AMS 595 FINAL PROJECT
A DEEP LEARNING BASED PDE SOLVER

Wenhan Gao, Jinglin Yang, Zhihao He
Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA
contact: wenhan.gao@stonybrook.edu

ABSTRACT

The deep-learning-based least squares method has shown successful results in solving high-dimensional non-linear partial differential equations (PDEs). Therefore, in this project, we use a deep learning based method to solve a hyperbolic equation.

Keywords Deep Learning · PDEs

AMS subject classifications 65M75 · 65N99

1 Introduction

There are various traditional numerical methods for solving partial differential equations, especially for the low-dimensional problems. However, when it comes to solve high-dimensional problems, the curse of dimensionality becomes a major computational challenge for many traditional methods; e.g., in the finite difference method [1], the computational complexity is exponential with respect to the number of dimensions. With that being said, traditional methods oftentimes are computationally intractable in solving high-dimensional problems.

Recent developments in computing capability and large storage media has made Deep Learning the fastest growing field in Artificial Intelligence. Deep Learning has had extraordinary successes in many fields including computer vision, natural language processing, etc.. Many recent studies [2, 3, 4, 5, 6, 7, 5, 8, 9] indicate Deep Learning surpasses other numerical methods especially in solving high- dimensional PDEs and PDEs with complex geometries or irregular domains.

Moreover, there are theoretical foundations for Deep Learning based PDE solvers. In approximation theory, deep neural networks (DNNs) have been proved to be an effective tool for function approximation. In [10, 11, 12, 13, 14, 15], it has been shown that ReLU-type DNNs can provide an accurate approximation to continuous functions and smooth functions. Deep Learning based PDE solvers have also been applied to solve problems in many areas of practical engineering and life sciences [16, 17, 18].

Therefore, in this project, we try to build a deep learning based PDE solver to solve a hyperbolic equation.

1.1 Neural Networks

A fully connected feed-forward neural network can be expressed as a composition of several activation functions; i.e.,

$$\phi(\boldsymbol{x};\boldsymbol{\theta}) = h_L \circ h_{L-1} \circ ... \circ h_1 \circ h_0(\boldsymbol{x}), \tag{1.1}$$

$$h_l(\boldsymbol{x}^l) = \sigma(\boldsymbol{W}^l\boldsymbol{x}^l + b^l), \text{ for } l = 0, 1, 2, ..., L-1, \qquad h_L(\boldsymbol{x}^L) = \boldsymbol{W}^L(\boldsymbol{x}^L + b^L),$$

where $\boldsymbol{x}$ is the input, L is the depth of the network, σ is a nonlinear activation function such as Tanh, Sigmoid, and ReLU, $\boldsymbol{\theta}$ contains all the $\boldsymbol{W}$-s and b-s is the set of parameters in the network.

With both linearity and non-linearity, we can find a set of parameters($\boldsymbol{W}$-s and b-s) to approximate most functions in the Sobolev Space. Note that we can describe many physical phenomena as a function; for example, binary classification can be described as a function that maps objects to either 1 or 0. To make the network approximation

accurate, we have an empirical loss functional $\mathcal{L}(\boldsymbol{\theta})$ that measures the difference between network predictions and true values, and naturally, the goal is to find the "best" set of parameters in the network that minimizes this loss functional: $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$. One can learn more about neural networks and deep learning in [19].

1.2 Solution Network Training

Consider the following boundary value problem for simplicity to introduce the deep-learning-based least squares method: find the unknown solution $u(\boldsymbol{x})$ such that

$$\begin{cases} \mathcal{D}u(\boldsymbol{x}) = f(\boldsymbol{x}), & \text{in } \Omega, \\ \mathcal{B}u(\boldsymbol{x}) = g(\boldsymbol{x}), & \text{on } \partial\Omega, \end{cases} \tag{1.2}$$

