Learning to Optimize Multigrid PDE Solvers

Shengyuan Wang
ID: 2024234346
wangshy2024@shanghaitech.edu.cn

Zhen Ji
ID:2024234272
jizhen2024@shanghaitech.edu.cn

Zhongqiang Zheng
ID:2024234374
zhengzhq2024@shanghaitech.edu.cn

Guideline

In this project, our group selected a paper based on neural network training PDE solver, reproduced and trained the paper, evaluated the performance of the PDE solver, and solved the problem proposed in the third assignment, compared the results with the results obtained by tifiss1.2, and evaluated the corresponding errors of the two methods. In the introduction part, PDE and some related knowledge are introduced, some core ideas of the original paper are introduced in the method part, the work we have done is introduced in our work part, the results obtained by the training of this paper are shown in the experiment part and compared with the training effect of the original paper, and the solution results and errors of the third job are compared. Our findings and conclusions are presented in the conclusion section.

1 Introduction

PDE and Multigrid methods

- **PDE:** Partial Differential Equations (PDEs) are a key tool for modeling diverse problems in science and engineering. In all but very specific cases, the solution of PDEs requires carefully designed numerical discretization methods, by which the PDEs are approximated by algebraic systems of equations. Practical settings often give rise to very large ill-conditioned problems, e.g., in predicting weather systems, oceanic flow, image and video processing, aircraft and auto design, electromagnetics, to name just a few. Developing efficient solution methods for such large systems has therefore been an active research area since many decades ago.
- **Multigrid Methods:** Multigrid methods are leading techniques for solving largescale discretized PDEs, as well as other large-scale problems. Introduced about half a century ago as a method for fast numerical solution of scalar elliptic boundary-value problems, multigrid methods have since been developed and adapted to problems of increasing generality and applicability. Despite their success, however, applying off-the-shelf multigrid algorithms to new problems is often non-optimal. In particular, new problems often require expertly devised prolongation operators, which are critical to constructing efficient solvers. This paper demonstrates that machine learning techniques can be utilized to derive suitable operators for wide classes of problems. Accuracy refers to the degree of consistency between the clustering results and the true image categories. High accuracy means that the clustering algorithm can effectively group similar images into the same category while correctly separating images of different categories. Scalability, on the other hand, focuses on the performance of the algorithm when dealing with large-scale datasets, particularly in terms of computational efficiency and resource requirements. A clustering algorithm with good scalability can process massive image

data efficiently without significant degradation in performance as the dataset size increases. Semantic understanding refers to the algorithms deep understanding of the image content, meaning it can perform reasonable clustering based on semantic information in the image, rather than relying solely on shallow features such as color or texture. This directly affects whether the algorithm can capture higher-level meanings and relationships in the image.

- **Scope:** We train a single deep network once to handle any diffusion equation whose (spatially varying) coefficients are drawn from a given distribution. Once our network is trained it can be used to produce solvers for any such equation. Our goal in this paper, unlike existing paradigms, is not to learn to solve a given problem, but instead to learn compact rules for constructing solvers for many different problems.
- **Unsupervised training:** The network is trained with no supervision. It will not be exposed to ground truth operators, nor will it see numerical solutions to PDEs. Instead, our training is guided by algebraic properties of the produced operators that allude to the quality of the resulting solver.
- **Generalization:** While our method is designed to work with problems of arbitrary size, it will suffice to train our system on quite small problems. This will be possible due to the local nature of the rules for determining the prolongation operators. Specifically, we train our system on block periodic problem instances using a specialized block Fourier mode analysis to achieve efficient training. At test time we generalize for size (train on 32×32 grid and test on a 1024×1024 grid), boundary conditions (train with periodic BCs and test with Dirichlet), and instance types (train on block periodic instances and test on general problem instances).

2 Related Work

A number of recent papers utilized NN to numerically solve PDEs, some in the context of multigrid methods. Starting with the classical paper of (Lagaris et al., 1998), many suggested to design a network to solve specific PDEs (Hsieh et al., 2019; Baqu\'e et al., 2018; Baymani et al., 2010; Han et al., 2018; Katrutsa et al., 2017; Mishra, 2018; Sirignano & Spiliopoulos, 2018; Sun et al., 2003; Shan et al., 2017; Wei et al., 2019), generalizing for different choices of right hand sides, boundary conditions, and in some cases to different domain shapes. These methods require separate training for each new equation.

