Deep operator learning: a talk for the seminar ”Scientific Machine Learning”

M. Malvestiti

CONTENTS

ABSTRACT

Deep operator learning is an innovative idea, inspired by the desire of simplifying the solution of ODEs and PDEs both in complexity and computational cost.
Its potential is enormous and it’s finding many applications in both the scientific and engineering realm.
In my talk I introduce the audience to the base concepts of the architecture, with careful attention to details and applications. At the same time, I don’t limit myself to a bare explanation, but I rather pose questions and reflections on the bright and dark sides of this technique.
With a careful division between reflections and actual pros and cons, I share with the reader part of my thought process and the critical thinking I adopted while studying the topic and attempting a replication.

1 Introduction

Operator learning was a concept introduced to me by Prof. Alfio Maria Quarteroni, in Politecnico di Milano, during one of the final lectures of the course ”Numerical Analysis of Partial Differential Equations”. The ground idea is incredible: taking away the effective but also pedantic and clunky machinery of numerical methods and replace it with an I/O black box, that will learn to reproduce the same computation with a mere fraction of the computational cost. It was presented as a magical tool to use after development, when many simulations have already been run and they can serve as a training basis. For that same course I did a long project with FEMs. Simulations took easily up to two hours. This is very time consuming in the latter phase of the work, especially when changes are made only to parameters and mathematical functions. Since then I’d been dreaming of having knowledge about this tool. This is why I put myself forward for this topic and why I approached it with great interest.

In sections 2 and 3, as well as in the talk, I present a very didactic explanation, but that is not the end, as I didn’t just want to scratch the surface. Instead I adopted a critical approach, mindful of possible implementations and questioning its effectiveness, strengths and weaknesses in multiple scenarios. This is the heart of the report and it’s written in section 5. Of course, in the extent of what I could manage, I tried to simulate those scenarios and get a practical answer by coding. You can read about this in section 4. This led me to make very interesting discoveries and to have a realistic picture of the state of this technology, which I tried to convey with all my passion during the 1h15min long talk on the 27th of November 2023.

Finally, in the following report I will try not only to summarise the topic, but also to motivate the choices behind the structure of the talk itself.

2 Theoretical standpoint

My talk was opened with the following question: *Which operator?* [1]

I had a misconception myself before studying the topic, believing that the network would learn the differential operator that characterises the differential equation. However this is not the starting point for [1], the original article in which this architecture was introduced. Let’s start from a general PDE:

$$\begin{cases} L(h(x)) = f(x) \\ +BC \end{cases} \tag{1}$$

[1] I will assume that the reader is well acquainted with the notions of function, operator and functional.

The network will not try to learn the differential operator L, but rather an operator G such that:

$$\begin{aligned} G\colon X &\longrightarrow Y \\ u &\longmapsto s = G(u) \end{aligned} \tag{2}$$

So, the operator is rather specific: it characterises the mapping from a single function u, which could be for example the dumping term of a pendulum or the forcing term of any kind of PDE, to the solution s. It must represent a one to one, function to function transformation.

It's interesting to notice that the operator in (2) is usually implicit. Let's take for example an Advection Reaction Diffusion PDE in conservative form, i.e. a system like (1) with $L(s) = -\nabla \cdot (\mu \nabla s + \underline{b} \cdot s) + \sigma s$ and u as the forcing term. Even in such a remarkable case, $s = G(u)$ is implicit.

The theoretical foundation for this whole branch of Scientific Machine Learning is the Universal Approximation Theorem for Operators [2], a more general version of the more famous Universal Approximation Theorem [3, 4], which states that neural networks can be used to approximate any continuous function to arbitrary accuracy if no constant is placed on the width and depth of the hidden layers. Our theorem states instead that a NN[2] with a single hidden layer can approximate accurately any nonlinear continuous functional or operator (even nonlinear).

Instead of quoting the whole theorem, that can be found in [1, 2], I will give an explanation, loosing some rigour in favour of clarity, as I did in my talk.

The base hypothesis is that we have a `continuous non-polynomial function` σ, such as any common activation function besides ReLU, and a `nonlinear continuous operator G`. This latter requirement poses very few restrictions, as it's very easy for an operator to be continuous, but at the same time it's really hard to verify whether it holds or not. Indeed this implicit operators are almost always nonlinear, meaning that we cannot rely on the characterization *continuous* $\iff$ *bounded*, or even unknown, as it is explained later in section 3.3. Anyways, such theorems provide a sufficient condition for convergence, not a necessary condition. Thus, in practice the following techniques can still be used and the theorem serves as an intuition and a justification to the DeepONet architecture.

