduce non linearity into the network. This is a crucial component for modeling nonlinear responses. The structure of the network can be complex. Once the network is fixed, the training is initiated and the task is to update the parameters appropriately. The network is trained using back-propagation which ultimately becomes the optimization problem of finding the parameters that minimize the loss function or the output errors. Gradually, as the parameters are adjusted, the network evolves and predicts the output with minimal errors. In short, this is the "learning pattern" of NN. In transfer learning, the goal is to store and access this previously gained knowledge from source data to reuse for second workflow. One needs to add a few high-level new layers see Fig. 1 on top of pre-trained fully connected layers in order to utilize the off-the-shelf representations from preceding deep layers. The core idea of transfer learning is inherent in the fact that neural networks are made up of layers which can be seen as interchangeable building blocks. Basically, in this approach we use well-trained, well-constructed models that have gained sufficient knowledge over the training process through larger or more generic data sets and apply them to boost the performance on smaller or more specific data sets. Fine-tuning begins, i.e, retraining of the entire model with a suitable learning rate and gradually the new layers learn to turn the old features into predictions on a target data set. In practice, differential learning rate a.k.a discriminative fine-tuning can be an effective strategy to customize the model. In this process, one may start with a fixed learning

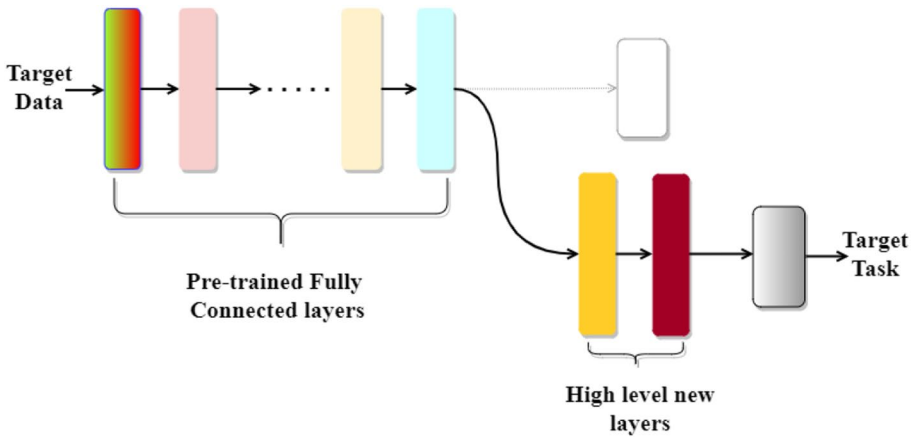

Fig. 1 Model architecture

rate $\alpha \in [0.001, 0.1]$ to instantiate the base or inception model with pre-trained weights and once the model starts to converge on source data, in the second workflow one needs to drop α preferably by a factor of $\kappa \in (2, 20)$ to avoid larger weight updates while embedding fine-tuning to the pretrained layers. In general, this is the guiding principle to update the parameters progressively from top-to-bottom to encourage the target model parameters to stay close to the parameters of pre-trained model. Freezing (when, $\alpha = 0$) has not been investigated in detail in the present work. For a relatively tiny neural network model this setting can be deemed robust. However, if the network is too large, for example ResNet, it is recommended to partition the entire network into small groups of layers and set different learning rates to each group during the training, otherwise the whole process would be very slow and memory intensive. Some excellent literature on this topic can be found in [27, 28].

2.2.1 Definitions

In their seminal paper [7] authors give an elegant mathematical definition of transfer learning. They introduce the notion of domain, task and marginal probabilities to present a framework for understanding transfer learning.

Definition 2.1 A domain consists of two components, a feature space $\mathcal{X}$ and a marginal probability distribution $\mathrm{P}(X)$, where $X = \{x_1, \ldots, x_k\} \in \mathcal{X}$. It is denoted as $\mathcal{D} = \{\mathcal{X}, \mathrm{P}(X)\}$.

Definition 2.2 Given a domain $\mathcal{D}$ define a task $\mathcal{T} = \{\mathcal{Y}, \mathrm{P}(Y)\}$ which consists of a label space $\mathcal{Y}$ and an objective predictive function $f_{(X,Y)}$ that can be learned from training data $\{(x_i, y_i)\} \subset X \times Y$.

Definition 2.3 Let $\mathcal{D}_S, \mathcal{D}_T$ be the source domain space and target domain space as well $\mathcal{T}_S, \mathcal{T}_T$ be the source learning task and targeted learning task, respectively, then Transfer Learning aims to improve the target predictive function f_T in $\mathcal{D}_T$ using its prior knowledge, i.e, from $\mathcal{D}_S, \mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$ and $\mathcal{T}_S \neq \mathcal{T}_T$

2.3 Implementation

The success of a neural network completely depends on its architecture. There is no exact formula for selecting an optimal architecture, and different problems demand different architectures. For example, activation functions are one of the core components and are used as gates to filter between "useful" and "not so useful" from the plethora of information. There exist numerous works in the literature [25, 29, 30] and the references therein, where authors have conducted comparative experiments to obtain the best possible results from an activation function. One thing to note is that the final layer should always be the linear function of its preceding layers. Apart from that, optimizers also have a significant influence for fast and easy convergence of the network. The optimizer shapes and molds the model into its most accurate possible form by updating the parameters. In our experiments, we have explored different optimizers, configuring the key hyper-parameters (momentum, learning rate etc.) to improve much as possible the accuracy of the model.

We now provide the implementation details of the algorithm and have highlighted some typical steps that one would have to follow for a Python based Tensorflow [31], an open source well documented and currently one of the most popular, fastest growing deep learning library. The idea is to demonstrate the general procedure. The full source code is available on Github.

Using the Xavier Initialization technique [32] the parameters are initialized in the following manner:

```
def initialize_NN(self,layers):
      weights = []
      biases = []
      num_layers = len(layers)
      for l in range(0, num_layers - 1):
          W = self.xavier_init(size=[layers[l], layers[l + 1]])
          b = tf.Variable(tf.zeros([1, layers[l + 1]]))
          weights.append(W)
          biases.append(b)
      return weights, biases

  def xavier_init(self, size):
      in_dim = size[0]
      out_dim = size[1]
      xavier_stddev = np.sqrt(2.0 / (in_dim + out_dim))
      return tf.Variable(tf.truncated_normal([in_dim, out_dim], stddev=
          ↪ xavier_stddev))
```

$u(\mathbf{x})$ is defined as follows:

$u(\mathbf{x})$ is defined as follows :

```
def net_u(self,x,y):

      X = tf.concat([x,y],1)

      u = self.neural_net(X,self.weights,self.biases)
```

The neural network is defined using an activation function:

The neural network is defined using an activation function :

```
     def neural_net(self,X,weights,biases):

      num_layers = len(weights) + 1

      H = 2.0*(X - self.lb)/(self.ub - self.lb) - 1.0
      for l in range(0,num_layers-2):
          W = weights[l]
          b = biases[l]
          H = tf.activation(tf.add(tf.matmul(H, W), b))
      W = weights[-1]
      b = biases[-1]
      Y = tf.add(tf.matmul(H, W), b)
      return Y
```

Define the base model and network learning:

Define the base model and network learning :

```
layers = [.,.,.,.] # say 4 layers
num_train_its = ... # number of training iterations
train_op = tf.optimizers(learning_rate1)
pred_model= PDE_model(layers, train_op, batch-size, num_epoch)
pred_model.network_learn(data1, num_train_its)
```

Transfer learning model and fine-tuning:

Transfer learning model and fine-tuning :

```
layers = [.,.,.,.,.,.,.] # additional high level layers on top of preceding
    ↪ layers
num_train_its = ...
train_op = tf.optimizers(learning_rate2)
pred_model= PDE_model(layers, train_op, batch-size, num_epoch)
     for layer in pred_model.layers[:k]:
        layer.trainable = False
     for layer in pred_model.layers[k:]:
        layer.trainable = True
pred_model.network_learn(data2, num_train_its)
```

2.4 Convergence result

Motivated by the results in [33, 34] we are also going to provide an existence theorem ensuring a feed forward multi-layer network U capable to approximate the solution of (1) along with optimizing (2). For the convenience sake we have assumed the existence of classical solution and consider,

$$\left.\begin{aligned} \mathcal{L}[u] &= \nabla \cdot \{\boldsymbol{\varphi}(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))\} + \eta(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x})) \\ \widehat{\partial}_r &\equiv \frac{\partial}{\partial u_{x_r}} \quad r = 1, 2, \ldots, n \\ \partial_s &\equiv \frac{\partial}{\partial x_s} \quad s = 1, 2, \ldots, n \\ \partial_{r,s} &\equiv \frac{\partial^2}{\partial x_r \partial x_s} \quad r, s = 1, 2, \ldots, n \end{aligned}\right\} \tag{3}$$

and $\mathcal{B}[\cdot]$ as a linear combinations of $\partial_x^{\boldsymbol{m}}$, where $|\boldsymbol{m}| \leq 2$. Therefore, (1) can re written as,

$$\underbrace{\sum_{r,s=1}^{n} \widehat{\partial}_s \varphi_r(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))\, \partial_{r,s} u}_{\mathcal{X}(\boldsymbol{x}, u, \nabla u)} + \underbrace{\sum_{r=1}^{n} \left[\frac{\partial}{\partial u} \varphi_r(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))\, \partial_r u + \partial_r \varphi_r(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))\right] + \eta(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))}_{\mathcal{Y}(\boldsymbol{x}, u, \nabla u)} = h(\boldsymbol{x})$$

In simplified form,

$$\mathcal{X}(\boldsymbol{x}) + \mathcal{Y}(\boldsymbol{x}) = h(\boldsymbol{x})$$

For any activation function σ (preferably "non-linear" and mandatorily "bounded") consider the set of NNs with a single hidden layer and ℓ neurons,

$$\mathfrak{M}_n^{\ell}(\sigma) := \{z : \mathbb{R}^n \mapsto \mathbb{R} \ : \ z(\boldsymbol{x}) = \sum_{i=1}^{\ell} \alpha_i \sigma(\boldsymbol{\omega} \cdot \boldsymbol{x} + \beta_i)\}$$

and the class of all functions implemented in such network for $\ell \in \mathbb{N}$

$$\mathfrak{M}_n(\sigma) := \bigcup_{\ell=1}^{\infty} \mathfrak{M}_n^{\ell}(\sigma)$$

Definition 2.4 A function $f : \mathbb{R}^i \times \mathbb{R}^j \mapsto \mathbb{R}$ is said to be Hölder Continuous if there exists a $\lambda \in (0, 1]$ and $M_f > 0$ such that,

$$|f(\boldsymbol{x}_i,\boldsymbol{x}_j)-f(\boldsymbol{y}_i,\boldsymbol{y}_j)| \leq M_f\left(||\boldsymbol{x}_i-\boldsymbol{y}_i||^{\lambda}+||\boldsymbol{x}_j-\boldsymbol{y}_j||^{\lambda}\right)$$

holds over $\mathbb{R}^i \times \mathbb{R}^j$. Here λ is called a Hölder exponent.