Some notable approaches in this line of work include (Shan et al., 2017), who learn to solve diffusion equations on a fixed grid with variable coefficients and sources drawn randomly in an interval. A convolutional NN is utilized, and its depth must grow (and it needs to be retrained) with larger grid sizes. (Hsieh et al., 2019) proposes an elegant learning based approach to accelerate existing iterative solvers, including multigrid solvers. The method is designed for a specific PDE and is demonstrated with the Poisson equation with constant coefficients. It is shown to generalize to domains which differ from the training domain. handle complex domain geometries by penalizing the PDE residual on collocation points. (Han et al., 2018; Sirignano & Spiliopoulos, 2018) introduce efficient methods for solving specific systems in very high dimensions. (Mishra, 2018) aims to reduce the error of a standard numerical scheme over a very coarse grid. (Sun et al., 2003) train a neural net to solve the Poisson equation over a surface mesh. (Baqu\'e et al., 2018) learn to simulate computational fluid dynamics to predict the pressures and drag over a surface. (Wei et al., 2019) apply deep reinforcement learning to solve specific PDE instances. (Katrutsa et al., 2017) use a linear NN to derive optimal restriction/prolongation operators for solving a single PDE instance with multigrid. The method is demonstrated on 1D PDEs with constant coefficients. The tools suggested, however, do not offer ways to generalize those choices to other PDEs without re-training.

More remotely, several recent works, e.g., (Chen et al., 2019; Chang et al., 2018) suggest an interpretation of neural networks as dynamic differential equations. Under this continuous representation, a multilevel strategy is employed to accelerate training in image classification tasks.

3 Method

Multigrid background and problem setting

Basic Framework: We focus on the classical second-order elliptic diffusion equation in two dimensions,

$$-\nabla \cdot (g\nabla u) = f$$

over a square domain, where g and f are given functions, and the unknown function u obeys some prescribed boundary. The equation is discretized on a square grid of $n \times n$ grid cells with uniform mesh-size h. The discrete diffusion coefficients g are defined at cell centers, while the discrete solution vector u and the discrete right-hand side vector f are located at the vertices of the grid, as illustrated in the 3×3 sub-grid depicted in Fig.1.

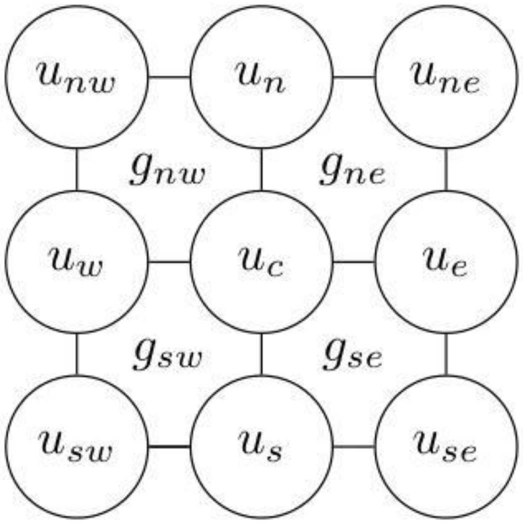


Figure 1. Sub-grid of 3×3. The discrete diffusion coefficients g are defined at cell centers. The discrete solution u and the discrete right hand side f are located at the vertices of the grid. The discrete equation for u_c has nine non-zero coefficients multiplying the unknowns u_c and its eight neighbors.

Employing bilinear finite element discretization, we obtain the following equation associated with the variable u_c,

$$\begin{aligned}
&-\frac{1}{3h^2}\left(g_{n,w}u_{n,w} + g_{n,e}u_{n,e} + g_{s,w}u_{s,w} + g_{s,w}u_{s,w}\right) \\
&-\frac{1}{6h^2}\{\left(g_{n,w} + g_{n,e}\right)u_n + \left(g_{n,e} + g_{s,e}\right)u_e + \left(g_{s,e} + g_{s,w}\right)u_s \\
&+ \left(g_{s,w} + g_{n,w}\right)u_w)\} + \frac{2}{3h^2}\left(g_{n,w} + g_{n,e} + g_{s,e} + g_{s,w}\right)u_c = f_c
\end{aligned}$$

Arranging these equations in matrix-vector form, we obtain a linear system

$$Au = f$$

Where $A_{c,j}$ is the coefficient multiplying u_j in the discrete equation associated with u_c. The term will refer to the 3×3 set of coefficients associated with the equation for u_c.

Multigrid Cycle: A course grid is defined by skipping every other mesh point in each coordinate, obtaining a grid of $\frac{n}{2} \times \frac{n}{2}$ grid cells and mesh-size 2h. A prolongation operator P is defined and it can be represented as a sparse matrix whose number of the rows is equal to the size of u and the number of columns is equal to the number of course-grid variables, approximately$\left(\frac{n}{2}\right)^2$.The two-grid version of the multigrid algorithm proceeds by applying one or my relaxation sweeps on the fine grid, obtaining an approximation $\tilde{u}$ to u, such that the remaining error, $u - \tilde{u}$ is smooth and can therefore be approximated well on the coarse grid. The linear system for the error is then projected to the coarse grid by the Galerkin method as follows. The coarse grid operator is defined as P^TAP and the right-hand-side is the restriction of the residual to the coarse grid, $P^T(f - A\tilde{u})$. This is typically followed by one or more additional fine grid relaxation sweeps, This entire process comprises a single two-grid iteration as formally described in Algorithm 1.