In the aforementioned hypotheses the theorem states that, with arbitrary accuracy ε, the following inequality holds:

$$\left| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma(\omega_k \cdot y + \zeta_k) \right| < \varepsilon \tag{3}$$

In this inequality it is easy to spot the structure of a neural network[3]. Actually, there is a combination of p NNs, that we will call *branches*, and a self standing one called *trunk*. The theorem guarantees that the output of this combination of neural networks, as described in the second addend of the left-hand side, converges to G(u)(y) with arbitrary accuracy.

3 The DeepONet

3.1 The architecture

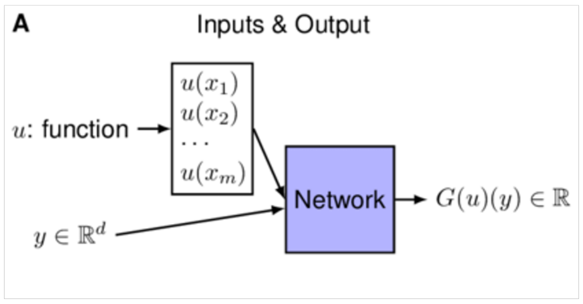


Figure 1: Input-Output scheme as reported in [1]

Definition (2) reads that the operator is an object that maps functions into other functions, but little it says on how to encode these functions as input and outputs of a neural network, which can only handle numbers. A process of discretization of all the quantities involved is necessary.

To encode the input function u, the authors of [1] engineered a solution involving the use of `sensors`. We can imagine them as a net of buoys in the sea, each one of them fixed in place, but free to move up and down with the passing of waves, while registering the vertical displacement. Given the coordinates of the sensors on a grid and their vertical displacement it is easy to

[2]Abbreviation for Neural Network, used throughout the whole document

[3]Take a general neuron connected to n other neurons in the previous layer. Let σ be an activation function, b a bias, x_i *and* ω_i the inputs and weights respectively. Then the output of this neuron is determined by $\sigma\left(b + \sum_{i=1}^{n} x_i \omega_i\right)$.

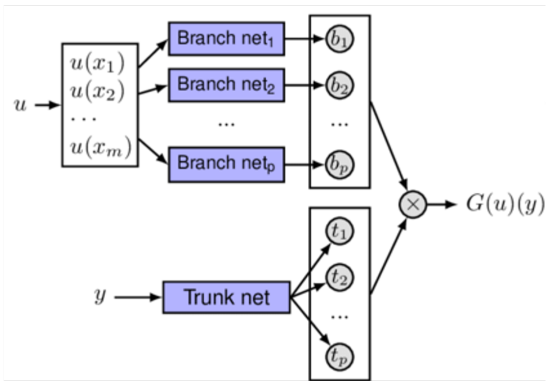
interpolate the values and reconstruct the shape of the wave with good approximation. Neural networks don't explicitly work like this, nonetheless this is a good intuition on how informative such a measure is.

Formally, we choose fixed points in the 1D/2D/3D domain and we evaluate the functions on them, then we feed this numbers to the branches of the NN. It's very important that the sensors remain always fixed, otherwise the network will completely lose understanding of the functions.

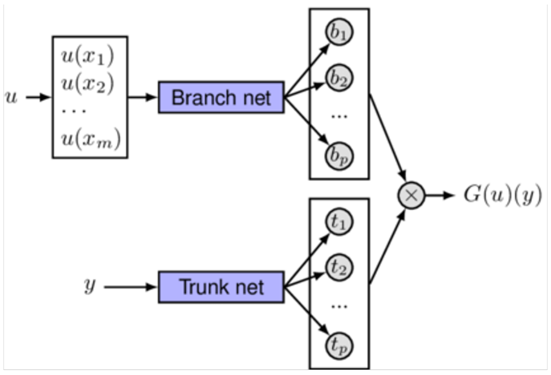
The encoding of the output function $G(u) = s$ is much simpler, since we can rely on point evaluations.

As beautifully shown in Fig. 2, the trunk of the network is fed with a fixed number of random evaluation points y, for example a random subset of nodes in a mesh.

Finally, the output of the p branches and of the trunk net are then combined in the way described by formula (3) to get the final point-wise estimation of $G(u)(y) = s(y)$.