Theorem 2.1 *Let, $\sigma \in \mathcal{C}^p(\mathbb{R}^n)$ be a non constant bounded function then, $\mathfrak{M}_n(\sigma)$ is an uniformly-2 dense on compact sets of $\mathcal{C}^p(\mathbb{R}^n)$.*

Proof See [33] □

Theorem 2.2 *Let us assume that (3) has a unique classical solution over $\bar{\Omega}$ and both the non linear terms $\partial_s\varphi_r(\boldsymbol{x},u,\nabla u)$, $\mathcal{Y}(\boldsymbol{x},u,\nabla u)$ are Hölder continuous in $(u,\nabla u)$ uniformly w.r.t $\boldsymbol{x}$. Then for every $\epsilon>0$ there exists a neural network $f_\theta \in \mathfrak{M}_d(\sigma)$ such that,*

$$\mathcal{E}[f_\theta] \leq \mathrm{C}(u,\nabla_{\boldsymbol{x}}u)\,\epsilon^{2\lambda}$$

where λ is a Hölder exponent.

Proof We have from Theorem 2.1 an existence of an $f_\theta(=f) \in \mathfrak{M}_n(\sigma)$ that is uniformly-2 dense on compacts of $\mathcal{C}^2(\mathbb{R}^n)$. In other words it yields that for every $\epsilon>0$ there is a $f \in \mathfrak{M}_n(\sigma)$ such that,

$$\sum_{|\boldsymbol{m}|\leq 2}\sup_{\boldsymbol{x}\in\bar{\Omega}}|\partial_{\boldsymbol{x}}^{(\boldsymbol{m})}u(\boldsymbol{x})-\partial_{\boldsymbol{x}}^{(\boldsymbol{m})}f(\boldsymbol{x};\boldsymbol{\theta})|<\epsilon \tag{4}$$

Therefore, with the assumptions of Hölder continuity we obtain,

$$\begin{aligned}
&||\mathcal{Y}(\boldsymbol{x},f,\nabla_{\boldsymbol{x}}f)-\mathcal{Y}(\boldsymbol{x},u,\nabla_{\boldsymbol{x}}u)||^2_{\pi_\Omega}\\
&=\frac{1}{N_{int}}\sum_{i=1}^{N_{int}}\left|\mathcal{Y}(\boldsymbol{x}_i,f(\boldsymbol{x}_i),\nabla_{\boldsymbol{x}_i}f)-\mathcal{Y}(\boldsymbol{x}_i,u(\boldsymbol{x}_i),\nabla_{\boldsymbol{x}_i}u)\right|^2\\
&\leq\frac{\mathcal{L}}{N_{int}}\sum_{i=1}^{N_{int}}\left[|f(\boldsymbol{x}_i;\boldsymbol{\theta})-u(\boldsymbol{x}_i)|^{2\lambda}+|\nabla_{\boldsymbol{x}_i}f(\boldsymbol{x}_i;\boldsymbol{\theta})-\nabla_{\boldsymbol{x}_i}u(\boldsymbol{x}_i)|^{2\lambda}\right]\\
&\preceq\epsilon^{2\lambda}
\end{aligned} \tag{5}$$

where "$\preceq$" implies that the inequality is independent of any important constants. Again considering the expression,

$$\begin{aligned}
&||\mathcal{X}(\boldsymbol{x}_i,u(\boldsymbol{x}_i),\nabla_{\boldsymbol{x}_i}u)\mathcal{X}(\boldsymbol{x}_i,f(\boldsymbol{x}_i),\nabla_{\boldsymbol{x}_i}f)||^2_{\pi_\Omega}\\
&=\frac{1}{N_{int}}\sum_{i=1}^{N_{int}}\left|\sum_{r,s=1}^{n}\left(\frac{\partial}{\partial u_{x_s}}\varphi_r(\boldsymbol{x}_i,u(\boldsymbol{x}_i),\nabla u(\boldsymbol{x}_i))\,\partial_{r,s}u\right.\right.\\
&\quad\left.\left.-\frac{\partial}{\partial f_{x_s}}\varphi_r(\boldsymbol{x}_i,f(\boldsymbol{x}_i),\nabla f(\boldsymbol{x}_i))\,\partial_{r,s}f\right)\right|^2\\
&\leq\frac{1}{N_{int}}\sum_{i=1}^{N_{int}}\left|\sum_{r,s=1}^{n}\left(\frac{\partial}{\partial u_{x_s}}\varphi_r(\boldsymbol{x},u(\boldsymbol{x}),\nabla u(\boldsymbol{x}))\right.\right.\\
&\quad\left.\left.-\frac{\partial}{\partial f_{x_s}}\varphi_r(\boldsymbol{x},f(\boldsymbol{x}),\nabla f(\boldsymbol{x}))\right)\partial_{r,s}u\right|^2_{\boldsymbol{x}=\boldsymbol{x}_i}\\
&\quad+\frac{1}{N_{int}}\sum_{i=1}^{N_{int}}\left|\sum_{r,s=1}^{n}\frac{\partial}{\partial f_{x_s}}\varphi_r(\boldsymbol{x},f(\boldsymbol{x}),\nabla f(\boldsymbol{x}))\,(\partial_{r,s}u-\partial_{r,s}f)\right|^2_{\boldsymbol{x}=\boldsymbol{x}_i}
\end{aligned}$$

Applying Hölders Inequality with exponents p, q we have,

$$\begin{aligned}
&\preceq\frac{1}{N_{int}}\sum_{r,s=1}^{n}\left[\left(\sum_{i=1}^{N_{int}}|\partial_{r,s}u(\boldsymbol{x}_i)|^{2p}\right)^{1/p}\right.\\
&\quad\cdot\left(\sum_{i=1}^{N_{int}}\left|\frac{\partial}{\partial u_{x_s}}\varphi_r(\boldsymbol{x},u(\boldsymbol{x}),\nabla u(\boldsymbol{x}))-\frac{\partial}{\partial f_{x_s}}\varphi_r(\boldsymbol{x},f(\boldsymbol{x}),\nabla f(\boldsymbol{x}))\right|^{2q}_{\boldsymbol{x}=\boldsymbol{x}_i}\right)^{1/q}\\
&\quad+\left(\sum_{i=1}^{N_{int}}\left|\frac{\partial}{\partial f_{x_s}}\varphi_r(\boldsymbol{x}_i,f(\boldsymbol{x}_i),\nabla f(\boldsymbol{x}_i))\right|^{2p}\right)^{1/p}\\
&\quad\left.\cdot\left(\sum_{i=1}^{N_{int}}|\partial_{r,s}u-\partial_{r,s}f|^{2q}_{\boldsymbol{x}=\boldsymbol{x}_i}\right)^{1/q}\right]\\
&\preceq\frac{1}{N_{int}}\sum_{r,s=1}^{n}\left[\sum_{i=1}^{N_{int}}\left(|\partial_{r,s}u(\boldsymbol{x}_i)|^{2p}\right)^{1/p}\right.\\
&\quad\cdot\left(\sum_{i=1}^{N_{int}}\left(|f(\boldsymbol{x};\boldsymbol{\theta})-u(\boldsymbol{x})|^{\lambda}+|\nabla_{\boldsymbol{x}}f(\boldsymbol{x};\boldsymbol{\theta})-\nabla_{\boldsymbol{x}}u(\boldsymbol{x})|^{\lambda}\right)^{2q}_{\boldsymbol{x}=\boldsymbol{x}_i}\right)^{1/q}\\
&\quad+\left(\sum_{i=1}^{N_{int}}\left|\frac{\partial}{\partial f_{x_s}}\varphi_r(\boldsymbol{x}_i,f(\boldsymbol{x}_i),\nabla f(\boldsymbol{x}_i))\right|^{2p}\right)^{1/p}\\
&\quad\left.\cdot\left(\sum_{i=1}^{N_{int}}|\partial_{r,s}u-\partial_{r,s}f|^{2q}_{\boldsymbol{x}=\boldsymbol{x}_i}\right)^{1/q}\right]\\
&\preceq\epsilon^{2\lambda}
\end{aligned} \tag{6}$$

which finally obtain using (4) and simplifying. Now following (4)–(6), consequently the loss function can be simplified into,