Algorithm 1 Two-Grid Cycle

```
1: Input: Discretization matrix $A$, initial approximation
   $u^{(0)}$, right-hand side $f$, prolongation matrix $P$, a relax-
   ation scheme, $k = 0$, residual tolerance $\delta$
2: repeat
3:     Perform $s_1$ relaxation sweeps starting with the cur-
       rent approximation $u^{(k)}$, obtaining $\tilde{u}^{(k)}$
4:     Compute the residual: $r^{(k)} = f - A\tilde{u}^{(k)}$
5:     Project the error equations to the coarse grid and
       solve the coarse grid system: $P^T APv^{(k)} = P^T r^{(k)}$
6:     Prolongate and add the coarse grid solution: $\tilde{u}^{(k)} =$
       $\tilde{u}^{(k)} + Pv^{(k)}$
7:     Perform $s_2$ relaxation sweeps obtaining $u^{(k+1)}$
8:     $k = k + 1$
9: until $r^{(k-1)} < \delta$
```

In the multigrid version of the algorithm, Step 5 is replaced by one or more recursive call yields the so-called multigrid V cycle, whereas two calls yield the W cycle. These recursive calls the repeated until reaching a very coarse grid, where the problem is solved cheaply by relaxation or an exact solve. The entire multigrid cycle thus obtained has linear computational complexity. The W cycle is somewhat more expensive than the V cycle but may be cost-effective in particularly challenging problems.

The error propagation equation of the two-grid algorithm is given by

$$e^{(k)} = Me^{(k-1)}$$

Where $M = M(A, P; S, s_1, s_2)$ is the two-grid error propagation matrix

$$M = S^{s_1} C S^{s_2}$$

Here, s_1 and s_2 are the number of relaxation sweeps performed before and after the coarse-grid correction phase, and the error propagation matrix of the coarse grid correction is given by

$$C = (I - P[P^T AP]^{-1})P^T A$$

Neural network building and introduction

Basic framework: we propose a scheme for learning a mapping from discretization matrices to prolongation matrices. We assume that the diffusion coefficients are drawn from some distribution $\mathcal{D}$ over the discretization matrices. A natural objective would be to seek a mapping that minimizes the expected spectral radius of the error propagation matrix $M(A, P)$, which governs the asymptotic convergence rate of the multigrid solver. Concretely, we represent the mapping with a neural network parameterized by θ that maps discretization matrices A to prolongations $P_\theta(A) \in \mathcal{P}$ with a predefined sparsity pattern. The relaxation scheme S is fixed to be Gauss-Seidel, and the parameters s_1, s_2 are set to 1. Thus, we arrive at the following learning problem:

$$min_{P_\theta \in \mathcal{P}} E_{A \sim D} \rho(M(A, P_\theta(A)))$$

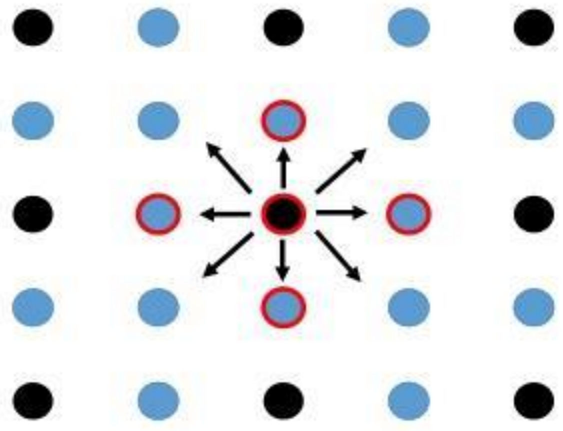


Figure 2. The input and the output of the network. The discs denote the (fine) grid points, where the black discs mark the subset of points selected as coarse grid points. The input of the network consists of the 3 × 3 stencils of the five points, denoted by the red cycles. The black arrows illustrate the output of the network, i.e., the contribution of the prolongation of one coarse point to its eight fine grid neighbors.

where $\rho(M)$ is the spectral radius of the matrix M, and $\mathcal{D}$ is the distribution over the discretization matrices A.