(a) STACKED version



(b) UNSTACKED version

Figure 2: The network comes in two versions: until now we described the stacked version, but in applications it is more practical to combine all the branches into a single one with a bigger output space. Credit: [1]

3.2 Training

The training of DeepONets is a supervised task, meaning that we run some training samples through the network of which we know the results a priori, we collect the output and compute a loss against the real value. Then we backpropagate the error adjusting the parameters. [4]

In this section we look at the structure of the batches, in subsection 3.3 we look instead at the sampling procedure.

$$[\mathbf{u}, \mathbf{y}, G\mathbf{u}(\mathbf{y})] = \left[\begin{bmatrix} [u^{(1)}(x_1), u^{(1)}(x_2), ..., u^{(1)}(x_S)] \\ \cdots \\ [u^{(1)}(x_1), u^{(1)}(x_2), ..., u^{(1)}(x_S)] \\ \hline [u^{(2)}(x_1), u^{(2)}(x_2), ..., u^{(2)}(x_S)] \\ \cdots \\ [u^{(2)}(x_1), u^{(2)}(x_2), ..., u^{(2)}(x_S)] \\ \hline . \\ . \\ . \\ \hline [u^{(N)}(x_1), u^{(N)}(x_2), ..., u^{(N)}(x_S)] \\ \cdots \\ [u^{(N)}(x_1), u^{(N)}(x_2), ..., u^{(N)}(x_S)] \end{bmatrix}, \begin{bmatrix} y_1^{(1)} \\ \cdots \\ y_P^{(1)} \\ \hline y_1^{(2)} \\ \cdots \\ y_P^{(2)} \\ \hline . \\ . \\ . \\ \hline y_1^{(N)} \\ \cdots \\ y_P^{(N)} \end{bmatrix}, \begin{bmatrix} Gu^{(1)}(y_1^{(1)}) \\ \cdots \\ Gu^{(1)}(y_P^{(1)}) \\ \hline Gu^{(2)}(y_1^{(2)}) \\ \cdots \\ Gu^{(2)}(y_P^{(2)}) \\ \hline . \\ . \\ . \\ \hline Gu^{(N)}(y_1^{(N)}) \\ \cdots \\ Gu^{(N)}(y_P^{(N)}) \end{bmatrix} \right] \tag{4}$$

Let's suppose we want to construct batches with N vectors $\mathbf{u}^{(i)}$ of evaluations of the corresponding function $u^{(i)}$ over S sensors and, after the application of the operator, evaluate on P evaluation points. The batch will look like in (4). It's worth remembering that sensors are fixed, but for each $\mathbf{u}^{(i)}$ it's possible to have different evaluation points $y_1^{(i)}$ ... $y_P^{(i)}$. Moreover it can be noticed that for each $\mathbf{u}^{(i)}$ there need to be P copies of $[u^{(i)}(x_1), u^{(i)}(x_2), ..., u^{(i)}(x_S)]$, explained by the fact that a complete run in the network needs a triplet $[\mathbf{u}, \mathbf{y}, G\mathbf{u}(\mathbf{y})]$[5]and that there are P times more $y_P^{(i)}$ than $\mathbf{u}^{(i)}$.

[4]For details on automatic differentiation and backpropagation refer to the materials of the talk held by colleague Lena Stelter on the 6th November 2023.

[5]Slight change in notation: previously y was not bold, i.e. in vector notation, to ease readability, but it must be considered that in 2D and 3D the evaluation points have multiple dimensions, too.

3.3 Two approaches

DeepONets can be used in two fundamentally different approaches, that drastically shape the way training data are collected.

$$\begin{bmatrix} \mathbf{u} \\ \mathbf{y} \\ G\mathbf{u}(\mathbf{y}) \end{bmatrix}$$

The first is the modelling approach: the physical phenomena has been described by a system of equations together with boundary and initial conditions. This is now a mathematical problem. We assume that no analytical solution exists or that it cannot be easily found, otherwise there would be no necessity of any simulation. Now the problem gets discretized and solved via some numerical algorithm, like a Finite Difference or a Finite Element Method. In this case we have labeled data: input $\mathbf{u}, \mathbf{y}$ and output $G\mathbf{u}(\mathbf{y})$.