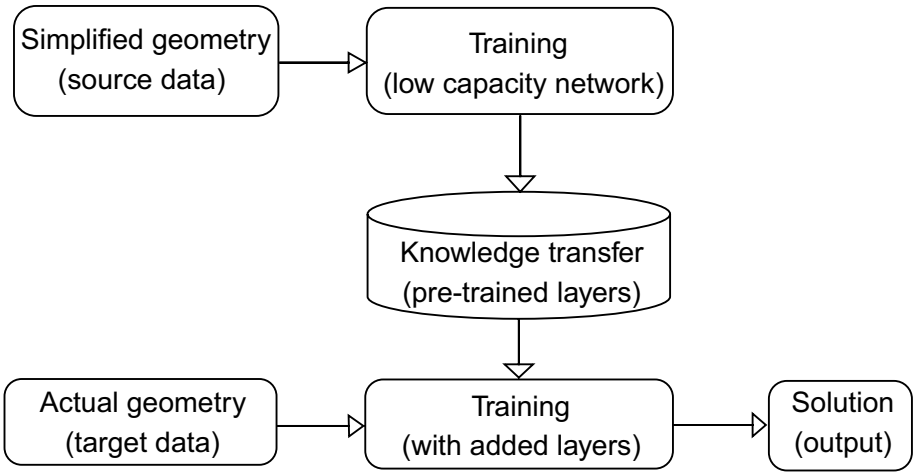

Fig. 2 Transfer learning applied to domain geometry adaptation

$$
\begin{aligned}
\mathcal{E}[f_\theta] &= ||\mathcal{L}[f_\theta] - h||^2_{\pi_\Omega} + ||\mathcal{B}[f_\theta] - \psi\ ||^2_{\pi_\gamma} \\
&= ||\mathcal{L}[f_\theta] - \mathcal{L}[u]\ ||^2_{\pi_\Omega} + ||\mathcal{B}[f_\theta] - \mathcal{B}[u]\ ||^2_{\pi_\gamma} \\
&\leq ||\mathcal{X}(\boldsymbol{x}, u, \nabla_{\boldsymbol{x}} u) - \mathcal{X}(\boldsymbol{x}, f_\theta, \nabla_{\boldsymbol{x}} f_\theta)||^2_{\pi_\Omega} \\
&\quad + ||\mathcal{Y}(\boldsymbol{x}, f_\theta, \nabla_{\boldsymbol{x}} f_\theta) - \mathcal{Y}(\boldsymbol{x}, u, \nabla_{\boldsymbol{x}} u)||^2_{\pi_\Omega} \\
&\quad + ||\mathcal{B}[f_\theta] - \mathcal{B}[u]\ ||^2_{\pi_\gamma} \\
&\leq \epsilon^{2\lambda} \\
&\leq \mathrm{C}\ \epsilon^{2\lambda}
\end{aligned}
$$

The last step is validated with an appropriate constant C that can be dependent upon the actual solution. □

Remark 1 Since $\bar{\Omega}$ is a compact subset in $\mathbb{R}^n$, therefore, above theorem trivially holds for locally Hölder continuous function.

2.5 Algorithm

A significant improvement over the performance of the base model, i.e, a positive transfer is an indication of a successful algorithm. The choice of pretraining and target tasks is intertwined closely and, therefore, for the sake of best target performance, it is beneficial to opt for similar pretraining tasks. The whole algorithm is tailored for an accurate and efficient approximation to obtain $\Gamma_{TL} < \Gamma_{CL}$. The general practice is to set a termination criteria by enforcing a Tol at the pretraining phase. Thus if the condition is not satisfied then this can serve as an error indicator which is employed to guide the parameters in the neural network architecture. The parameters get updated accordingly and this process continues iteratively. To the end summarizing all steps, we obtain the basic scheme shown in Fig. 2 and detailed in Algorithm 1.

Algorithm 1: Error estimation through transfer learning

Input: Training data sets, number of layers, α, ζ.

Initialize : Provide number of iteration steps, epochs and the model.

Initialize the neural network
Initialize the weights of the network using Xavier initialization technique
Calculate the loss function $\mathcal{E}[\theta]$ for the current mini-batch points

Pretraining : Train the model on source task
Minimize the loss up to *Tol*

Final Estimation :
Modify internal architecture
Jointly train newly added layers and layers of the base model
Evaluate model by observing the performance of Γ_{TL}

if $\Gamma_{TL} < \Gamma_{CL}$ **then**
    STOP
else
    tune the parameters, adjust *Tol* & repeat steps until the criteria is satisfied

3 NURBS-based geometrical modelling

In various applications of describing objects (2- or 3-D), NURBS (Non Uniform Rational B-Splines) approximation have become a standard mathematical tool to create and represent complex shapes regardless of whether freeform or an analytical surface. In general, they are widely used in the area of CAD geometry and because of their elegant algorithmic properties (smoothness, possibility of local modifications etc.) NURBS ideally offer designers the possibility to easily manipulate control points, weights, vertices, control curvature while generating a complicated geometry. In addition, they also provide a reasonably compact and intuitive representation for the construction. A thorough explanation is beyond the scope of this article, readers are referred to [35–40] for more in depth descriptions and applications.

NURBS are an extension of B-Splines to piecewise rational functions, which adds the ability to exactly represent some simple shapes, such as circles, ellipsoids, etc. In other words, NURBS are defined as a ratio of two polynomial B-Spline functions. Non uniform is the concept that some portion of a defined object can be elongated or shortened relative to other portion of overall shape. The geometric modeling typically consists of two meshes: a physical mesh and a control mesh. While the physical mesh is a representation of the original geometry, the control mesh forms a scaffold of the geometry. NURBS shapes are defined by the degrees, weights, knot vector and set of control points which are the inputs to be provided by the user. These determine a mapping between the parameter space, which is the unit segment for curves, unit square for surfaces and unit cube for volumes, and the physical space. A knot vector is an increasing set of coordinates in the parametric space generally denoted by $\Xi = \{\xi_k\}_{k=1}^{n+p+1}$, where ξ_k is the kth knot, n is the number of basis functions and p is the polynomial order. Depending on the degree of basis functions and the number of control points a knot can be repeated several times and it is even possible to insert a new knot without changing the curve geometry and parameterization, see Fig. 3, which is known as h-refinement. Knot refinement offers a wide range of tools to design and analyse the shape information. A univariate rational basis function is defined as :

$$\mathcal{R}_{k,p}(\xi) := \frac{w_k N_{k,p}(\xi)}{\sum_{k=1}^{n} w_i N_{k,p}(\xi)} \;;\; k \in [1, p+1]$$

Here, $\{N_{k,p}(\xi)\}_{k=1}^{n}$ is the set of basis functions of the B-Spline curves, $\{w_k \;:\; w_k > 0\}_1^n$ is the set of NURBS weight. Choosing appropriate weights permits the definition of varieties

of curves and in particular if all the weights are equal its reduces to the B-Spline basis. These basis functions are multiplied with a set of weights and control points and summed up to generate a NURBS geometry. The implementation is done using NURBS-Python (geomdl) library. Following snippet of code is an illustration how to generate a 2D NURBS curve and visualizing it using NURBS-Python [40].

```
from geomdl import NURBS

class Annulus(NURBS):
  """
  Class for defining an annular ring
  Input: radint,radext - interior and exterior radius of the ring
  """
  def __init__(self, radint, radext):
      geomData = dict()

      # Users set degrees
      geomData['degree_u'] = x_1
      geomData['degree_v'] = x_2

      # Users set control points
      geomData['ctrlpts_size_u'] = y_1
      geomData['ctrlpts_size_v'] = y_2

      geomData['ctrlpts'] = [...,...,...,.....]

      geomData['weights'] = [...,...,...,.....]

      # Users set knot vectors
      geomData['knotvector_u'] = [a_1,a_2,..,....]
      geomData['knotvector_v'] = [b_1,b_2,..,....]
      super().__init__(geomData)

# Plot the curve
Annulus.PlotSurf()
```

4 Numerical examples

In this section, based on several well known benchmark problems, we are presenting our experimental results for empirical validation. Each problem requires their own modified architecture depending on the operator and domain. The implementation details about the network architecture, such as number of layers, neurons on each layer, activation function etc, have been provided with each example. In some cases, the exact solution does exist and we have also validate the results showcasing the improvement in the performance achieved. In all cases, we have used a uniform distribution of the collocation points; however, the influence of the arrangement and spacing of collocation points reflects the algorithm performance which was discussed in detail in our earlier work [25]. Training neural networks consumes a lot of time and computational resources as well. Therefore, it is desirable to terminate the number of iterations once a certain level of accuracy has been reached. Two main factors reduce the number of iterations required for convergence. The first

