CS230

Deep Learning for Partial Differential Equations (PDEs)

Kailai Xu kailaix@stanford.edu
Bella Shi bshi@stanford.edu
Shuyi Yin syin3@stanford.edu

Abstract

Partial differential equations (PDEs) have been widely used. However, solving PDEs using traditional methods in high dimensions suffers from the curse of dimensionality. The newly emerging deep learning technqiues are promising in resolving this problem because of its success in many high dimensional problems. In this project we derived and proposed a coupled deep learning neural network for solving the Laplace problem in two and higher dimensions. Numerical results showed that the model is efficient for modest dimension problems. We also showed the current limitation of the algorithm for high dimension problems and proposed an explaination.

1 Introduction & Related Work

Solving PDEs (partial differential equations) numerically is the most computation-intensive aspect of engineering and scientific applications. Recently, deep learning emerges as a powerful technique in many applications. The success inspires us to apply such a technique to solving PDEs. One of the main feature is that it can represent complex-shaped functions effectively compared to traditional finite basis function representations, which may require large number of parameters or sophisticated basis. It can also be treated as a black-box approach for solving PDEs, which may serve as a first-to-try method. The biggest challenge is that this topic is rather new and there are few literature (for some examples on this topic, see [2]-[7] that we can refer to. Whether it can work reasonably well remains to be explored.

$$L(\theta) = (\mathcal{L}\hat{u}(x,t|\theta) - f)^2 \tag{1}$$

reads to a very accurate solution (up to 7 digits) to

$$\mathcal{L}u = f \tag{2}$$

where $\mathcal{L}$ is some certain differential operator.
Since 2017, many authors begin to apply deep learning neural network to solve PDE. Raissi, et al[3] considers,

$$u_t = N(t, x, u_x, u_{xx}, ...), \quad f := u_t - N(t, x, u_x, u_{xx}, ...) \tag{3}$$

where $u(x,t)$ is also represented by a neural network. Parameters of the neural network can be learnt by minimizing

$$\sum_{i=1}^{N}(|u(t^2, x^i) - u^i|^2 + |f(t^i, x^i)|^2) \tag{4}$$

where the term $|u(t^2, x^i) - u^i|^2$ tries to fit the data on the boundary and $|f(t^i, x^i)|^2$ tries to minimize the error on the collocation points.

I.E. Lagaris et al has already applied artificial neural network to solve PDEs. Limited by computational resources they only used a single hidden layer and model the solution $u(x,t)$ by a neural network $\hat{u}(x, t(\theta))$.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Dataset and Features

The data set is generated randomly both on the boundaries and in the innerdomain. For every update in Line 3, we randomly generate $\{(x_i, y_i)\}_{i=1}^{M}, (x_i, y_i) \in \partial\Omega$ and compute $g_i = g_D(x_i, y_i)$. The data are then fed to (2) for computing loss and derivatives.

For every update in Line 4, we randomly generate $\{(x_i, y_i)\}_{i=1}^{N}, (x_i, y_i) \in \Omega$ and compute $f_i = f(x_i, y_i)$. The data are then fed to (1) for computing loss and derivatives.

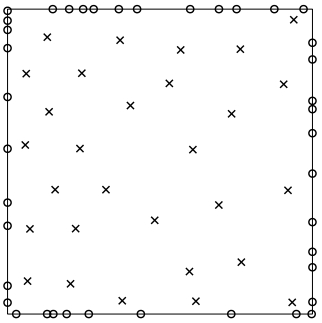


Figure 1: Data Generation for Training the Neural Network. We sample randomly from the inner domain Ω and its boundary $\partial\Omega$. Here $\times$ represents the sampling points within the domain and $\circ$ represents the sampling points on the boundary.

