Solving PDE with deep learning

Xinyu Xiao

Peking University

June 16, 2021

Table of Content

DGM method [5]

Consider a PDE:

$$\begin{cases} (\partial_t + \mathcal{L})\, u(t,\boldsymbol{x}) = 0, & (t,\boldsymbol{x}) \in [0,T] \times \Omega \\ u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}), & \boldsymbol{x} \in \Omega \\ u(t,\boldsymbol{x}) = g(t,\boldsymbol{x}), & (t,\boldsymbol{x}) \in [0,T] \times \partial\Omega \end{cases}$$

Neural network approximation

$$u(x,t) \approx \hat{u}(x,t;\theta)$$

DGM loss

$$\begin{aligned} J(\theta) &= loss_1 + loss_2 + loss_3 \\ loss_1 &= \left\| \frac{\partial \hat{u}}{\partial t}(t,x;\theta) + \mathcal{L}\hat{u}(t,x;\theta) \right\|^2_{[0,T]\times\Omega,\nu_1} \\ loss_2 &= \|\hat{u}(t,x;\theta) - g(t,x)\|^2_{[0,T]\times\partial\Omega,\nu_2} \\ loss_3 &= \|\hat{u}(0,x;\theta) - u_0(x)\|^2_{\Omega,\nu_3} \end{aligned}$$

Why DGM method?

In general, a Neural Network has form:

$$\hat{u}(x,\theta) = \sum_{i=1}^{m} a_i \varphi(x;\theta_i)$$

If we minimize

$$J(\theta) =< L\hat{u} - f, L\hat{u} - f >$$

Then we can get:

$$\frac{\partial J}{\partial a_i} =< L\hat{u} - f, \varphi_i >= 0,\ i = 1, 2, \ldots, m$$

which is exactly the Galerkin Method. So at this point of view, we can call this method as problem-driven spectral method.

Deep Ritz[2]

Consider the Poisson equation:

$$\begin{aligned} -\Delta u &= f, x \in \Omega \\ u &= g, x \in \partial\Omega \end{aligned}$$

We can solve the optimization problem to get solution

Ritz loss

$$J(u) = \int_{\Omega} \left(\frac{1}{2} |\nabla_x u(x)|^2 - f(x)u(x) \right) dx + \beta \int_{\partial\Omega} (u(x) - g(x))^2 ds$$

optimization

$$\hat{\theta} = argmin\, L(u(\cdot, \theta))$$

SGD

SGD

When we can not consider the full batch gradient descent, we can use a batch gradient descent:

$$L(\theta) = \int_{\Omega} h(x;\theta)dx \approx \frac{1}{N}\sum_{i=1}^{N} h(x_i,\theta)$$

$$\frac{\partial L}{\partial \theta} = \int_{\Omega} \frac{\partial}{\partial \theta} h(x;\theta)dx \approx \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial \theta} h(x_i,\theta)$$

where x_i are uniformly sampled on Ω.

PINN loss[3]

$$R(x,t) = u_t + Lu$$

$$loss = MSE_u + MSE_f$$

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} \left|u\left(t_u^i, x_u^i\right) - u^i\right|^2, \quad MSE_R = \frac{1}{N_f}\sum_{i=1}^{N_f} \left|R\left(t_f^i, x_f^i\right)\right|^2$$

calculate differential operator

In pytorch, we can calculate the differential operator in a direct manner.

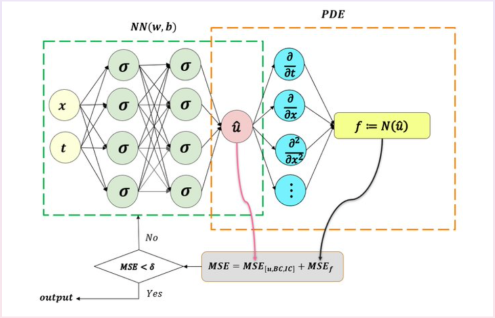

PDE
$NN(w,b)$
x
t
σ
$\hat{u}$
$\frac{\partial}{\partial t}$
$\frac{\partial}{\partial x}$
$\frac{\partial^2}{\partial x^2}$
$f \coloneqq N(\hat{u})$
No
$MSE < \delta$
$MSE = MSE_{[u,BC,IC]} + MSE_f$
Yes
output

Figure: PINN[3]

calculate differential operator

We can also calculate operator in a finite difference approximation.For example:

Finite diffrence

- time direction:Runge-kutta[3]