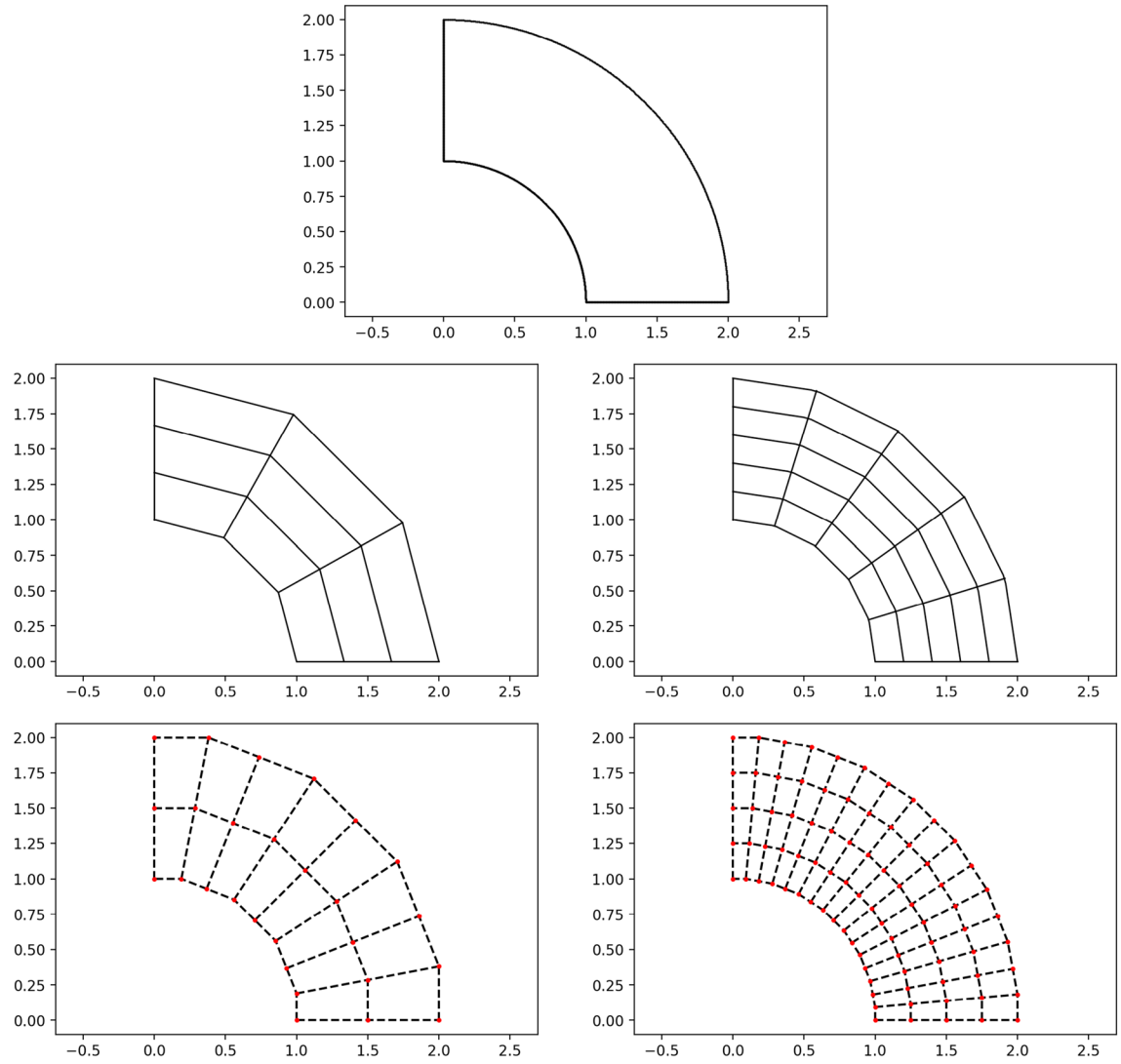

Fig. 3 (From top to bottom) Original geometry to be modeled. Modeling the geometry using linear splines. Modeling via NURBS, where the control points are marked by red

is to train the network using sufficient amount of data on a simpler, regular domain which saves a lot of computation time. The second is setting the number of hidden layers to be reasonably small, while still retaining sufficient accuracy. Besides, by leveraging pre-trained model architecture and parameters transfer learning allows to use the learned high level representation of a given data structure and apply it to fewer new training data set. For all the problems, we have used a combination of SGD and Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm to train the parameters $\boldsymbol{\theta}$

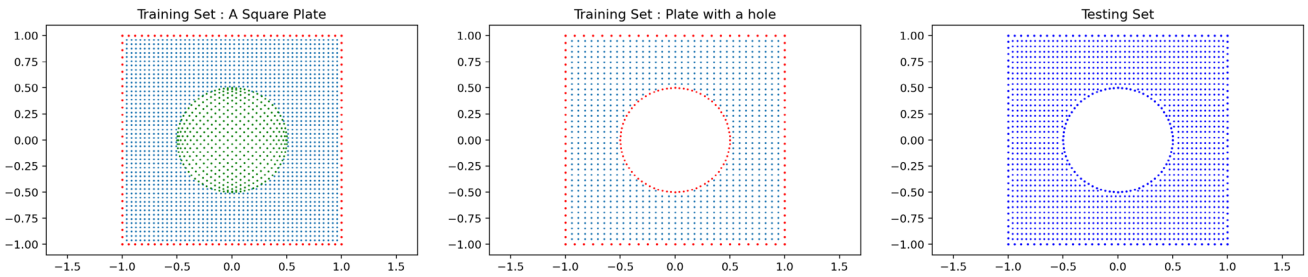

Fig. 4 Distribution of boundary and interior collocation points over training sets and test points over testing set

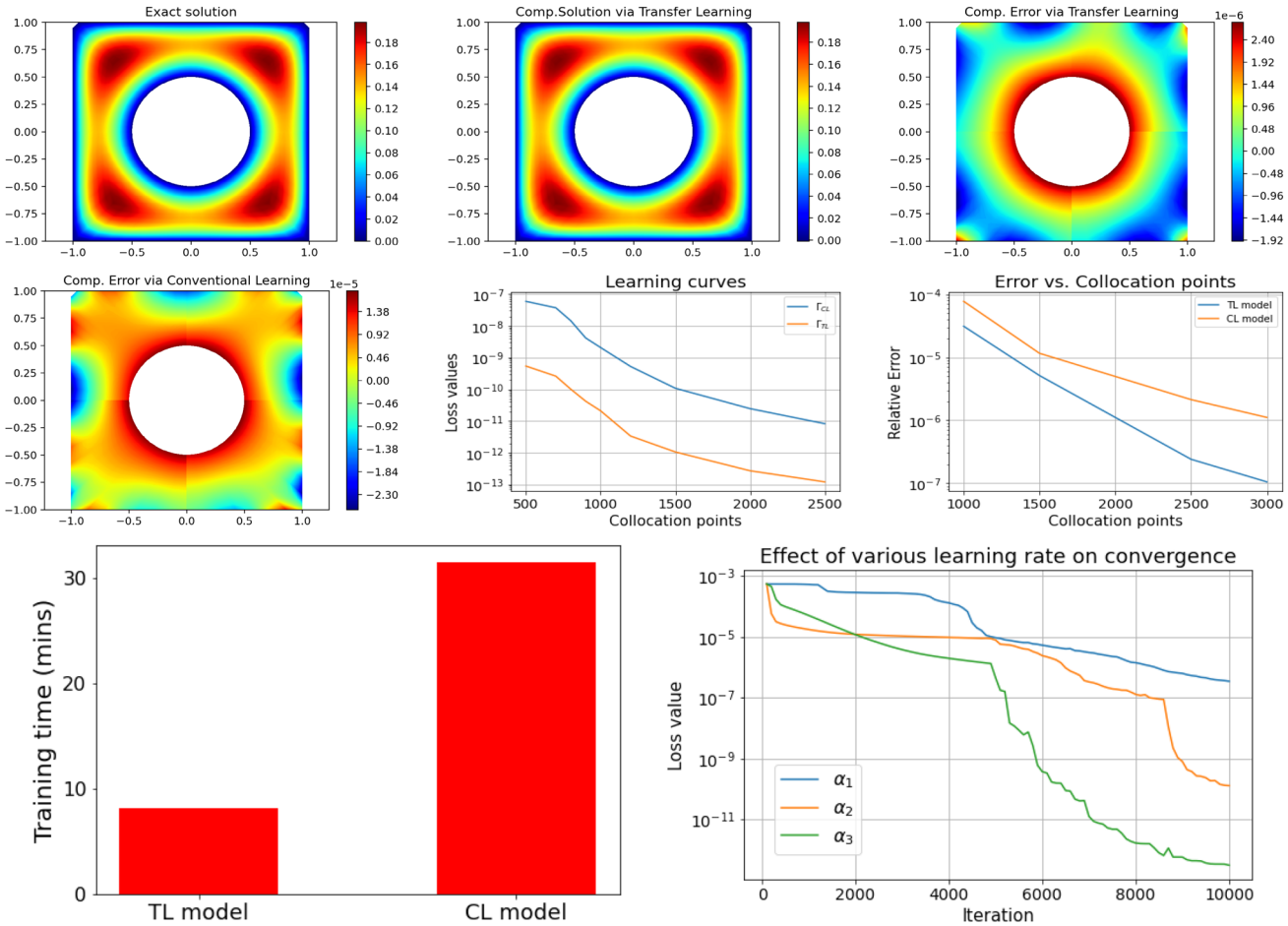

Fig. 5 Comparison of solutions and both pointwise and relative errors. Learning curves and training time. Comparison between different learning rates

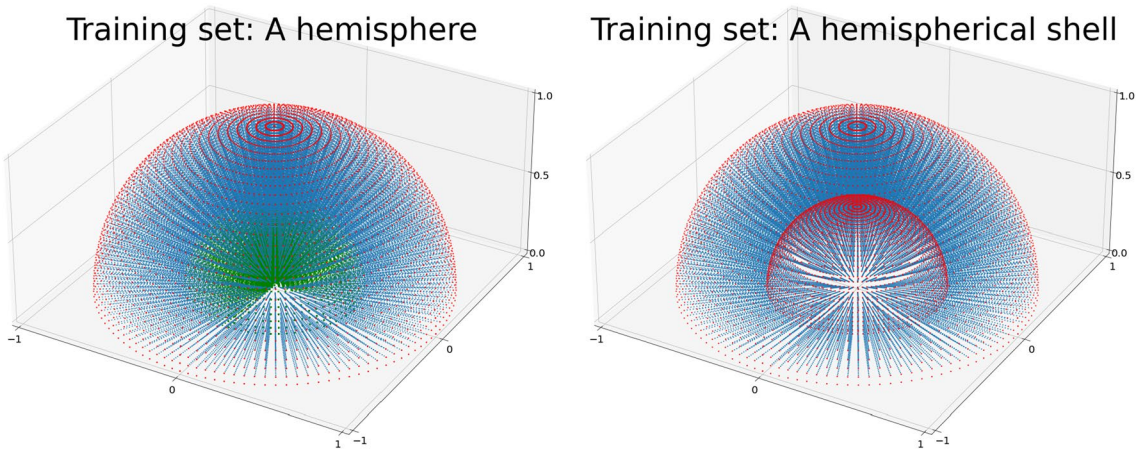

Fig. 6 Distribution of boundary (red) and interior (blue) and (green) collocation points over the training set