The second situation is the experimental setup. The model underlying the phenomena is not known a priori, but there is a physical setup of the experiment, with fixed sensors. Sensors will directly register data, functioning as entries for vector $\mathbf{u}$, and for different inputs $\mathbf{y}$, for example voltages, we register various outputs $G\mathbf{u}(\mathbf{y})$ at the end receiver of the experiment. The generality of the operator allows this, but it's not guaranteed at all that the real underlying operator is continuous, therefore the Universal Theorem of Approximation for Operators doesn't hold and there is no guarantee of convergence. I focused all my study on the first approach and the papers I visioned, that were published after [1], all follow the first approach.

3.3.1 How training works

- Randomly select N functions $u(x)$. This is achievable in multiple ways. The authors of [1] suggested and tried Gaussian Random Fields, Chebychev Polynomials, Legendre Polinomials and Poly-fractonomials. My experiments were all conducted on GRF.
- Select S sensors
- Evaluate every $u(x)$ at the sensors, obtaining the vector $\mathbf{u}$

For every $\mathbf{u}$:

- Use a numerical solver on the original $u(x)$ (or an interpolation if not available) and get a solution on the whole computational grid
- Randomly sample P nodes from the computational grid. Extract and store the solution only on those nodes

3.3.2 How inference phase works

Given a function $\hat{u}(x)$ we want to evaluate $G\mathbf{u}$ on the whole computational grid. Input data must be prepared in the right shape:

- Evaluate $\hat{u}(x)$ at the sensors (the same of training)
- Pack this vector together with the desired sample points of the domain (usually the whole computational grid)

  Note: Particularly annoying for 2D grids that must be unrolled in a vector in this phase.
- Feed the input to the DeepONet, the solution will be computed with a single forward pass in the network

4 Code

4.1 Premise

The code I presented to the class is far from perfect, but it would have been hard to do much better with the goal I had in mind and the time at disposal. It required four weeks of work and plenty of debugging. It would have been easier to run a few example offered by the author himself, Lu Lu, from his official library on GitHub, namely deepxde [5] [6], or his secondary repository deeponet [7], which relies on the former and acts as interface to solve PDEs. The code he provides runs smoothly and allows to play with the DeepONet, but it doesn't help to show what happens in the background. My goal, instead, was to guide the audience step by step in the procedure, so that they could recognise the key concepts that I had mentioned beforehand with the slides. My idea, was therefore to rely on both the latest release of the deepxde library, also downloadable with pip, and an older version, namely v0.11.2[6]. The latest release is definitely more feature complete, but hides key components in complex inherited classes, while the older version is less advanced but allows direct control on every step. This choice entails the need of importing the libraries locally and it's not elegant. Anyways, I'm happy with the result, since it accomplished the goal of guiding the audience in a personalised experience.

The integral code can be found on my Github, accessible from here [8] or from the link on the slides uploaded on the cloud platform HeiBox.

[6]I must give credit for the intuition of using v0.11.2 to Jan Sprengel and Robin Janssen, fellow students who attended this seminar in SoSe 2023. However, my approach differs from theirs, as they also rely on the deeponet repository [7], while I do everything from scratch for didactic purposes.

4.2 Heat equation

4.2.1 Learning the operator from the forcing term u(x) to the solution s(x,t)

$$\begin{cases} \frac{\partial}{\partial t}s(x,t) - \frac{\partial^2}{\partial x^2}s(x,t) = u(x) \quad \forall x,t \in [0,1]\times[0,T] \\ s(0,t) = 1 \quad \forall t \in [0,T] \\ s(L,t) = 0 \quad \forall t \in [0,T] \\ s(x,0) = s_0(x) \quad \forall x \in [0,L] \end{cases} \tag{5}$$

I chose the heat equation as an example of PDE that would be easy to show, but also not so easy that the implementation of a DeepONet would feel unnecessary. We want to learn the operator that maps from the forcing term to the solution of the system.

The first thing I show is the generation of $u(x)$ with GRFs, then I propose a manual solver for the heat equation with the Crank Nicholson scheme, since it's a method of the second order both in time and space, thus providing an accurate reference solution. At this point I manually demonstrate the generation of the batches of training samples, as shown in 4. Data gets then packaged in a special OpDataSet object and passed to the network. Without indulging on hyper-parameters tuning, I train the DeepONet.