Inferring P from local information: The network we construct receives an input vector of size

45, consisting of a local subset of the discretization matrix A, and produces an output that consists of 4 numbers, which in turn determine the 9 nonzero entries of one column of the prolongation matrix P. Existing multigrid solvers for diffusion problems on structured grid, infer the prolongation weights from local information. Following their approach, we construct our network to determine each column j of P from five 3 × 3 stencils. Specifically, the input to the network is composed of the stencil of the fine grid point coinciding with coarse point j, and the stencils of its four immediate neighbors, marked by the red circles in Fig. 2.

For the output we note that the sparsity pattern imposed on P implies that each column has at most nine non-zero elements, where each non-zero element $P_{i,j}$ is the prolongation weight of the coarse grid point j to a nearby fine grid point i . Geometrically, this means that a coarse grid point contributes only to the fine-grid point with which it coincides (and the corresponding prolongation coefficient is set to 1) and to its eight fine-grid neighboring points, as illustrated in Fig. 2. Only the four prolongation coefficients corresponding to the nearest neighbors are learned; the four remaining prolongation coefficients, marked by diagonal arrows in Fig. 2, are then calculated such that any grid function u obtained by prolongation from the coarse grid satisfies $Au = 0$ at these four grid points. The complete prolongation matrix P is constructed by applying the same network repeatedly to all the coarse points.

The inference from local information maintains the efficiency of the resulting multigrid cycle, as the mapping has constant time computation per coarse grid point, and we construct P by applying the network repeatedly to all coarse grid points. Moreover, the local nature of the inference allows application of the network on different grid-sizes. Further details are provided in Section 4.

Fourier analysis for efficient training: The fact that the network determines P locally does not mean that it suffices to train on very small grids. Because the method is to be used for large problems, it is critical that the subspace spanned by the columns of P will approximate well all algebraically smooth errors of large problems, as discussed. This implies that such errors should be encompassed in the loss function of the training phase. In practice, our experiments show that good performance on large grids is already obtained after training only on a 32×32 grid, which is not very large but still results in an error propagation matrix M of size 1024×1024.

The main computational barrier of the loss is due to the coarse-grid correction matrix C , whose computation requires inversion of the matrix $P^T AP$ of size $(\frac{n}{2})^2 \times (\frac{n}{2})^2$ elements. To overcome this prohibitive computation, we introduce two surrogates. First, we relax the spectral radius of the error propagation matrix with its squared Frobenious norm, relying on the fact that the Frobenious norm bounds the spectral radius from above, yielding a differentiable quantity without the need for (expensive) spectral decomposition. Secondly, we train on a relatively limited class of discretization matrices A, which are called block-circulant matrices, allowing us to train efficiently on large problems, because it requires inversion only of small matrices, as explained below. Due to the local dependence of P on A, we expect that the resulting trained network would be equally effective for general (non-block-periodic) A, and this is indeed borne out in our experiments.

The block-periodic framework allows us to train efficiently on large problems. To do so, we exploit a block Fourier analysis technique that was recently introduced independently in several variants and for different applications. Classical Fourier analysis has been employed for quantitative prediction of two-grid convergence factors since the 1970s. This technique, however, is exact only in very special cases of constant-coefficient operators and simple boundary conditions. Here, in contrast, we need to cater to arbitrary discrepancies in the values of the diffusion coefficients of neighboring grid cells, which imply strongly varying coefficients in the matrix A, so classical Fourier analysis is not appropriate.

To apply the new block Fourier analysis, we partition our $n \times n$ grid into equal-sized square blocks of $c \times c$ cells each, such that all the $\frac{n}{c} \times \frac{n}{c}$ blocks are identical with respect to their cell g values, but within the block the g values vary arbitrarily, according to the original distribution. This can be thought of as tiling the domain by identical blocks of $c \times c$ cells. Imposing periodic boundary conditions, we obtain a discretization matrix A that is block-circulant. Furthermore, due to the dependence of on A, the matrix M itself is similarly block-circulant and can be written as

$$M = \begin{bmatrix} M_0 & M_1 & \dots & M_{b-2} & M_{b-1} \\ M_{b-1} & M_0 & M_1 & \dots & M_{b-2} \\ M_{b-2} & M_{b-1} & M_0 & \dots & M_{b-3} \\ \dots & \dots & \dots & \dots & \dots \\ M_1 & \dots & M_{b-2} & M_{b-1} & M_0 \end{bmatrix}$$

where M_j , $j = 0, \dots, b-1$, are $c^2 \times c^2$ blocks and $b = \frac{n^2}{c^2}$ .This special structure has the following important implication. M can easily be block-diagonalized in a way that each block of size $c^2 \times c^2$ on the diagonal has a simple closed form that depends on the elements of A and a single parameter associated with a certain Fourier component. As a result, the squared Frobenius norm of the matrix M, which constitutes the loss for our network, can be decomposed into a sum of squared Frobenius norms of these small easily computed blocks, requiring only the inversion of relatively small matrices.