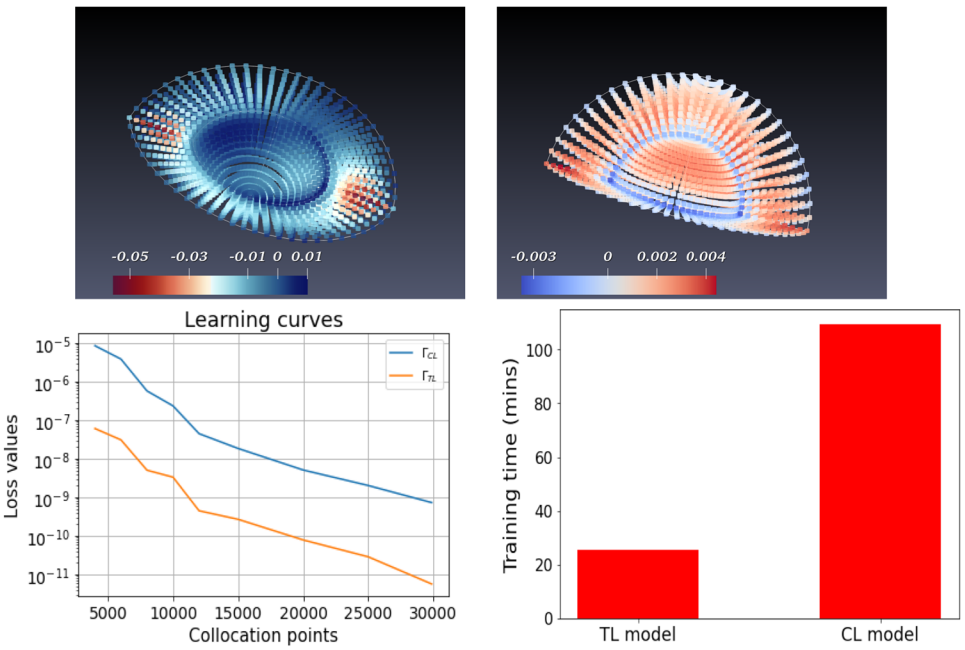

Fig. 7 Transfer learning solutions for $y > 0$ and $y \leq 0$. Comparisons of learning curves and training period

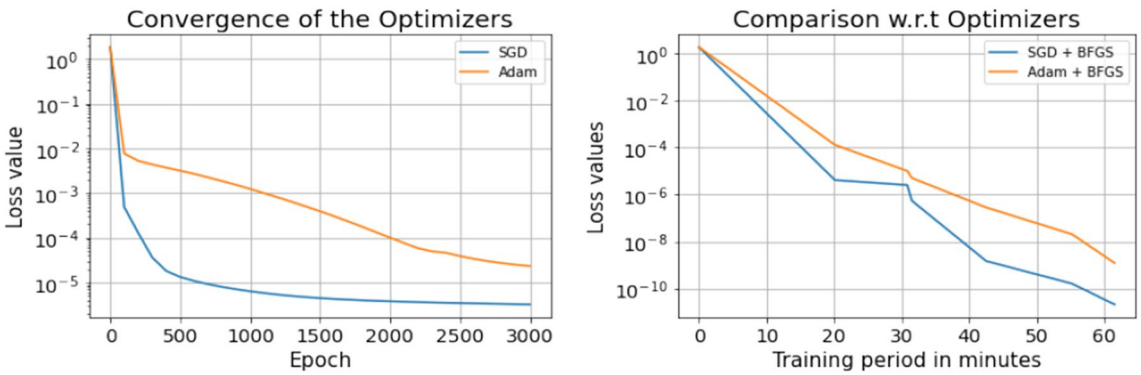

Fig. 8 Demo comparisons on a Hemisphere

following Xavier initialization technique. The training points are uniformly spaced over the entire domain Ω. The experiments have been carried out using the TensorFlow [31] framework on Google Colab GPU.

4.1 Example-I : 2D Poisson equation ($p = 2$, $\delta = 0$)

Consider,

$$\mathcal{A}_{\delta,p}(u) \equiv \text{div}((\delta^2 + |\nabla u|^2)^{\frac{p-2}{2}} \nabla u)$$

Define,

$$\Omega := \{(x, y) \in \mathbb{R}^2 \ : \ x^2 + y^2 > 1/4 \ \ \& \ \ |x|, |y| < 1\}$$

and the problem,

$$\begin{aligned} -\mathcal{A}_{\delta,p}(u) &= f(x, y) \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

To obtain the exact solution we take, for example

$$\mathcal{U}(x, y) := \begin{cases} \left(x^2 + y^2 - \frac{1}{4}\right)(1 - x^2)(1 - y^2) & \text{if } (x, y) \in \Omega \\ 0 & \text{otherwise} \end{cases}$$

and plug it into the problem to compute f. In the numerical model we consider the following domains : a square plate of edge length 2 and a similar length plate with a circular hole of half-unit radius at its center. We generate N uniformly distributed collocation points inside the domain and M uniformly distributed points on the boundary. Once the NNs have been trained, we can then evaluate on any number of points. Initially, the model is trained on a square plate which

effectively serve as an inception model. The training process on a square plate (i.e., the source task) is less expensive relative to the training process on a plate with hole (i.e., the target task). To relate these two tasks more closely and improve the learning, the source domain has been subdivided into two different data sets. The data set represented by blue dots is trained to minimize the loss function evaluated over Ω, while the data set represented by green dots is trained to match the boundary conditions. Once the learning process is invoked and the loss reaches a certain tolerance, we stop the process. The stopping criteria is again based on trial-error so that the algorithm never fails to converge. Now we utilize this previously constructed model architecture and most of the learned parameters, and then using standard training methods to learn the remaining, non-reused parameters of the new model corresponding to the plate with hole.

In the source task, we have used 3 hidden layers of 40 neurons each and the entire domain is discretized with $N \approx 5600$ interior, $M \approx 1800$ boundary training points as shown in Fig 4. On the target task, we have considered 2 more hidden layers on top of preceding trained layers, comprising of 35 neurons in each hidden layer, and the domain contains $N = 2000$ interior, $M = 500$ boundary training points. In both cases, the swish activation function has been used and SGD optimizer with $\alpha \in (0.0001, 0.1]$ and $\zeta \in (0.3, 1]$. In the source task, a relatively larger learning rate was used and once the layers have been trained to converge we then retrain the whole model end-to-end with layer specific learning rate. In particular, we set a moderate learning rate for the pretrained layers to avoid the time-cost. For the sake of experiment, we consider three different training schedules viz. "Lower learning rate", "Larger learning rate" and "Differential learning rate". We emphasize that all the schedules require the same amount of training iterations. From our findings, we conclude that larger learning rate puts the model under a higher risk of exploding gradients and failure to converge, while a lower learning rate consumes excessive amounts of time. Therefore, following standard practice, we attempt to predict the regime of learning rates, where the optimal performance can be achieved. In this phase we choose $\alpha_1 = 0.001$, $\alpha_2 = 0.008$ and $\alpha_3 = (0.0076, 0.08)$ and continue to use the same set of hyperparameters across tasks. The evolution plots are depicted in Fig 5. We find that for a fixed computational budget the best performance is always achieved in differential learning rate. Nevertheless there is a trade-off and one needs to go over trial-error process depending on the model. The error plots from a conventional learning and transfer learning are also depicted in Fig. 5. In addition, performance comparisons between a model trained from scratch (i.e., CL model) and a pretrained model (i.e., TL model) are demonstrated by learning curves. Since the exact solution is known, we also compare relative errors, which are defined as:

$$\mathcal{L}_2 := \frac{\sqrt{\sum_{i=1}^{\mathcal{N}_{\text{pred}}} \mathcal{U}^2_{\text{err}}(x_i, y_i)}}{\sqrt{\sum_{i=1}^{\mathcal{N}_{\text{pred}}} \mathcal{U}^2(x_i, y_i)}}$$

where $\mathcal{N}_{\text{pred}}$ is the number of testing points and $\mathcal{U}_{\text{err}}(x, y) = \mathcal{U}(x, y) - U_\theta(x, y)$. Together this is a clear cut evidence how transfer learning boosts the computation performance with smaller data set, possessing the benefits of less training time for a NN model and resulting in a lower generalization error in comparison to conventional learning, where the model needs to train from scratch on a larger data set to reach a desired level of accuracy.