4.2.2 Learning the operator from the forcing term u(x,t) to the solution s(x,t)

$$\begin{cases} \frac{\partial}{\partial t}s(x,t) - \frac{\partial^2}{\partial x^2}s(x,t) = u(x,t) \quad \forall x,t \in [0,1]\times[0,T] \\ \dots \end{cases} \tag{6}$$

In this second notebook the input function to the operator to be learned is u(x,t), o it is dependent both on space and time. The goal here is to transfer our knowledge to a more sophisticated scenario and investigate how solid the technique is.

Now the sensors must be arranged on a mesh grid. This increases exponentially the computational cost for having the same density of sensors on the domain. Gaussian Random Fields instead are easily adaptable from line to surfaces and I showcase them with the beautiful graphics offered by the python module *plotly*. The numerical solver is only slightly different.

This examples was the first of many reflections that sparkled in my mind and it was used to introduce the audience to the next phase, where many question and doubts were openly discussed.

5 Reflections

5.1 Open questions, limitations and reflections

Number of sensor for higher dimensional functions

The first reflection was introduced to the class with the second coding example (sec. 4.2.2). What if the input function has many dimensions? The number of sensors required to achieve the same accuracy increases exponentially with the dimension of the input function space. Let's imagine a 1D domain of length L, where we put S equispaced sensors. To maintain the same density on a 2D domain of size $L*L$, we will need S^2 sensors, on a 3D domain we'll need S^3, so on and so forth. This is what happened in the example when shifting from a forcing term only dependant on space to one dependant both on space and time.

A possible solution is to abandon the sensors' approach in high dimensions. Lu Lu et al. [1] suggest an alternative approach, consisting in projecting u to a set of basis functions and then using the coefficients as a representation of u, but they don't explore it any further. Moreover the authors expect that in the future the input trajectories might be expressed by NNs. However no theory has been published on the matter yet.

Complex geometries

Every example provided by the authors works on rectangular domains. I didn't personally try DeepONets on complex meshes myself, because it would have been an enormous effort to set up a FEM and adapt the deepxde code for it. However I believe that a clever positioning of the sensors could do the job and let the architecture learn the geometry together with the network. I am worried though, that the network would struggle to follow the boundaries of the meshes, causing imperfections. This is totally acceptable if the subject of interest is in the center of the mesh, but problematic if the nature of the study involves the interaction at the border.

Boundary conditions

In many scientific or engineering applications the boundary conditions play a major role in the simulated phenomena and sometimes they are even the main subject of the study. How can we create a DeepONet that takes BCs as input[7]? For the sake of simplicity, let's only take in consideration Dirichlet BCs. They can be described as coefficient (or functions) multiplied by a

[7]This is something quoted by the authors as a possible usage for their network

Dirac δ : they spike up at the border and are zero everywhere else. This means that placing sensors in the middle of the mesh would be a waste and, at the same time, a lack of sensors on the border would fail to catch the spike nature of the function.

A possible remedy could be to distribute many sensors in multiple thin layers against the borders of the mesh, as always causing a noticeable increase in computational cost. But I fear this would never be as good as passing the BCs as parameters or as applying a correction on the rhs of a FD or by introducing a lifting operator on a FEM.

5.2 Limitations addressed in following works

Single input limitation and Multiple Input DeepONet

Until now the input function was one, for instance in sec. 4.2.2 it was the forcing term, but the initial datum was fixed. What if we wanted to learn the operator from two functions $(u1, u2)$ to s? A solution is provided by *P. Jin et al.* in [9], where they prove a new universal approximation theorem for continuous multiple-input operators and introduce MIONet, an evolution of DeepONet, whose main difference lies in the merging of the outputs of branches and trunk, now obtained via Hadamard product followed by summation.

Flexibility issues

What if after training offline a DeepONet for mapping some u to the solution, we realise we have to change something else. Referring to the coding example: what if after having completed the training for the operator mapping from the forcing term to the solution, the need arises to change the boundary conditions? Must the training be repeated from scratch? In other words, how flexible are these networks? If this fails, then they lose all their advantage against classical numerical solvers.

A partial answer is again provided by Multiple Input DeepONets. This way we can learn an operator that maps from both the values of the BCs, the initial datum and the forcing term to the solution space. However the solution is only partial, since, in order to assure coverage in every situation, training is also needed for various initial data and values of the BCs, running again in an exponential increase of the computational cost.

As a verdict, Multiple Input DeepONets allow to learn much more elaborate and realistic operators, but don't solve the flexibility issue of DeepONets, that limits their use to particular situations, where all but a function is fixed.