where $\partial\Omega$ is the boundary of the domain, $\mathcal{D}$ and $\mathcal{B}$ are differential operators in Ω and on $\partial\Omega$, respectively. The goal is to train a neural network, denoted by $\phi(\boldsymbol{x};\boldsymbol{\theta})$ to approximate the ground truth solution $u(\boldsymbol{x})$ of the PDE (1.2). In the least squares method (LSM), the PDE is solved by finding the optimal set of parameters $\boldsymbol{\theta}^*$ that minimizes the loss functional:

$$\begin{aligned} \boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) :&= \arg\min_{\boldsymbol{\theta}} \|\mathcal{D}\phi(\boldsymbol{x};\boldsymbol{\theta}) - f(\boldsymbol{x})\|_2^2 + \lambda\|\mathcal{B}\phi(\boldsymbol{x};\boldsymbol{\theta}) - g(\boldsymbol{x})\|_2^2 \\ &= \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x}\in\Omega}\left[|\mathcal{D}\phi(\boldsymbol{x};\boldsymbol{\theta}) - f(\boldsymbol{x})|^2\right] + \lambda\mathbb{E}_{\boldsymbol{x}\in\partial\Omega}\left[|\mathcal{B}\phi(\boldsymbol{x};\boldsymbol{\theta}) - g(\boldsymbol{x})|^2\right] \\ &\approx \arg\min_{\boldsymbol{\theta}} \frac{1}{N_1}\sum_{i=1}^{N_1} \left|\mathcal{D}\phi\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right) - f\left(\boldsymbol{x}_i\right)\right|^2 + \frac{\lambda}{N_2}\sum_{j=1}^{N_2} \left|\mathcal{B}\phi\left(\boldsymbol{x}_j;\boldsymbol{\theta}\right) - g\left(\boldsymbol{x}_j\right)\right|^2, \end{aligned} \tag{1.3}$$

where λ is a positive hyper-parameter that weights the boundary loss. The last step is a Monte-Carlo approximation with $\boldsymbol{x}_i \in \Omega$ and $\boldsymbol{x}_j \in \partial\Omega$ being N_1 and N_2 allocation points sampled from respective probability densities that $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ follow. In this project, the temporal coordinate is regarded as another spatial coordinate, so the time cannot go to infinity.

To understand this loss functional, it penalizes the network by the extend to which the network approximation violates the PDE and possible boundary/initial conditions; it minimizes the difference between the L.H.S. and R.H.S. of the equation in Equation 1.2.

The network is trained to solve the minimization problem defined in (1.3) via Adam [20] (a variant of stochastic gradient descent), where the empirical loss is defined by

$$J(\boldsymbol{\theta}) = \frac{1}{N_1}\sum_{i=1}^{N_1} \left|\mathcal{D}\phi\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right) - f\left(\boldsymbol{x}_i\right)\right|^2 + \frac{\lambda}{N_2}\sum_{j=1}^{N_2} \left|\mathcal{B}\phi\left(\boldsymbol{x}_j;\boldsymbol{\theta}\right) - g\left(\boldsymbol{x}_j\right)\right|^2, \tag{1.4}$$

and $\boldsymbol{\theta}$ is updated by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla J(\boldsymbol{\theta}), \tag{1.5}$$

where α is the learning rate.

The network is trained to find the best parameters in the network such that when the network takes input points, it will output network predictions that are close to the true solution of the PDE:

Algorithm 1.1: Network Training(main.py)

Result: parameters $\boldsymbol{\theta}^*$
Require: PDE (1.2)
Set n = total iterations/epochs, N_1 and N_2 for size of sample points in Ω and on $\partial\Omega$ respectively ;
Initialize $\phi(\boldsymbol{x};\boldsymbol{\theta})$, k = 0; **while** $k < n$ **do**
  Generate uniformly distributed training points, $\left\{\boldsymbol{x}_i^1\right\}_{i=1}^{N_1} \subset \Omega$ and $\left\{\boldsymbol{x}_j^2\right\}_{j=1}^{N_2} \subset \partial\Omega$;
  Loss: $J(\boldsymbol{\theta}) = \frac{1}{N_1}\sum_{i=1}^{N_1}\left|\mathcal{D}\phi\left(\boldsymbol{x}_i^1;\boldsymbol{\theta}\right) - f\left(\boldsymbol{x}_i^1\right)\right|^2 + \frac{\lambda}{N_2}\sum_{j=1}^{N_2}\left|\mathcal{B}\phi\left(\boldsymbol{x}_j^2;\boldsymbol{\theta}\right) - g\left(\boldsymbol{x}_j^2\right)\right|^2$;
  Update: $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha\nabla J(\boldsymbol{\theta})$;
  $k++$;
end
return $\boldsymbol{\theta}^* := \boldsymbol{\theta}$;