4.1.1 Example-II: 3D Poisson equation

In the second example, we consider the Poisson problem with discontinuous right hand side on a three dimensional irregular domain defined by,

$$\Omega := \left\{ (x, y, z) \in \mathbb{R}^3 \ : \ \frac{1}{4} < x^2 + y^2 + z^2 < 1 \ \ \&, \ \ z \geq 0 \right\}$$

The governing equations are as follows:

$$-\mathcal{A}_{\delta,p}(u) = f(x, y, z) \quad \text{in } \Omega$$
$$u = 0 \quad \text{on } \partial\Omega$$

where

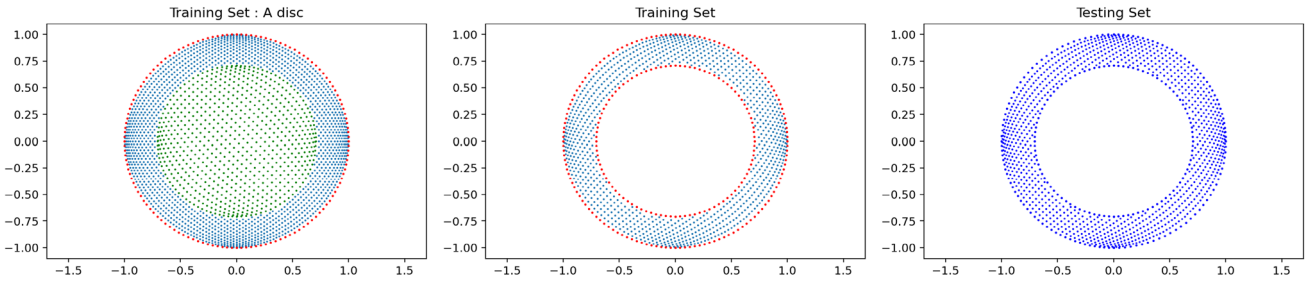

Fig. 9 Distribution of boundary and interior collocation points over training sets and test points over testing set

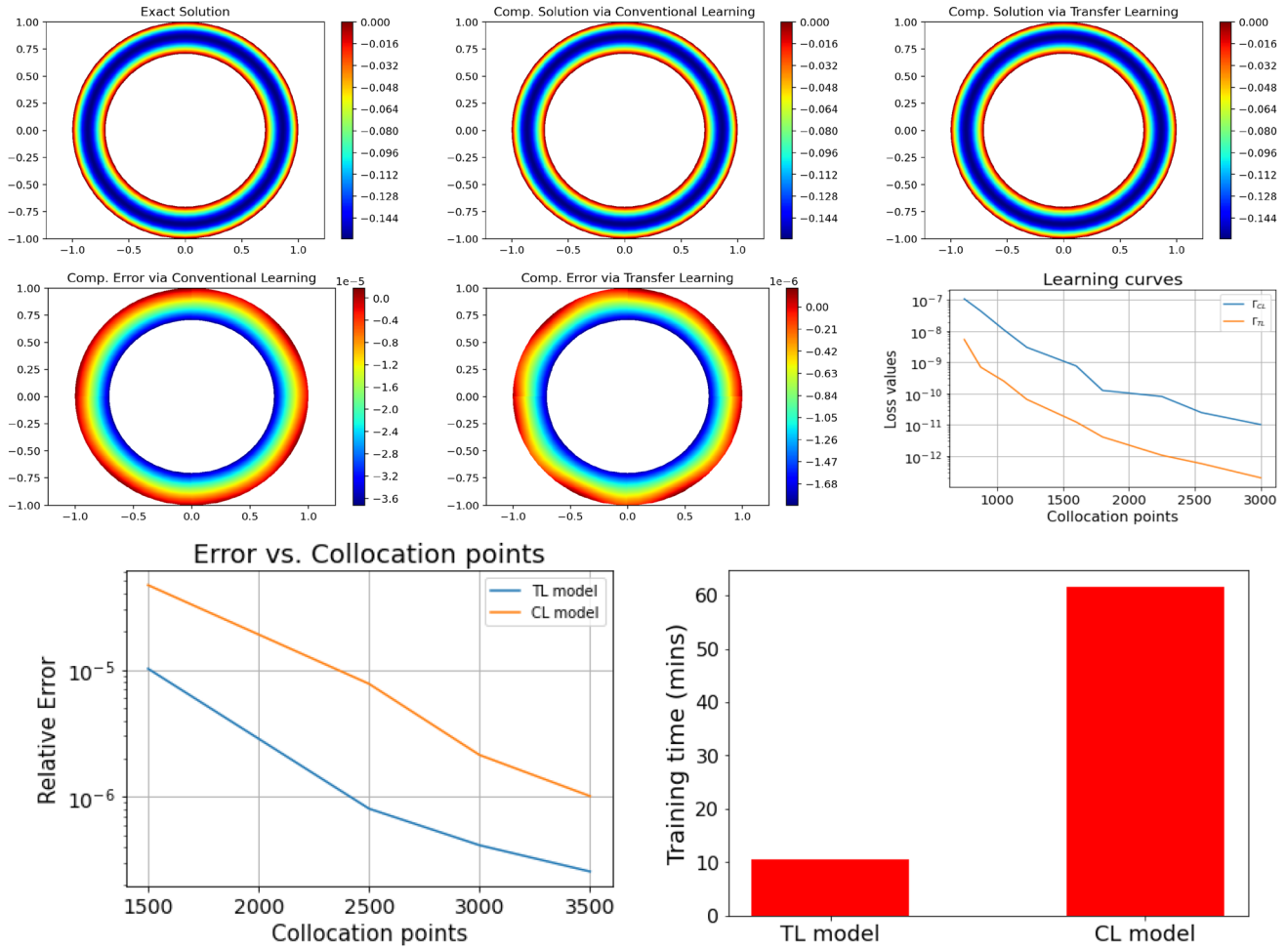

Fig. 10 Comparison of solutions and pointwise errors. Learning curves, relative errors and training duration

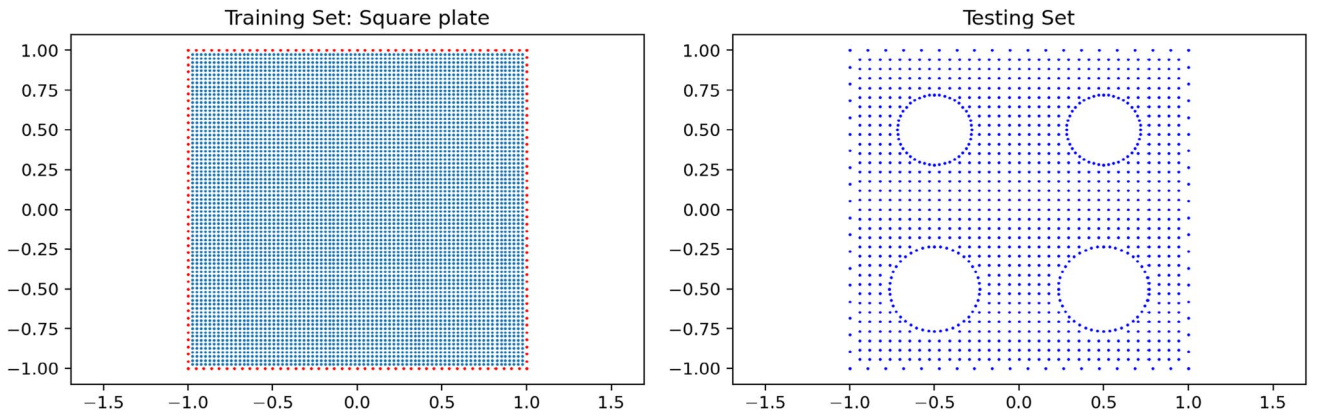

Fig. 11 Training set and evaluation set

$$f(x,y,z) = \begin{cases} \frac{\sqrt{x^2+y^2+z^2}}{6} & \text{if } y \leq 0 \\ x+y-\sqrt{z} & \text{if } y > 0 \end{cases}$$

The exact solution $\mathcal{U}$ is not known in this case; therefore, loss function is used as a measurement of how successful our model is at predicting the ground truth. In the source task, we have chosen a unit hemispherical domain filled with $N = 64000$ interior and $M = 15000$ boundary training points, see Fig. 6. The neural network model contains 4 hidden layers, each with 80 neurons and a combination of "swish-tanh" is chosen as the activation function. On the other hand, for the target task, we consider a hemispherical shell filled with $N \approx 22000$ interior and $M = 8000$ boundary training points and the network model is constructed with 2 more hidden layers of equal width of 50 neurons and "swish" activation function is applied. In addition, a fine tuned SGD optimizer followed by BFGS has been used for both the training process. The transfer learning solution in this case is shown in Fig. 7. As before, the interesting parts are the training time and loss values which are analyzed in the figures. As it can be seen in case of conventional learning, because of excessive training points on the same domain and complexity in the network model, the procedure is computationally expensive. In addition, the model ends up with worse loss values in comparison to transfer learning.

4.2 SGD vs Adam

More recently, researchers are focusing on SGD with momentum instead of vanilla SGD. With the goal of training faster and more accurate neural nets, our empirical results demonstrates that SGD + ζ converges much better than Adam. Despite its widespread popularity, under specific circumstances Adam sometimes fails to converge to an optimal solution. Exploratory studies [41–43] in this direction highlight the possible inabilities of adaptive optimization technique, such as Adam compared to SGD. We conducted an experiment with Adam and SGD. On the same problem setup, we train our model on the data set obtained from the hemispherical domain. We follow the same learning rate scheme. Figure 8 illustrates the efficacy of the respective optimizers. Evidently, this experiment suggests that SGD achieves a better accuracy compared to Adam, not only in faster decaying the loss values but also with respect to the training time and number of epochs necessary to attain that performance.

4.3 Example-III : $p = 1.5$, $\delta = 0.1$

In this example we consider p Laplacian problem with homogeneous Dirichlet condition and right hand side f is chosen to match the exact solution. However, the problem in this case has low regularity. This example is somewhat

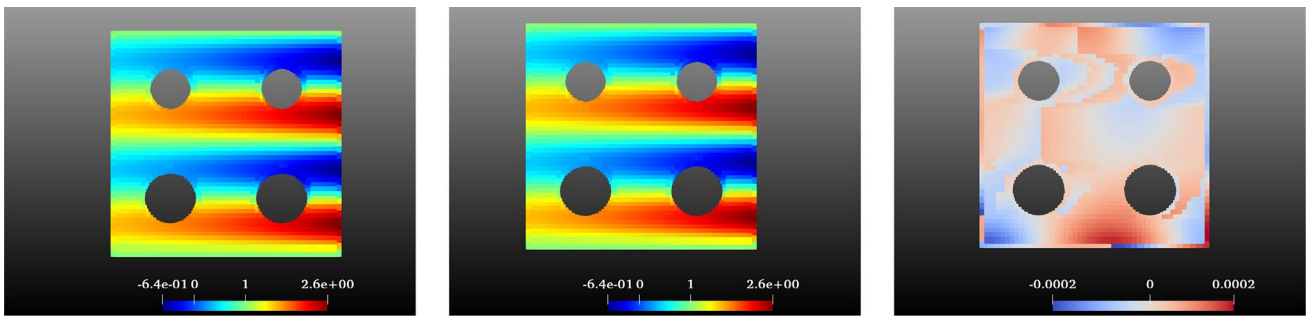

Fig. 12 (Left to right) Exact solution (U) , TL solution ($U_{\sim\theta}$) and pointwise error differences ($U - U_{\sim\theta}$)