The theoretical foundation of this essential tool is summarized briefly below.

Block diagonalization of block circulant matrices: Let the $n \times n$ matrix K be block-circulant, with b blocks of size k. That is, $n = bk$, and the elements of K satisfy:

$$K_{l,j} = K_{mod(l-k,n),mod(j-k,n)}$$

with rows, column, blocks, etc., numbered starting from 0 for convenience. Here, we are adopting the MATLAB form $mod(x, y)$ = "x modulo y", i.e., the remainder obtained when dividing integer x by integer y . Below, we continue to use l and j to denote row and column numbers, respectively, and apply the decomposition:

$$l = l_0 + tk, \quad j = j_0 + sk$$

Where $l_0 = mod(l,k), t = \left[\frac{l}{k}\right], j_0 = mod(j,k), s = \left[\frac{j}{k}\right]$. Note that $l, j \in \{0, \dots, n-1\}$; $l_0, j_0 \in \{0, \dots, k-1\}$; $t, s \in \{0, \dots, b-1\}$

Let the column vector

$$v_m = \left[1, e^{i\frac{2\pi m}{n}}, \dots, e^{i\frac{2\pi m j}{n}}, \dots, e^{i\frac{2\pi m(n-1)}{n}}\right]^*$$

Denote the unnormalized mth Fourier component of dimension n, where $m = 0, \dots, n-1$. Finally, let W denotes the $n \times n$ matrix whose nonzero values are comprised of the elements of the first b Fourier components as follows:

$$W_{l,j} = \frac{1}{\sqrt{b}} \delta_{l0,j0} v_s(l)$$

Where $v_s(l)$ denotes the lth element of v_s, and δ is the Kronecker delta. Then we have:

Theorem 1. W is a unitary matrix, and the similarity transformation $\widehat{K} = W * KW$ yields a block-diagonal matrix with b-blocks of size $k \times k$, $\widehat{K} = \text{blockdiag}\left(\widehat{K}^{(0)}, \dots, \widehat{K}^{(b-1)}\right)$. Furthermore, if K is band-limited modulo n such that all the nonzero elements in the lth row of $K, l = 0, \dots, n-1$, are included in $\left\{K_{l,mod(l-\alpha,n)}, \dots, K_{l,l}, \dots, K_{l,mod(l+\beta,n)}\right\}$ and $\alpha + \beta + 1 \leq k$, then the nonzero elements of the blocks are simply

$$\widehat{K}^{s}_{l0,mod(l0+m,k)} = e^{-i\frac{2\pi sm}{n}} K_{l,mod(l0+m,n)},$$
$$l_0 = 0, \dots, k-1, \qquad m = -\alpha, \dots, \beta$$

We finally compute the loss

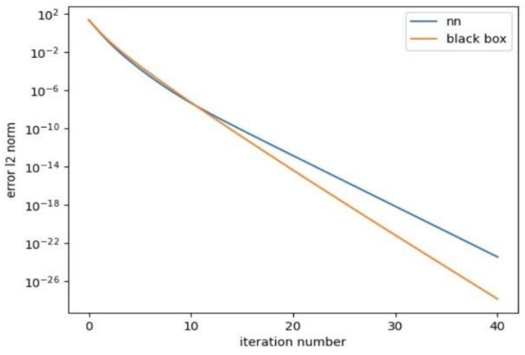
$$\|M\|_F^2 = \left\|\widehat{M}\right\|_F^2 = \sum_{s=0}^{b-1} \left\|\widehat{M}^{(s)}\right\|_F^2$$

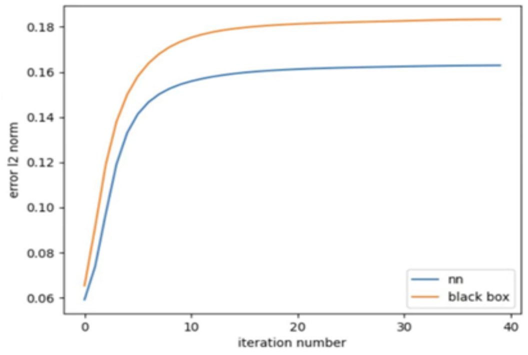
Where $\widehat{M} = \text{blockdiag}\left(\widehat{M}^{(s)}, \dots, \widehat{M}^{(b-1)}\right)$. Note that, $\left\|\widehat{M}^{(s)}\right\|_F^2$ is cheap to compute since $\widehat{M}^{(s)}$ is of size $c^2 \times c^2$.