1.3 Specific PDE Problem to Solve

Consider the following hyperbolic equation to solve:

$$\begin{aligned} \frac{\partial^2 u(x,t)}{\partial t^2} - \Delta_x u(x,t) &= f(x,t), \quad && \text{in } \Omega := \omega \times \mathbb{T}, \\ u(x,t) &= g_0(x,t), \quad && \text{on } \partial\Omega = \partial\omega \times \mathbb{T}, \\ u(x,0) = h_0(x) \quad \frac{\partial u(x,0)}{\partial t} &= h_1(x), \quad && \text{in } \omega, \end{aligned} \tag{1.6}$$

where x is 2-dimensional, $\omega := \{x : |x| < 1\}$, $\mathbb{T} = (0,1)$. $g_0(x,t) = 0$, $h_0(x) = 0$, $h_1(x) = 0$, and $f(x,t)$ is given appropriately so that the exact solution is $u(x,t) = \left(\exp\left(t^2\right) - 1\right) \sin\left(\frac{\pi}{2}(1-|x|)^{2.5}\right)$. Δ_x denotes the Laplace operator taken in the spatial variable x only. Note that this PDE is from [21].

1.4 Accuracy Assessment

To measure the accuracy of the model, we have 61103 testing points$\{\boldsymbol{x}_i^t\}_{i=1}^{61103} \subset \Omega$, and we judge the performance of the code by the overall relative ℓ_2 error at these points. The overall relative ℓ_2 error is defined as follows

$$error_{\ell^2}(\boldsymbol{\theta}) := \frac{\left(\sum_{i=1}^{61103} \left|\phi\left(\boldsymbol{x}_i^t; \boldsymbol{\theta}\right) - u\left(\boldsymbol{x}_i^t\right)\right|^2\right)^{\frac{1}{2}}}{\left(\sum_{i=1}^{6113} \left|u\left(\boldsymbol{x}_i^t\right)\right|^2\right)^{\frac{1}{2}}},$$

These testing points are grid points in the domain Ω given by:

$$\{(t,x,y) : t = 0.02i,\ \ x = 0.05j,\ \ y = 0.05k,\ \ i,\ \ j,\ \ k \in \mathbb{Z}\ \ s.t.\ \ (t,x,y) \in \Omega\}$$

2 Implementation

2.1 About the Program

There are 5 python scripts used in the program to solve the PDE(1.2).

- **main.py**: the main script in which the network trainning takes place. It does not take any input, but it imports other scripts including network_setting.py, problem_setting.py, and utilities.py. It will produce a .pt file that contains the set of parameter values of the trained network and also 2 .mat files that contains the sequence of training time and l2error in each epoch.
- network_setting.py contains the setup(architecture) of the neural network.
- problem_setting.py contains in formation related to the PDE(1.2) such as the boundary operator, differential operator.
- utilities.py contains some useful tools such as generating training points, evaluating ℓ_2 error.
- **get_result.py** is used if one wants to know the solution for some specific points. To use this script, we need the trained neural network, i.e., the .pt file produced by main.py, and imports network_setting.py file.

There are also 2 additional python scripts used to visualize how the PDE system evolves over time.

- **get_data_for_visualization.py** will produce 3 .mat files. One contains a set of grid points, one contains corresponding network solution, and one contains corresponding true solution. This script needs network_setting.py, problem_setting.py, and the .pt file generated by main.
- **visualization.py** needs above 3 .mat files, and it will produce two GIFs showing the solution given by the network and the true solution.