The need of many paired input-output observations and the rise of Physics-Informed DeepONets

To leverage all the advantage of PINNs, combining the knowledge of the PDEs into the network loss, thus giving a boost in convergence rate and accuracy. This is what was presented by S. Wang et al. in [10]. According to the authors, a trained physics informed DeepOnet model can predict the solution of $\mathscr{O}(10^3)$ time-dependent PDEs in a fraction of a second – up to three orders of magnitude faster compared to a conventional PDE solver.

Fixed sensors and Variable Input DeepONet

This is in my opinion the least impactful innovation. Variable Input DeepONets, introduced by M. Prasthofer et al. in [11], allow to move or change the amount of sensors from one input or test sample to the next. Besides allowing some hypothetical algorithms for better sensor placement, the only straightforward advantage concerns the experimental approach, where the necessity of moving physical sensors might occur for external reasons. It is anyway a very welcomed generalisation.

Deep Transfer Operator

This might reveal itself as the game changer in the next few years. With CNNs and LLMs it's nowadays common practice to fine tune or do transfer learning on very heavy but powerful NNs, obtaining results that would not be achievable by any handmade network trainable in commonly affordable GPU hours. I hope that the Deep Transfer Operator proposed in [12] will unleash the full potential of this technology in the scientific computing community.

6 Conclusions

Deep operator learning is a very interesting approach, that, once mature and further developed, could reveal itself as a decisive tool in the hands of the scientific community. The power of these NNs lies in the quickness of the inference phase compared to the full runtime of a numerical method. In this regard they excel by a substantial margin.

However they remain a tool for experts, unlike many other fields of application of neural networks.

They are a situational tool: they are extremely efficient when the variable functions lie in the same role in the PDE's scheme and the simulations are repetitive enough to justify the training cost. But as discussed in section 5, they lack flexibility in multifaceted scenarios.

Overall, I'm really happy to have chosen this topic. I hope I have conveyed the same passion in this report as in the talk itself, which I closed with a very fascinating example, namely the application of a special DeepONet to the simulation of high mach number flows around bluff objects [13].

My only regret is that time and page constraints in the talk and in this report respectively didn't allow me to dive deeper in the more recent evolutions of the architecture, which I found very interesting reads.

References

[1] Lu Lu et al. "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators". In: *Nature Machine Intelligence* 3.3 (Mar. 2021), pp. 218–229. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00302-5. URL: http://dx.doi.org/10.1038/s42256-021-00302-5.

[2] Tianping Chen and Hong Chen. "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems". In: *IEEE Transactions on Neural Networks* 6.4 (1995), pp. 911–917. DOI: 10.1109/72.392253.

[3] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[5] Lu Lu et al. *DeepXDE: A deep learning library for solving differential equations*. URL: https://github.com/lululxvi/deepxde.

[6] Lu Lu et al. "DeepXDE: A deep learning library for solving differential equations". In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: 10.1137/19M1274067.

[7] Lu. *deeponet: Learning nonlinear operators via DeepONet*. URL: https://github.com/lululxvi/deeponet.

[8] Matteo Malvestiti. *DeepOperatorLearning*. URL: https://github.com/Matteo-Malve/DeepOperatorLearning.

[9] Pengzhan Jin, Shuai Meng, and Lu Lu. "MIONet: Learning multiple-input operators via tensor product". In: (2022). arXiv: 2202.06137 [cs.LG].

[10] Sifan Wang, Hanwen Wang, and Paris Perdikaris. "Learning the solution operator of parametric partial differential equations with physics-informed DeepOnets". In: (2021). arXiv: 2103.10974 [cs.LG].

[11] Michael Prasthofer, Tim De Ryck, and Siddhartha Mishra. *Variable-Input Deep Operator Networks*. 2022. arXiv: 2205.11404 [cs.LG].

[12] Somdatta Goswami et al. "Deep transfer operator learning for partial differential equations under conditional shift". In: *Nature Machine Intelligence* 4.12 (2022), pp. 1155–1164. DOI: 10.1038/s42256-022-00569-2. URL: https://doi.org/10.1038/s42256-022-00569-2.

[13] Zhiping Mao et al. "DeepM&Mnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators". In: *Journal of Computational Physics* 447 (Dec. 2021), p. 110698. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2021.110698. URL: http://dx.doi.org/10.1016/j.jcp.2021.110698.