4 Our Work

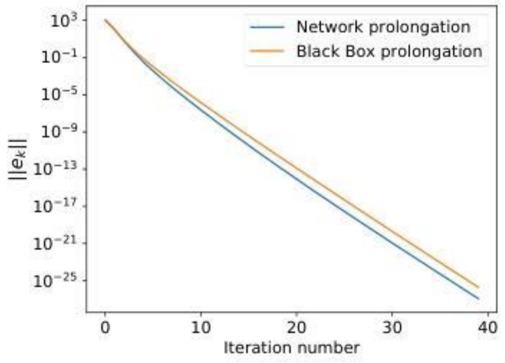
Train the model

The code of the paper has been open sourced on Github, and we have reproduced the paper on the Tensflow platform, and we have obtained the following results:





The results obtained from the original thesis training are as follows:



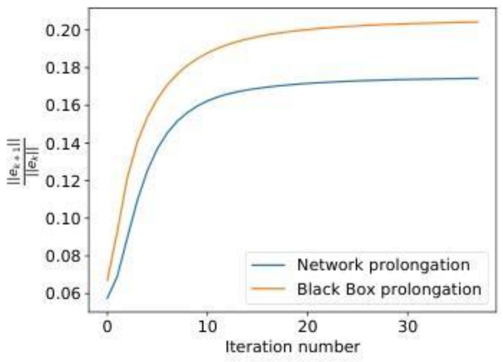


Figure 3. W-cycle performance, averaged over 100 problems with grid size 1024 × 1024 and Dirichlet Boundary conditions. Left: error norm as a function of iterations (W cycles). Right: error norm reduction factor per iteration.

The training results we obtained were slightly worse than those in the paper, either because the original paper used a specific processing method, such as additional data augmentation or specific parameter tuning details, or because the original paper was retained and presented the best results after multiple training sessions

We want to use this model to solve some of the problems we have learned, in order to combine with the content of our class, we chose the example problems of the third assignment to test, hoping to get the error of the model's predicted solution and the real solution when solving some practical problems, and compare it with the error when using ifiss3.6 provided in the classroom to solve the actual problem, so as to have a deeper understanding of the function of the PDE solver designed in this paper.

Code Writing

Since the original paper used to test the performance of the model is to compare the model with the Black Box prolongation method, and we expect it to solve the same problem and compare the accuracy with ifiss3.6, we write a code to solve the problem on the third job example, we choose to generate a mesh of 32×32, and then discretize the right end term, generate a vector corresponding to the mesh, and then call the model to solve, The Prolongation matrix is generated by the model, and the multi-mesh method is used to accelerate the solution, and finally the results are saved and visualized to calculate the error.

Considering that the idea of this paper is to iterate on the initial mesh with thick and thin meshes, it is not fair to set the same mesh size in tifiss1.2, so we define the mesh size as 64×64 in tifiss1.2, which is twice the mesh size when solving with the PDE solver.

Partial differential equation selection

We used a total of four equations for the test，Use separately problem1，problem2，problem3，problem4 numbering. The specific equation is as follows:

Problem1

$$U_{xx} + U_{yy} = -32\pi^2 \sin(4\pi x)\cos(4\pi y) \quad for\ x, y \in \Omega$$

$$U = \sin(4\pi x)\cos(4\pi y) \quad for\ x, y \in \Omega$$

Problem2

$$U_{xx} + U_{yy} = -8\pi^2 \sin(2\pi x)\cos(2\pi y) \quad for\ x, y \in \Omega$$

$$U = \sin(2\pi x)\cos(2\pi y) \quad for\ x, y \in \Omega$$

Problem3

$$U_{xx} + U_{yy} = -32\pi^2 \sin(4\pi x)\sin\,(4\pi y)\, for\ x, y \in \Omega$$

$$U = \sin(4\pi x)\sin(4\pi y) \quad for\ x, y \in \Omega$$

Problem4

$$U_{xx} + U_{yy} = -32\pi^2 \cos(4\pi x)\cos\,(4\pi y)\, for\ x, y \in \Omega$$

$$U = \cos(4\pi x)\cos(4\pi y) \quad for\ x, y \in \Omega$$

Where $\Omega = (0,1) \times (0,1)$

The four equations selected are very similar in form, and the last three are all variations of the example problem in the third job, and we set the problem so similar to each other mainly to test how well the PDE solver can solve some equations with very small changes, and whether it can give the correct solution.

5 Experiment

Software and hardware dependencies: The experimental environment used Tensflow with 2 RTX 3090 GPUs.