The program requires a Python Interpreter, 3.9 is used when developing the code, however, 3.7 and above should suffice. Moreover, the following libraries is used in the program: pandas, scipy.io, matplotlib, imageio, os , glob, numpy, torch(PyTorch), timeit.

One thing to mention here is that for first-order derivatives in the problem, we can use PyTorch Autograd to evaluate. However, for second-order derivatives, Autograd is not outputing correct partial derivatives, so we used numerical differentiation(Appendix A) with step size $h = 0.0001$.

2.2 Outline of the Code

For the network training part, the outline is given in Algorithm1.1. For the animation part, the code imports the data(.mat files) we produced and reshape them to suitable shape. The code will build the 3D coordinates and plot all XYZ point corresponding to each time and save as .png files. All of the plot will save in the folder temporarily. The code will sort all the plots in the order of time and generate GIFs. To save the space, the code will delete all the .png files at the end.

2.3 Shortcoming of the Code

In low dimensions, traditional numerical methods definitely will outperform network-based solvers. However, due to the constraint of computational resources, we can only present the case in low dimensions.

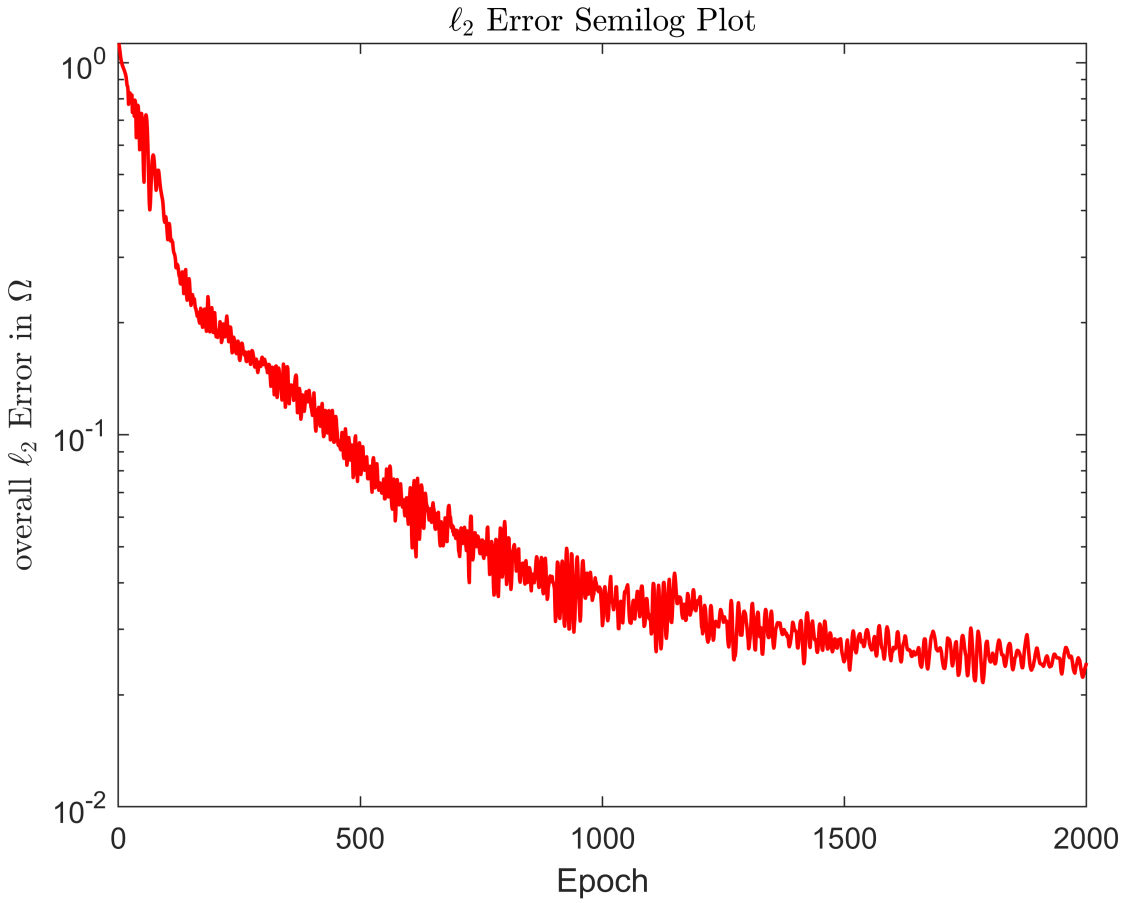
2.4 Result

The neural network is trained for 2000 epochs, which took 233.54 seconds to train on NVIDIA GeForce GTX 1050 Ti Graphics Cards.