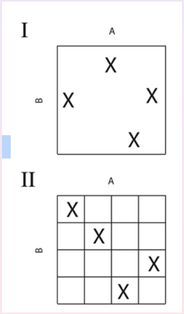
$$\begin{aligned} u^{n+c_i} &= u^n - \Delta t \textstyle\sum_{j=1}^q a_{ij} \mathcal{N}\left[u^{n+c_j}\right], \quad i = 1, \ldots, q \\ u^{n+1} &= u^n - \Delta t \textstyle\sum_{j=1}^q b_j \mathcal{N}\left[u^{n+c_j}\right] \end{aligned}$$

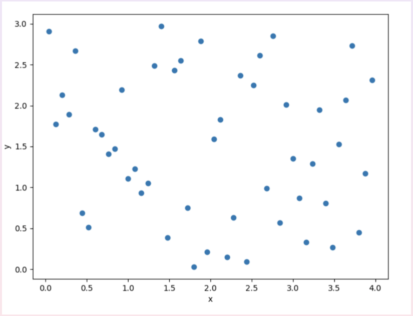
- space direction:[5]

$$\sum_{i,j=1}^{d} \rho_{i,j}\sigma_i(x)\sigma_j(x)\frac{\partial^2 f}{\partial x_i x_j}(t,x;\theta) = \lim_{\Delta\to 0}\mathbb{E}\left[\sum_{i=1}^{d}\frac{\frac{\partial f}{\partial x_i}(t,x+\sigma(x)W_\Delta;\theta) - \frac{\partial f}{\partial x_i}(t,x;\theta)}{\Delta}\sigma_i(x)W_\Delta^i\right]$$

sampling trick:Latin Hypercube Sampling strategy

In Latin Hypercube sampling is similar to having N rooks on a chess board without threatening each other.[Wiki]





We can use more sampling trick here such as importance sampling and MCMC.

Table of Content

Burgers equation

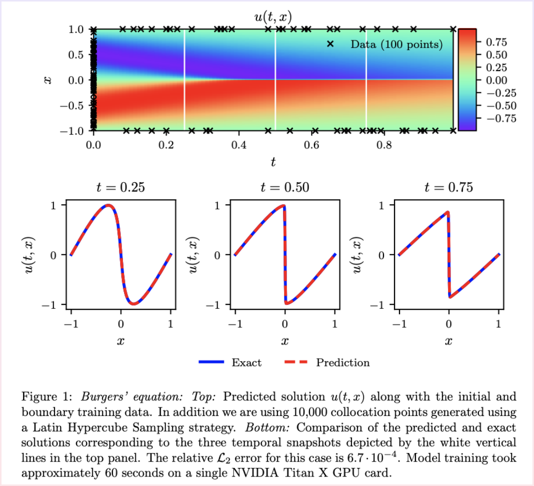
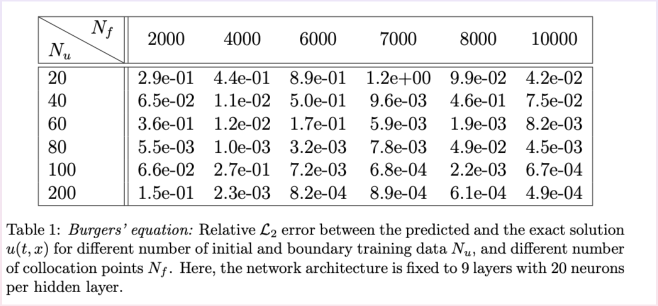


Figure 1: *Burgers' equation: Top:* Predicted solution $u(t, x)$ along with the initial and boundary training data. In addition we are using 10,000 collocation points generated using a Latin Hypercube Sampling strategy. *Bottom:* Comparison of the predicted and exact solutions corresponding to the three temporal snapshots depicted by the white vertical lines in the top panel. The relative $\mathcal{L}_2$ error for this case is $6.7 \cdot 10^{-4}$. Model training took approximately 60 seconds on a single NVIDIA Titan X GPU card.