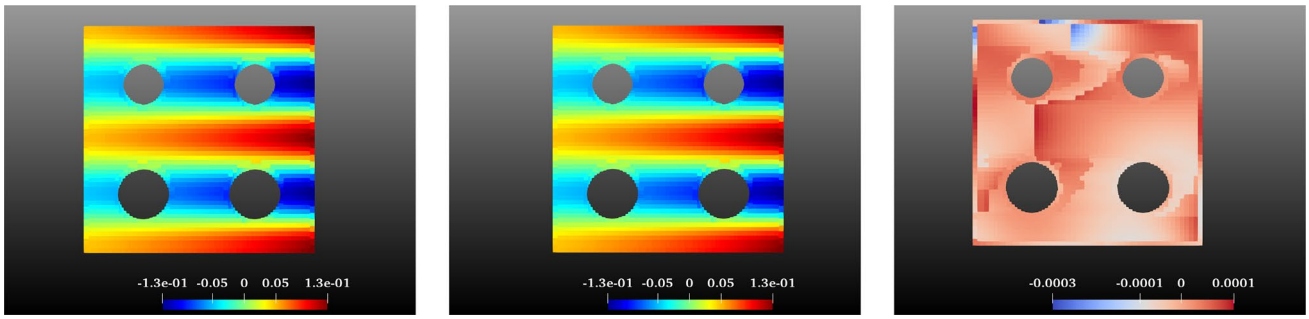

Fig. 13 (Left to right) Exact solution (V) , TL solution ($V_{\sim\theta}$) and pointwise error differences ($V - V_{\sim\theta}$)

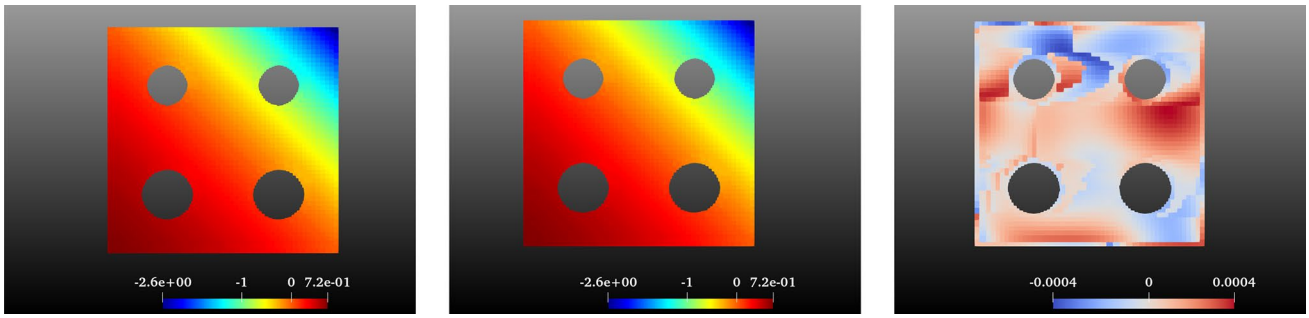

Fig. 14 (Left to right) Exact solution (P) , TL solution ($P_{\sim\theta}$) and pointwise error differences ($P - P_{\sim\theta}$)

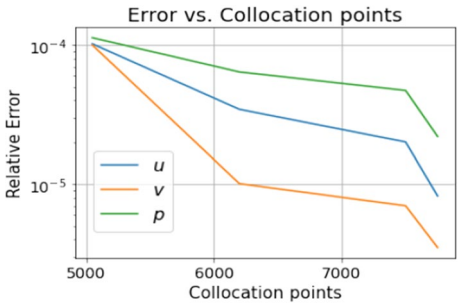

Fig. 15 Relative errors on test domain

motivated by an example in [44]. The degenerate nature of the p type problems makes the study of their regularity properties difficult, and in general, even with smooth problem data, high regularity for the solution u is not guaranteed. It has been shown in [45] that under some additional assumptions on the problem data, one can obtain a desired level of regularity which is sufficient to ensure an optimal convergence of the loss function. In this case, our domain is an annular ring,

$$\Omega := \left\{ (x, y) \in \mathbb{R}^2 \ : \ \frac{1}{2} < x^2 + y^2 < 1 \right\}$$

For the manufactured solution, we consider:

$$\mathcal{U}(x, y) := \begin{cases} \frac{1}{2\pi} \sin[2\pi(x^2 + y^2)] & \text{if } (x, y) \in \Omega \\ 0 & \text{otherwise} \end{cases}$$

Following previous techniques, the source task here is to train the model on a unit disc filled with $N = 6400$ interior points and $M \approx 2000$ boundary points, see Fig 9. We found for this model the hyperparameters : 3 hidden layers each with 50 neurons and "swish" activation were effective. Swish is bounded below but unbounded above and non-monotonic in nature; however, the evaluation cost per iteration is higher. Therefore, whenever possible, it should be combined with tanh or ReLU layers. This is what we also follow in our targeted task. Furthermore, here the data set contains $N \approx 2500$ interior points and $M = 900$ boundary points, we have 2 more hidden layers of equal width of 30 neurons, and as mentioned earlier SGD followed by BFGS is applied in both the training process. Finally, the comparison with respect to the exact solution and also loss function evaluation determines how well our algorithm models the data set. By taking a look at the figures, Fig. 10, constructed by our algorithm, one can conclude that even in the case for less regularized PDE and also involving a complex domain, the transfer learning approximation reaches an error closer to $\approx 10^{-7}$ while leading to a significant speedup in comparison to conventional learning.

4.4 Example-IV : Navier–Stokes equations

In the final example, we turn our attention to an important class of partial differential equation that describes the dynamics of an incompressible Newtonian fluid flow. Because of the regularity issues, here we have studied only the stationary version of the problem in two dimensional bounded domains. The domains are described in Fig. 11. To explore the effectiveness of task-to-task learning, in this particular example we have conducted the experiment of implementing the pretrained model directly without any target task specific modification. In general, for a better target performance, it is always beneficial to choose a similar pretraining task. Therefore, in our experiment, we consider the square plate as source domain consisting of uniformly distributed $N \approx 6700$ interior and $M = 1050$ boundary collocation points for training the source task model. We then use this pretrained model for initialization and evaluate its performance on a geometrically different target task model, i.e., plate with multiple holes.

The problem is motivated by an example in [46].

$$\begin{aligned} \mu \Delta \mathbf{v} = \mathbf{v} \cdot \nabla \mathbf{v} + \nabla p + \boldsymbol{\tau} \quad & \text{in } \Omega, \\ \nabla \cdot \mathbf{v} = 0 \quad & \text{in } \Omega, \\ \mathbf{v} = \boldsymbol{\varphi} \quad & \text{on } \partial\Omega. \end{aligned}$$

Here, $\mu = 0.01$ is the kinemetic viscosity coefficient, $\mathbf{v}(x, y) = [v_1(x, y)\ \ v_2(x, y)]$ & $p(x, y)$ are the Eulerian velocity and pressure fields. We choose the boundary conditions and τ so that the exact solution is given by,

$$v_1(x, y) = 1 + e^{\frac{x}{2}} \sin 2\pi y$$
$$v_2(x, y) = \frac{1}{4\pi} e^{\frac{x}{2}} \cos 2\pi y$$
$$p(x, y) = 1 - e^{\frac{2x+2y}{\pi}}$$

As mentioned earlier, our goal here is to train the model on a simpler regular domain, e.g., square plate and then we are generalizing the gained knowledge by applying this model on the targeted domain, e.g., plate with multiple holes of different dimensions. We avoid the training process on such domain which is quite challenging. Nevertheless, transfer learning allows to deal with these scenarios by leveraging the pre-built model from source task. The basic architecture consists of 3-hidden layer neural network containing 60 neurons in each layer. We have trained one neural network with 3 outputs but, it can also be done individually to approximate $\mathbf{u}$ and p. This seems to be much harder, because training different architectures and attaining optimal hyperparameters is itself a daunting task and it also requires a lot of computational time. Besides, the former task is more suited for grasping the underlying equations. The weights are initialized using a Xavier initialization, while the biases are generated using a normal distribution with mean 0 and standard deviation 1. They are trained using an SGD optimizer with the fine tuned learning rate and momentum followed by BFGS. The first two hidden layers are connected with "swish" activation and the final layer is with "tanh" activation. The resulting prediction error is validated against the test data. Figures 12, 13, 14 provide a comparison between the exact solution and the transfer learning outcome. It can be observed that the predicted outputs and exact solutions are quite close on the testing sets, which evidently supports the success of the pretrained model over the targeted task. Therefore, from these empirical findings, we conclude that even without learning every task specific features, TL implementations are still adaptable enough to achieve reasonable accuracy with only a pretrained model. Thus by learning quite generic feature of the target domain, the model is capable of capturing the intricate non linear behavior of N–S equation in that domain. Moreover, as an addendum, we have achieved such performances with zero training duration on target model. To quantify the accuracy of this novel approach we also compute the relative error as:

$$\mathcal{L}_2^u := \frac{\sqrt{\sum_{i=1}^{\mathcal{N}_{\text{pred}}} \mathcal{U}_{\text{err}}^2(x_i, y_i)}}{\sqrt{\sum_{i=1}^{\mathcal{N}_{\text{pred}}} \mathcal{U}^2(x_i, y_i)}}$$

where $\mathcal{N}_{\text{pred}}$ is the no. of testing points and $\mathcal{U}_{\text{err}}(x, y) = \mathcal{U}(x, y) - U_{\sim\theta}(x, y)$. Similarly, $\mathcal{L}_2^v$ and $\mathcal{L}_2^p$ are also defined. To the end an experimental assessment of performance is demonstrated in Fig. 15.