2.4.1 ℓ_2 Error Decay

At the end, it reached an ℓ_2 error of 0.0244(2.44%). One can certainly further tune hyperparameters and train more epochs to reach a lower error. Figure 2.1 presents the ℓ_2 error decay.

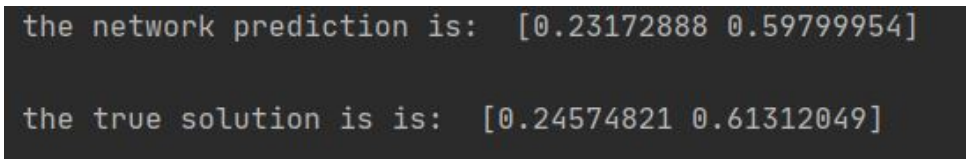
Figure 2.1: ℓ_2 Error Decay as Network Trains



2.4.2 Sample Predictions

Here, two points (0.8, 0.4, 0.3), (0.9, 0.3, 0.2) is sent to the network to get network predictions. Figure 2.2 presents the comparison between true solutions and network predictions.

Figure 2.2: Sample Predictions



2.4.3 Sample Interface in Network Training

Figure 2.3: Sample Feedback Interface

2.4.4 Animation of the PDE Solutions

The following Links will show the solutions of the PDE over time:**The True Solution**, **The Network Prediction**

2.4.5 Discussion of the Result

As mentioned in the shortcomings, the result is not perfect. Most traditional solvers can outperform network-based solvers in low dimensions. Although the network in this work reaches an ℓ_2 error of 0.0244, it can definitely be lowered if one trains the network for more epochs with suitable learning rate. Moreover, there can be better network architectures, such as CovNet, and better mathematical models that use the weak formulation of the PDE [5]. Overall, the network can give an accurate prediction to the PDE solution although it is still not perfect.

3 Description of Collaboration

Here is the break down of work distribution:

- Developing Mathematical Model: Wenhan Gao, Jinglin Yang
- Network Framework: Wenhan Gao, Jinglin Yang
- Hyperparameter Tuning: Wenhan Gao, Zhihao He
- Network Training: Jinglin Yang
- Experiment data handling: Wenhan Gao, Jinglin Yang
- Animated Visualization: Zhihao He
- Debuging and Testing: Wenhan Gao, Jinglin Yang, Zhihao He
- Organizing Files/Version Control: Wenhan Gao, Jinglin Yang, Zhihao He
- Making Slides and Writing Report: Wenhan Gao

References

[1] Andrew R. and D. F. Griffiths. *The finite difference method in partial differential equations*. Wiley, Chichester, [England], 1980.

[2] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018. ISSN 0027-8424. doi: 10.1073/pnas.1718942115. URL https://www.pnas.org/content/115/34/8505.

[3] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[4] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

[5] Weinan E and Bing Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1), 2018. ISSN 2194-6701.

[6] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences - PNAS*, 115(34):8505–8510, 2018. ISSN 0027-8424.

[7] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018. ISSN 0021-9991.

[8] Jianguo Huang, Haoqin Wang, and Haizhao Yang. Int-deep: A deep learning initialized iterative method for nonlinear problems. *Journal of Computational Physics*, 419:109675, 2020. ISSN 0021-9991.

[9] Arnulf Jentzen, Diyora Salimova, and Timo Welti. A proof that deep artificial neural networks overcome the curse of dimensionality in the numerical approximation of kolmogorov partial differential equations with constant diffusion and nonlinear drift coefficients, 2019.