3 Approach

Our main focus is elliptic PDEs, which is generally presented as:

$$\begin{aligned} -\frac{\partial}{\partial x}\big(p(x,y)\frac{\partial u}{\partial x}\big) - -\frac{\partial}{\partial y}\big(q(x,y)\frac{\partial u}{\partial y}\big) + r(x,y)u = f(x,y), \qquad & \text{in } \Omega \\ u = g_D(x,y), \quad & \text{on } \Omega_D \\ p(x,y)\frac{\partial u}{\partial x}\frac{\partial y}{\partial s} - q(x,y)\frac{\partial u}{\partial y}\frac{\partial x}{\partial s} + c(x,y)u = g_N(x,y), \quad & \text{on } \Omega_N \end{aligned} \tag{5}$$

In the case $p = q \equiv -1, r \equiv 0, \partial\Omega_D = \partial\Omega$, we obtain the Poisson equation:

$$\begin{cases} \triangle u = f, & \text{in } \Omega \\ u = g_D & \text{on } \partial\Omega \end{cases} \tag{6}$$

In the case $f = 0$, we obtain Laplace equation.
Our algorithm for solving Poisson equation (12) is as follows: we approximate u with

$$u(x, y; w_1, w_2) = A(x, y; w_1) + B(x, y) \cdot N(x, y; w_2)$$

where $A(x, y; w_1)$ is a neural network that approximates the boundary condition

$$A(x, y; w_1)|_{\partial\Omega} \approx g_D \tag{7}$$

and where $B(x, y)$ satisfies $B(x, y)|_{\partial\Omega} \equiv 0$. Our network looks like this fig. 2.

4 Experiments/Results/Discussion

To demonstrate the effectiveness of our method, we applied method of manufactured solution (MMS), i.e. construct some equations with analytic solutions and then compare the numerical solution with the analytic solutions. We consider problems on the unit square $\Omega = [0, 1]^2$, and the reference solution is computed on a uniform 50×50 grid. The test is on well-behaved solutions, solutions with peaks and some other less regular solutions. All of them were manufactured.

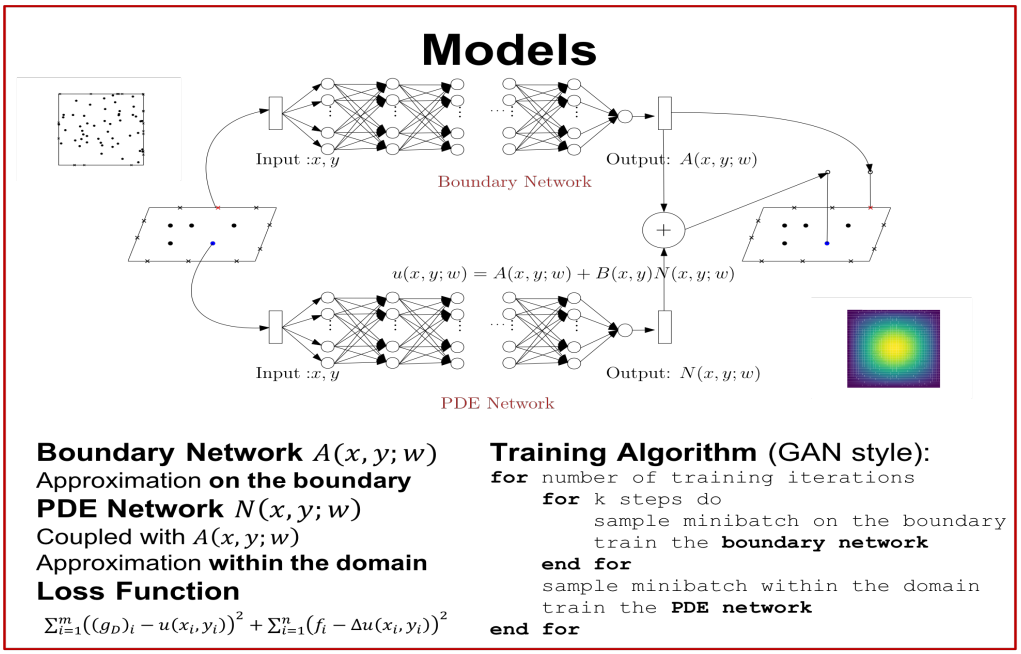


Figure 2: The structure of our network. We proposed this idea at the inspiration of GAN, where we update the boundary network for several times before we update the PDE network once. Our loss function is made up of two components: boundary loss and interior loss.