Figure: result from [3]

Burgers equation:systematic studies

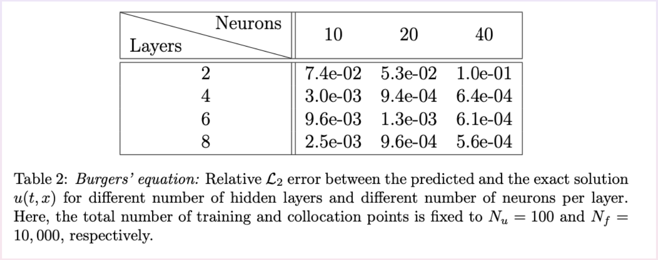


N_u \ N_f	2000	4000	6000	7000	8000	10000
20	2.9e-01	4.4e-01	8.9e-01	1.2e+00	9.9e-02	4.2e-02
40	6.5e-02	1.1e-02	5.0e-01	9.6e-03	4.6e-01	7.5e-02
60	3.6e-01	1.2e-02	1.7e-01	5.9e-03	1.9e-03	8.2e-03
80	5.5e-03	1.0e-03	3.2e-03	7.8e-03	4.9e-02	4.5e-03
100	6.6e-02	2.7e-01	7.2e-03	6.8e-04	2.2e-03	6.7e-04
200	1.5e-01	2.3e-03	8.2e-04	8.9e-04	6.1e-04	4.9e-04

Table 1: *Burgers' equation:* Relative $\mathcal{L}_2$ error between the predicted and the exact solution $u(t,x)$ for different number of initial and boundary training data N_u, and different number of collocation points N_f. Here, the network architecture is fixed to 9 layers with 20 neurons per hidden layer.

Figure: result from [3]

Burgers equation:systematic studies



Layers \ Neurons	10	20	40
2	7.4e-02	5.3e-02	1.0e-01
4	3.0e-03	9.4e-04	6.4e-04
6	9.6e-03	1.3e-03	6.1e-04
8	2.5e-03	9.6e-04	5.6e-04

Table 2: *Burgers' equation:* Relative $\mathcal{L}_2$ error between the predicted and the exact solution $u(t, x)$ for different number of hidden layers and different number of neurons per layer. Here, the total number of training and collocation points is fixed to $N_u = 100$ and $N_f = 10,000$, respectively.

Figure: result from [3]

A High dimensional case

Helmoholz equation

$\Omega = [0,1]^d$.

$$-\Delta u + \pi^2 u = f(x)$$

Definition (relative L2 error)

$$\text{error } = \left(\int_\Omega \left(u^*\left(x;\theta^*\right) - u(x)\right)^2 \, \mathrm{d}x\right)^{\frac{1}{2}} \cdot \left(\int_\Omega (u(x))^2 \, \mathrm{d}x\right)^{-\frac{1}{2}}$$

The paper [1] compared the performance of DGM and Deepritz.

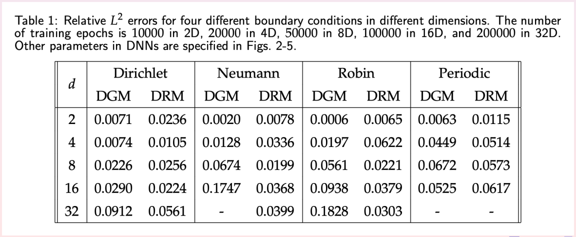


Table 1: Relative L^2 errors for four different boundary conditions in different dimensions. The number of training epochs is 10000 in 2D, 20000 in 4D, 50000 in 8D, 100000 in 16D, and 200000 in 32D. Other parameters in DNNs are specified in Figs. 2-5.

d	Dirichlet		Neumann		Robin		Periodic	
	DGM	DRM	DGM	DRM	DGM	DRM	DGM	DRM
2	0.0071	0.0236	0.0020	0.0078	0.0006	0.0065	0.0063	0.0115
4	0.0074	0.0105	0.0128	0.0336	0.0197	0.0622	0.0449	0.0514
8	0.0226	0.0256	0.0674	0.0199	0.0561	0.0221	0.0672	0.0573
16	0.0290	0.0224	0.1747	0.0368	0.0938	0.0379	0.0525	0.0617
32	0.0912	0.0561	-	0.0399	0.1828	0.0303	-	-

Disccusion

Advantage:

- meshfree: a path to high-dimensional PDE.
- handle complex region.

Disadvantage:

- No convergence theorem.
- Low precision compared with classical method(FEM,spectral method).