5 Conclusions

In this paper, we proposed a novel approach to predict the solutions of complex BVPs while capturing the features of interest in the input domain using the mechanisms and strategies of transfer learning from the perspectives of data and model. We have shown how the proposed framework is superior in various aspects to existing protocols. Our findings enable the efficient use of optimizers and analyse how to control the learning rate and momentum coefficients in order to achieve near-identical model performance on the test set with the same number of training iterations but significantly fewer parameter updates. Furthermore, the implementation of NURBS based modeling possesses profound computational benefits of designing the shape of domain, since the time taken to analyze is greatly reduced and consequently our results have high accuracy even in the case of more sophisticated shapes of the boundary. Towards this end, experiments have been conducted to evaluate the performance of the proposed model to handle the mainstream area in domain adaptation algorithm. The comparisons of different models clearly reflect that selection of transfer learning model is an important research topic when solving complex problems for practical applications. Several techniques remain open to explore and a wider range of new approaches are still require, to solve the knowledge transfer problems in more complex scenarios for example how to tackle negative transfer learning [47, 48], catastrophic forgetting [49, 50] etc, would be our future direction to study extensively. It is also left to mention that, as one of the popular and promising subject in machine learning algorithm optimization method remains a major bottleneck and deserves further systematic analysis. Recent research [51–53] shows that this gap can be eliminated by careful use of classical momentum or Nesterov accelerated gradient based techniques. Some other commonly employed technique includes time based decay learning rate and adaptive learning rate. Studying this area is also a subject in our future work. Finally, this method could be applied to other problems, where domain adaptation is important, such as classifying whole-slide images in medical applications.

Acknowledgements AC acknowledge the support of the ERC Starting Grant no: 802205.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Hinton GE, Osindero S, Teh Y-W (2006) A fast learning algorithm for deep belief nets. Neural Comput 18(7):1527–1554
2. Nair V, Hinton GE (2010) Rectified linear units improve restricted boltzmann machines. In: Icml
3. Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov RR, Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580
4. Goodfellow I, Warde-Farley D, Mirza M, Courville A, Bengio Y (2013) Maxout networks. In: International conference on machine learning, PMLR, pp 1319–1327
5. Agostinelli F, Hoffman M, Sadowski P, Baldi P, Learning activation functions to improve deep neural networks. arXiv preprint arXiv:1412.6830
6. McCulloch WS, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. Bull Math Biophys 5(4):115–133
7. Pan SJ, Yang Q (2009) A survey on transfer learning. IEEE Trans Knowl Data Eng 22(10):1345–1359
8. Weiss KR, Khoshgoftaar TM (2016) An investigation of transfer learning and traditional machine learning algorithms. In: 2016 IEEE 28th international conference on tools with artificial intelligence (ICTAI), IEEE, pp 283–290
9. Goswami S, Anitescu C, Chakraborty S, Rabczuk T (2020) Transfer learning enhanced physics informed neural network for phase-field modeling of fracture. Theoret Appl Fract Mech 106:102447
10. Chakraborty S (2021) Transfer learning based multi-fidelity physics informed deep neural network. J Comput Phys 426:109942
11. Taylor ME, Stone P, Transfer learning for reinforcement learning domains: a survey. J Mach Learn Res 10(7)
12. Ge L, Gao J, Zhang A (2013) Oms-tl: a framework of online multiple source transfer learning. In: Proceedings of the 22nd ACM international conference on information & knowledge management, pp 2423–2428
13. Ammar HB, Eaton E, Luna JM, Ruvolo P (2015) Autonomous cross-domain knowledge transfer in lifelong policy gradient reinforcement learning. In: Twenty-fourth international joint conference on artificial intelligence
14. Zhang Y, Yang Q (2018) An overview of multi-task learning. Natl Sci Rev 5(1):30–43
15. Shoeleh F, Yadollahi MM, Asadpour M (2020) Domain adaptation-based transfer learning using adversarial networks. Knowl Eng Rev 35:e7
16. Kouw WM, Loog M, An introduction to domain adaptation and transfer learning. arXiv preprint arXiv:1812.11806
17. Magill M, Qureshi F, de Haan H (2018) Neural networks trained to solve differential equations learn general representations. In: Advances in neural information processing systems, pp 4071–4081
18. Yosinski J, Clune J, Bengio Y, Lipson H (2014) How transferable are features in deep neural networks? In: Advances in neural information processing systems, pp 3320–3328
19. Lu J, Behbood V, Hao P, Zuo H, Xue S, Zhang G (2015) Transfer learning using computational intelligence: a survey. Knowl-Based Syst 80:14–23
20. Zhuang Z, Tan M, Zhuang B, Liu J, Guo Y, Wu Q, Huang J, Zhu J, Discrimination-aware channel pruning for deep neural networks. arXiv preprint arXiv:1810.11809
21. Yim J, Joo D, Bae J, Kim J (2017) A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 4133–4141
22. Schaefferkoetter P, Michopoulos JG, Song JH (2021) Strong-form meshfree collocation method for non-equilibrium solidification of multi-component alloy. Eng Comput:1–15
23. Beel A, Song J-H (2021) Strong-form meshfree collocation method for multibody thermomechanical contact. Eng Comput:1–20
24. Yoon Y-C, Song J-H (2021) Interface immersed particle difference method for weak discontinuity in elliptic boundary value problems. Comput Methods Appl Mech Eng 375:113650
25. Chakraborty A, Wick T, Zhuang X, Rabczuk T (2021) Multigoal-oriented dual-weighted-residual error estimation using deep neural networks. arXiv e-prints arXiv–2112
26. Dissanayake M, Phan-Thien N (1994) Neural-network-based approximations for solving partial differential equations. Commun Numer Methods Eng 10(3):195–201
27. Howard J, Gugger S (2020) Fastai: a layered api for deep learning. Information 11(2):108
28. Howard J, Ruder S, Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146
29. Saha S, Nagaraj N, Mathur A, Yedida R, Sneha H (2020) Evolution of novel activation functions in neural network training for astronomy data: habitability classification of exoplanets. Eur Phys J Spec Top 229(16):2629–2738
30. Samaniego E, Anitescu C, Goswami S, Nguyen-Thanh VM, Guo H, Hamdia K, Zhuang X, Rabczuk T (2020) An energy approach to the solution of partial differential equations in computational mechanics via machine learning: concepts, implementation and applications. Comput Methods Appl Mech Eng 362:112790
31. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), pp 265–283
32. Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics, JMLR Workshop and Conference Proceedings, pp 249–256
33. Hornik K (1991) Approximation capabilities of multilayer feedforward networks. Neural Netw 4(2):251–257
34. Sirignano J, Spiliopoulos K (2018) Dgm: a deep learning algorithm for solving partial differential equations. J Comput Phys 375:1339–1364
35. Hughes TJ, Cottrell JA, Bazilevs Y (2005) Isogeometric analysis: cad, finite elements, nurbs, exact geometry and mesh refinement. Comput Methods Appl Mech Eng 194(39–41):4135–4195
36. Cottrell JA, Hughes TJ, Bazilevs Y (2009) Isogeometric analysis: toward integration of CAD and FEA. Wiley, New York
37. Piegl L, Tiller W (1996) The NURBS book. Springer, Berlin
38. Nguyen VP, Anitescu C, Bordas SP, Rabczuk T (2015) Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simul 117:89–116
39. Dimas E, Briassoulis D (1999) 3d geometric modelling based on nurbs: a review. Adv Eng Softw 30(9–11):741–751
40. Bingol OR, Krishnamurthy A (2019) Nurbs-python: an open-source object-oriented nurbs modeling framework in python. SoftwareX 9:85–94

41. Wilson AC, Roelofs R, Stern M, Srebro N, Recht B, The marginal value of adaptive gradient methods in machine learning. arXiv preprint arXiv:1705.08292
42. Keskar NS, Socher R, Improving generalization performance by switching from adam to sgd. arXiv preprint arXiv:1712.07628
43. Reddi SJ, Kale S, Kumar S, On the convergence of adam and beyond. arXiv preprint arXiv:1904.09237
44. Toulopoulos I, Wick T (2017) Numerical methods for power-law diffusion problems. SIAM J Sci Comput 39(3):A681–A710
45. Liu W, Barrett JW (1993) A remark on the regularity of the solutions of the p-laplacian and its application to their finite element approximation. J Math Anal Appl 178(2):470–487
46. Galdi G (2011) An introduction to the mathematical theory of the Navier-Stokes equations: steady-state problems. Springer, Berlin
47. Zhang W, Deng L, Wu D, Overcoming negative transfer: a survey. arXiv preprint arXiv:2009.00909
48. Wang Z, Dai Z, Póczos B, Carbonell J (2019) Characterizing and avoiding negative transfer. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 11293–11302
49. Chen X, Wang S, Fu B, Long M, Wang J, Catastrophic forgetting meets negative transfer: batch spectral shrinkage for safe transfer learning
50. Kirkpatrick J, Pascanu R, Rabinowitz N, Veness J, Desjardins G, Rusu AA, Milan K, Quan J, Ramalho T, Grabska-Barwinska A et al (2017) Overcoming catastrophic forgetting in neural networks. Proc Natl Acad Sci 114(13):3521–3526
51. Sutskever I, Martens J, Dahl G, Hinton G (2013) On the importance of initialization and momentum in deep learning. In: International conference on machine learning, PMLR, pp 1139–1147
52. Zhou K, Jin Y, Ding Q, Cheng J (2020) Amortized Nesterov's momentum: a robust momentum and its application to deep learning. In: Conference on uncertainty in artificial intelligence, PMLR, pp 211–220
53. Liu C, Belkin M (2019) Accelerating sgd with momentum for over-parameterized learning. In: International conference on learning representations