4.1 Example 1: Well-behaved Solution

Consider the analytical solution

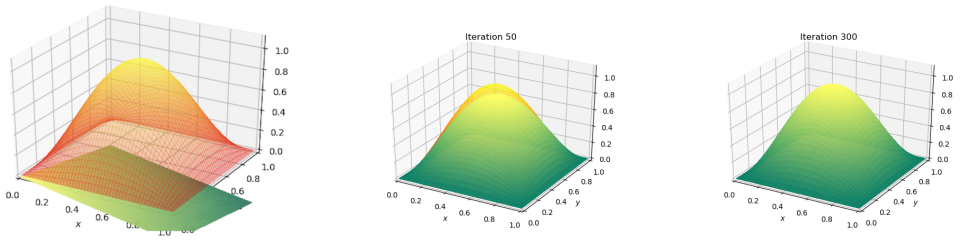
$$u(x, y) = \sin \pi x \sin \pi y, \quad (x, y) \in [0, 1]^2 \tag{8}$$

then we have

$$f(x, y) = \Delta u(x, y) = -2\pi^2 \sin \pi x \sin \pi y \tag{9}$$

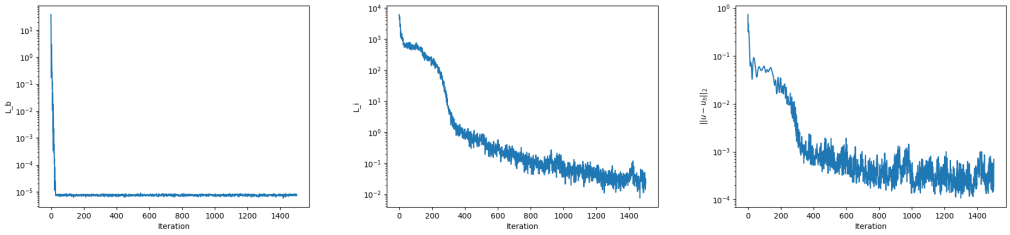
and the boundary condition is zero boundary condition.

The hyper-parameters for the network include $\varepsilon = 10^{-5}$, $L = 3$, $n^{[1]} = n^{[2]} = n^{[3]} = 64$. In figures below, the red profile shows exact solutions while the green profile shows the neural network approximation. Figure 3 *(a)* shows the initial profiles of the neural network approximation, while fig. 3 *(c)* shows that after 300 iterations it is nearly impossible to distinguish the exact solution and the numerical approximation.



(a) (b) (c)

Figure 3: Evolution of the neural network approximation. At 300 iteration, it is hard to distinguish numerical solution and exact solution by eyes.



(a) (b) (c)

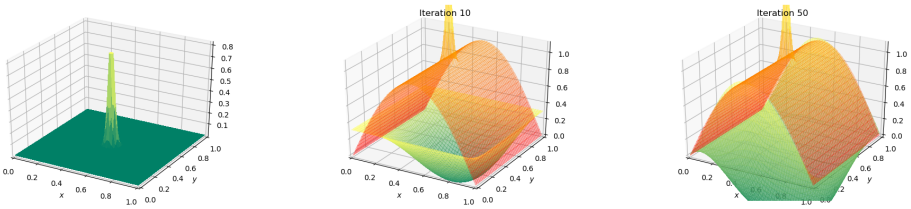
Figure 4: Loss and error of the neural network. For L_b, we have adopted an early termination strategy and therefore the error stays around 10^{-5} once it reaches the threshold. The PDE error L_i and L_2 error approximation keeps decreasing, indicating effectiveness of the algorithm.

4.2 Example 2: Solutions with a Peak

Consider the manufactured solution

$$u(x, y) = \exp[-1000(x - 0.5)^2 - 1000(y - 0.5)^2] + v(x, y) \tag{10}$$

where $v(x, y)$ is a well-behaved solution, such as the one in section 4.1. Figure 5 shows the plot of $\exp[-1000(x - 0.5)^2 - 1000(y - 0.5)^2]$. We classify this kind of function as 'ill-behaved' due to its dramatical change at a small area and therefore leads to very large Laplacian. This can be difficult for the neural network approximation to work. The gradient of the optimizer can be very large and a general step size will take the solution to badly behaved ones. As an illustration, let $v(x, y) = \sin(\pi x)$. If we do not do the singularity subtraction and run the neural network program directly, we obtain fig. 5. We see that the numerical solution diverges.