Table of Content

PFNN:Penalty free NN

While we do not want to introduce penalty terms for boundary conditions, we can use a penalty free method[4].

$$w_{\boldsymbol{\theta}}(\boldsymbol{x}) = g_{\boldsymbol{\theta}_1}(\boldsymbol{x}) + \ell(\boldsymbol{x}) f_{\boldsymbol{\theta}_2}(\boldsymbol{x})$$

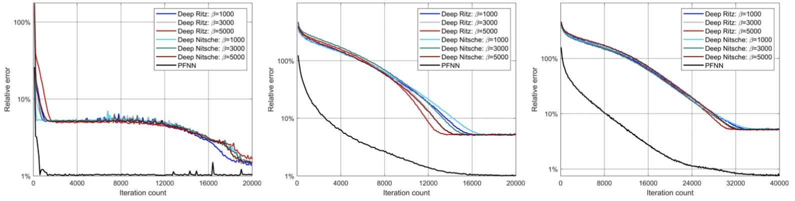
where $l(x)$ is a index function(perhaps a distance function to boundary).

$$\begin{cases} \ell(\boldsymbol{x}) = 0, & \boldsymbol{x} \in \Gamma_D \\ \ell(\boldsymbol{x}) > 0, & \text{otherwise.} \end{cases}$$

Two neural networks, one for boundary conditions, another for internal points. In this way, we can solve two optimization problems without constrain.

100D cube poission-like equtions

$$\begin{cases} -\nabla \cdot \left(|\nabla u|^{p-2} \nabla u \right) + u + c = 0, & \text{in } \Omega \\ u = \varphi, & \text{on } \Gamma_D \\ \left(|\nabla u|^{p-2} \nabla u \right) \cdot \boldsymbol{n} + \alpha u = \psi, & \text{on } \Gamma_R \end{cases}$$

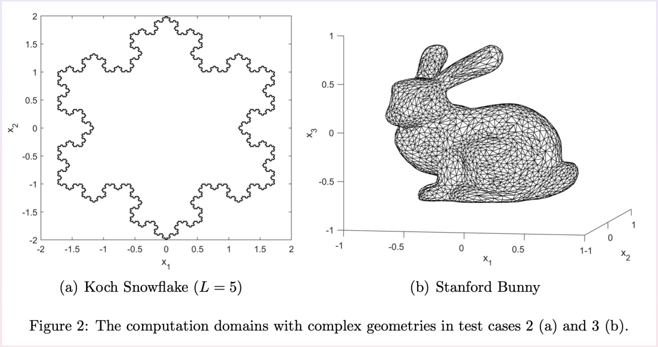


(a) $p = 1.2$ (b) $p = 3.6$ (c) $p = 4.8$

Figure 4: Error evolution history of various methods during the training process.

Figure: region

complex region



(a) Koch Snowflake ($L = 5$)

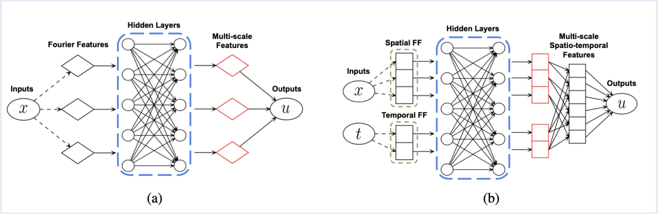
(b) Stanford Bunny

Figure 2: The computation domains with complex geometries in test cases 2 (a) and 3 (b).

Figure: region

Fourier feature embeddings

[6] show that we can introduce a fourier embedding layer to solve multi-scale problem.



Fourier embedding

$$\gamma^{(i)}(\boldsymbol{x}) = \begin{bmatrix} \cos\left(2\pi \boldsymbol{B}^{(i)}\boldsymbol{x}\right) \\ \sin\left(2\pi \boldsymbol{B}^{(i)}\boldsymbol{x}\right) \end{bmatrix}$$

[1] Jingrun Chen, Rui Du, and Keke Wu. "A Comparison Study of Deep Galerkin Method and Deep Ritz Method for Elliptic Problems with Different Boundary Conditions". In: *Communications in Mathematical Research* 36.3 (2020), pp. 354–376. ISSN: 2707-8523. DOI: https://doi.org/10.4208/cmr.2020-0051. URL: http://global-sci.org/intro/article_detail/cmr/17853.html.

[2] Weinan E and Bing Yu. *The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems*. 2017. arXiv: 1710.00211 [cs.LG].

[3] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[4] Hailong Sheng and Chao Yang. "PFNN: A penalty-free neural network method for solving a class of second-order boundary-value problems on complex geometries". In: *Journal of Computational Physics* 428 (Mar. 2021), p. 110085. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.110085. URL: http://dx.doi.org/10.1016/j.jcp.2020.110085.

[5] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations". In: *Journal of Computational Physics* 375 (Dec. 2018), pp. 1339–1364. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.08.029. URL: http://dx.doi.org/10.1016/j.jcp.2018.08.029.

[6] Sifan Wang, Hanwen Wang, and Paris Perdikaris. *On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks*. 2020. arXiv: 2012.10047 [cs.LG].

unsrt