[10] Dmitry Yarotsky. Optimal approximation of continuous functions by very deep ReLU networks. In Sébastien Bubeck, Vianney Perchet, and Philippe Rigollet, editors, *Proceedings of the 31st Conference On Learning Theory*, volume 75 of *Proceedings of Machine Learning Research*, pages 639–649. PMLR, 06–09 Jul 2018. URL http://proceedings.mlr.press/v75/yarotsky18a.html.

[11] Dmitry Yarotsky and Anton Zhevnerchuk. The phase diagram of approximation rates for deep neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 13005–13015. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/979a3f14bae523dc5101c52120c535e9-Paper.pdf.

[12] Zuowei Shen, Haizhao Yang, and Shijun Zhang. Deep network approximation characterized by number of neurons. *Communications in Computational Physics*, 28(5):1768–1811, 2020. ISSN 1991-7120. doi: 10.4208/cicp.OA-2020-0149.

[13] Jianfeng Lu, Zuowei Shen, Haizhao Yang, and Shijun Zhang. Deep network approximation for smooth functions. *SIAM Journal on Mathematical Analysis*, 53(5):5465–5506, 2021. doi: 10.1137/20M134695X. URL https://doi.org/10.1137/20M134695X.

[14] Zuowei Shen, Haizhao Yang, and Shijun Zhang. Optimal approximation rate of ReLU networks in terms of width and depth. *Journal de Mathématiques Pures et Appliquées*, to appear. doi: 10.1016/j.matpur.2021.07.009.

[15] Sean Hon and Haizhao Yang. Simultaneous neural network approximations in sobolev spaces. *arxiv:2109.00161*, 2021.

[16] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin, 2017.

[17] Christoph Reisinger and Gabriel Wittum. Efficient hierarchical approximation of high-dimensional option pricing problems. *SIAM J. Scientific Computing*, 29:440–458, 01 2007. doi: 10.1137/060649616.

[18] Justin Sirignano, Jonathan F. MacArt, and Jonathan B. Freund. Dpm: A deep learning pde augmentation method with application to large-eddy simulation. *Journal of Computational Physics*, 423:109811, 2020. ISSN 0021-9991.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[21] Yiqi Gu, Haizhao Yang, and Chao Zhou. Selectnet: Self-paced learning for high-dimensional partial differential equations. *Journal of Computational Physics*, 441:110444, 2021. ISSN 0021-9991.

Appendix A Numerical Differentiation

Definition A.1. *A real scalar function $\phi(\boldsymbol{x})$ of d variables is a rule that assigns a number $\phi((\boldsymbol{x}) \in \mathbb{R}$ to an array of numbers $\boldsymbol{x} = (x_1, x_2, x_3, ..., x_d)$, $\boldsymbol{x} \in \mathbb{R}^d$.*

Definition A.2. *Let $\phi(\boldsymbol{x})$ be a real scalar function of d variables defined on $\mathbb{R}^d$. The **partial derivative** of $\phi(\boldsymbol{x})$ at a point $a \in \mathbb{R}^d$ with respect to $x_i,\ i = 1, 2, 3, ..., d$ is given by*

$$\frac{\partial \phi}{\partial x_i}(a) = \lim_{h_i \to 0} \frac{\phi(a + h\boldsymbol{E_i}) - \phi(a)}{h}$$

, where $h \in \mathbb{R}$, $h\boldsymbol{E_i} \in \mathbb{R}^d$ is an array of all zeros except the i-th element, which equals to h.

Definition A.3. *Let $\phi(\boldsymbol{x})$ be a real scalar function of d variables defined on $\mathbb{R}^d$, $a \in \mathbb{R}^d$, and $h \in \mathbb{R}$. The **numerical differentiation** estimate of $\frac{\partial \phi}{\partial x_i}(a)$ with respect to $x_i,\ i = 1, 2, 3, ..., d$ is given by:*

$$\frac{\partial \phi}{\partial x_i}(a) \approx \frac{\phi(a + h\boldsymbol{E_i}) - \phi(a)}{h}$$

, note that the truncation error is of order $O(h)$.