(a) (b) (c)

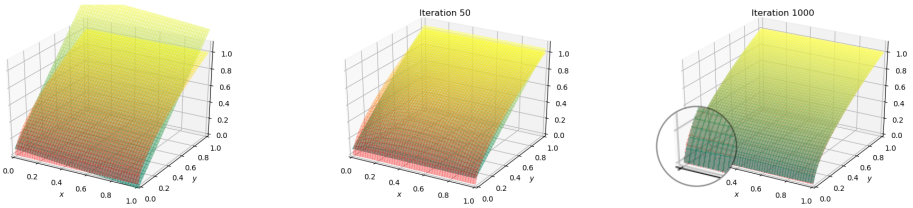
Figure 5: Plot of $\exp[-1000(x - 0.5)^2 - 1000(y - 0.5)^2]$, which has a peak at $(0.5, 0.5)$, and the evolution across iterations. This extreme behavior proposes difficulty for neural network approximation, since the Laplacian around $(0.5, 0.5)$ changes dramatically and can be very large. Solution converges on the boundary after several iterations but because of the peak, it diverges within the domain.

4.3 Example 3: Less Regular Solutions

Consider the following manufactured solution

$$u(x, y) = y^{0.6} \tag{11}$$

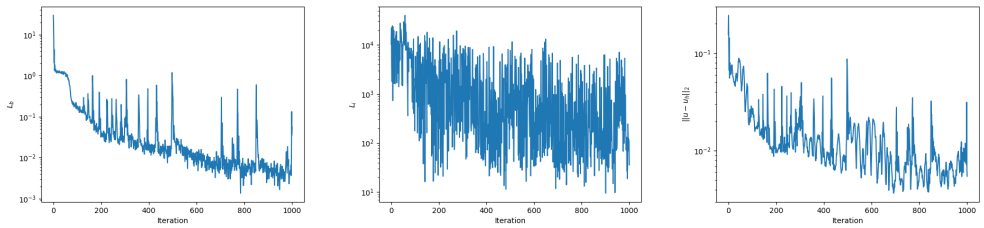
the derivative of the solution goes to infinity as $y \to \infty$, which makes the loss function hard to optimize. In the meanwhile, we notice that for any interval $[\delta, 1], \delta > 0$, the derivatives of $u(x, y)$ on $[0, 1] \times [\delta, 1]$ is bounded, that is to say, only the derivatives near $y = 0$ bring trouble to the optimization. Figure fig. 6 shows the initial profile of the neural network approximation. See Section 4.1 for detailed description of the surface and notation. The convergence is less satisfactory than section 4.1. Figure 8 shows the loss and error plot with respect to training iterations. The boundary loss did not reach the



(a) (b) (c)

Figure 6: Numerical solution at iteration 1000. We can see that near $y = 0$, there is data mismatch between numerical solution and exact solution and network approximation. At $y \approx 0$, the derivative $\frac{\partial u}{\partial y} \to \infty$. We can see the distortion between approximation solution and exact solution near $y = 0$.

threshold ε within 1000 iterations. Both the loss and the error show oscillatory behavior across the iterations, although they possess decreasing trends. This shows the difficulty when we deal with less regular solutions. Even at 1000 iteration, we can still observe obvious mismatch between the exact



(a) (b) (c)

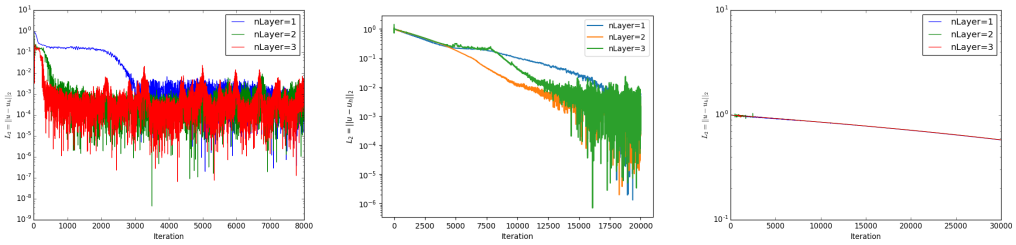
Figure 7: Loss and error of the neural network. The boundary loss did not reach the threshold ε within 1000 iterations. Both the loss and the error show oscillatory behavior across the iterations, although they possess decreasing trends.

solution and the numerical solution (fig. 6).

4.4 Higher dimensions

After our trial on 2D problem, we further tried 5D, 7D, 8D and 10D, varying the number of layers for each case. From the plots presented, we observe that, when dimension is small, increasing the number of layers does not help with the ultimate performance of our approximation. The convergence speed, however, increased when there are more layers in the network. As we see, this holds true for *(a)* and *(b)*, where dimensions are 2 and 5.

Our approximation for high dimensions is constrained by computation power, surprisingly. In 7D approximation, all settings show promises, as the L_2 loss is on the decline. But we do not see the convergence because the training iterations are not enough and obviously, to converge, we need a lot more steps. For 10D domain approximation, the case may be either the approximation does not help at all, or the converge takes so long that we do not have enough iterations to observe it. This result is reasonable, since for high dimensional problem, the curve itself is complex and difficult to approximate, and our network may simply do not have enough parameters to finish this task. We here do not have the space to plot for 8D and 10D approximation, the process runs so long even on AWS that it almost looks like the loss is not being reduced at all.



(a) (b) (c)

Figure 8: Performance of approximation for domains in different dimensions. In low dimensional problems, we see the convergence fast, and the more layers we have in the neural network, the faster convergence we observe; the number of layers do not affect the ultimate performance, measured in L_2 loss. The higher the dimensions are, the more difficult for us to see convergence, or in other words, the more iterations the approximation requires to achieve convergence. Notice in *(c)*, for 7D domain approximation, we cannot see the convergence in 30K iterations, but in general the loss is being reduced.

5 Conclusion/Future Work

In this project, we proposed novel deep learning approaches to solve the Lapalce equation,

$$\begin{cases} \triangle u = f, & \text{in } \Omega \\ u = g_D & \text{on } \partial\Omega \end{cases} \tag{12}$$

We also showed results for different kinds of solutions, and discussed the high dimension cases as well.
In the future, we will generalize results to other types of PDEs, and also investigate algorithms for ill-behaved solutions, such as peaks, exploding gradients, oscillations, etc.

6 Contributions

Kailai worked on mathematical formulation of the methods, while Bella and Shuyi worked on the model tuning and realization. The source code can be seen at https://github.com/kailaix/nnpde.

References

[1] Wang, Zixuan, et al. *Sparse grid discontinuous Galerkin methods for high-dimensional elliptic equations.* Journal of Computational Physics 314 (2016): 244-263.

[2] I.E. Lagaris, A. Likas and D.I. Fotiadis. *Artificial Neural Networks for Solving Ordinary and Partial Differential Equations*, 1997.

[3] Maziar Raissi. *Deep Hidden Physics Models, Deep Learning of Nonlinear Partial Differential Equations*, 2018.

[4] Justin Sirignano and Konstantinos Spiliopoulos. *DGM: A deep learning algorithm for solving partial differential equations*, 2007.

[5] Weinan E, Jiequn Han, and Arnulf Jentzen. *Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations*, 2017.

[6] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. *PDE-Net, Learning PDEs from data*, 2018.

[7] Modjtaba Baymani, Asghar Kerayechian, Sohrab Effati. *Artificial Neural Networks Approach for Solving Stokes Problem*, Applied Mathematics 2010.

[8] M.M. Chiaramonte and M. Kiener. *Solving differential equations using neural networks.*

[9] Peng, Shige, and Falei Wang. *BSDE, path-dependent PDE and nonlinear Feynman-Kac formula.* Science China Mathematics 59.1 (2016): 19-36.