Experimental process: Comparing the visualization results of using the PDE solver and using ifiss3.6 to solve the third job, although the visualization results are theoretically the same, this can help us to preliminarily judge whether the PDE solver can solve this problem, because if the visualization results are different, it proves that there must be something wrong in the code and needs to be modified. Then, comparing the errors of the two, we selected four problems, in addition to the problem of the third job, there is also a simple deformation of the problem, in order to better evaluate the error of the PDE solver to solve this problem and tifiss1.2 to solve this problem. The error is presented in a tabular format.

Comparison of visualization results: Take problem1 as an example.

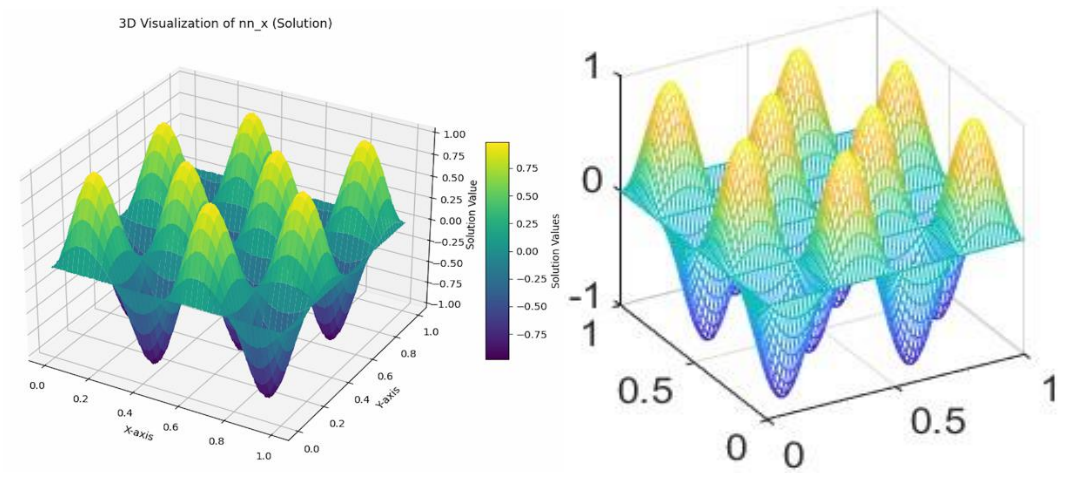


Figure 4. The left figure is the visualization result generated by the PDE solver, and the right figure is the visualization result generated by ifiss3.6, you can see that their visualization results are the same, the value of u is between (0, 1), which is very close to the real solution, which shows that the PDE solver can indeed solve the problem, but there may be some differences in accuracy

Comparison and analysis of errors: The following table is the statistical results of the errors of the four methods on the four problems, and it is obvious that the V-cycle has been able to achieve a good accuracy, and the W-cycle is far more than the other three methods, but considering that the W-cycle itself is a cyclic iterative process, it is not surprising that the accuracy is very high, but when we compare it with the original paper, we find that the accuracy of the original paper can even reach $^{10-19}$ I think it may be caused by the poor training of our model, and I think that if the training effect reaches the level of the original paper, the accuracy will be greatly increased.

Method	Problem1	Problem2	Problem3	Problem4
v-cycle	1.6702×10^{-7}	6.5742×10^{-8}	1.6714×10^{-7}	1.6686×10^{-7}
w-cycle	3.7276×10^{-10}	6.3744×10^{-11}	3.7346×10^{-10}	3.7270×10^{-10}
P1 element	0.0013	3.0195×10^{-4}	0.0013	0.0012
P2 element	2.8432×10^{-5}	3.5278×10^{-6}	2.8486×10^{-5}	2.8374×10^{-5}

We also found a very interesting thing, the accuracy of problem1 is very close to problem3 and problem4, they are only somewhat similar in structure, they all use the multiplication of sine and cosine functions, but problem1 is a combination of sine function and cosine function, problem 2 is a combination of sine function and sine function, problem 3 is a combination of cosine function and cosine function, and their visualization results are not the same. But the error between their approximate solution and the real solution is strikingly similar. On the contrary, problem2 is more similar to problem1 in structure, only changing the period of the sine function and the cosine function, which has a great impact on the error, and the larger the period, the smaller the error value. We think this may be related to the smoothness of the plane formed by the exact solution of the problem, as shown in the figure below:

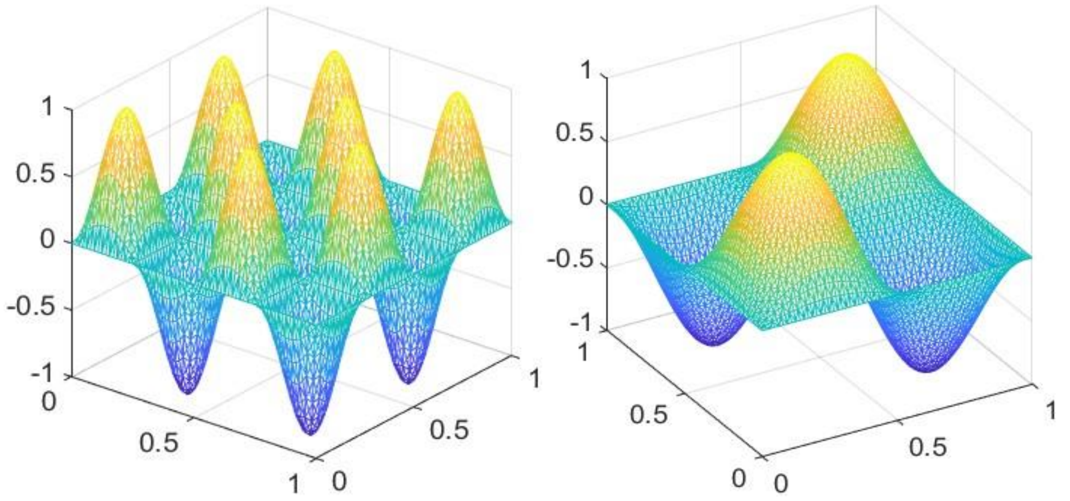


Figure 5. The left image is the visualization of problem 1, the right image is the visualization of problem 2, the left image reaches more peaks, and the right image appears smoother in comparison, which we think makes the approximate solution better fit the real solution.

6 Conclusion

On the whole, The error of the approximate solution and the real solution obtained by the PDE solver to solve the diffusion problem is smaller than that obtained by tifiss1.2, which means that the approximate solution obtained by the PDE solver is closer to the real solution, the advantage of the PDE solver obtained by training is that it can solve a class of problems efficiently and with high quality and not only limited to a certain problem, the disadvantage is that it takes too long to train the model, the advantage of tifiss1.2 is that it does not need to be trained, it can be used only after downloading and installing and setting the path, and it can also ensure high accuracy when solving the PDE problems it supports. The downside is that it can only solve the PDE problems that it supports to solve, because it is a teaching library function. Therefore, when we need to solve some common problems in daily life, we can easily solve them with tifiss1.2, and when we want to solve a certain type of problem with higher accuracy, we can train a neural network, use multi-mesh iteration method or other methods to help us get a better approximate solution

7 Contribution Percent

Shengyuan Wang（35%）　　Zhen Ji（35%）　　Zhongqiang Zheng(30%)

1.Paper Reading:

Responsible for: Shengyuan Wang, Zhen Ji, Zhongqiang zheng

2.Paper Reproduction:

Responsible for: Shengyuan Wang, Zhen Ji

3.Inprovements-ideas:

Responsible for: Shengyuan Wang, Zhen Ji

4.Inprovements-code:

Responsible for: Shengyuan Wang

5.English Report Writing:

Responsible for: Zhen Ji, Zhongqiang zheng

6.Presentation and Proposal Preparation and PPT:

Responsible for：Zhen Ji, Zhongqiang zheng

8 Reference

I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans Neural Netw,* 9(5):987-1000, 1998.

Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning neural PDE solvers with convergence guarantees. *arXiv preprint arxiv-1906.01200*, 2019.

Pierre Baqu\'e, Edoardo Remelli, Fran\c{c}ois Fleuret, and Pascal Fua. Geodesic convolutional shape optimization. *arXiv preprint arxiv-1802.04016*, 2018.

Modjtaba Baymani, Asghar Kerayechian, and Sohrab Effati. Artificial Neural Networks Approach for Solving Stokes Problem. *Applied Mathematics,* 01(04):288-292, 2010.

Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proc Natl Acad Sci U S A,* 115(34):8505-8510, 2018.

Alexandr Katrutsa, Talgat Daulbaev, and Ivan Oseledets. Deep multigrid: learning prolongation and restriction matrices. *arXiv preprint arXiv:1711.03825*, 2017.

Siddhartha Mishra. A machine learning framework for data driven acceleration of computations of differential equations. *arXiv preprint arXiv:1807.09519*, 2018.

Justin Sirignano, and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics,* 375:1339-1364, 2018.

Mingui Sun, Xiaopu Yan, and R.J. Sclabassi. Solving partial differential equations in real-time using artificial neural network signal processing as an alternative to finite-element analysis. In *International Conference on Neural Networks and Signal Processing, 2003. Proceedings of the 2003,* pp. 381–384, December 2003.

Tao Shan, Wei Tang, Xunwang Dang, Maokun Li, Fan Yang, Shenheng Xu, and Ji Wu. Study on a poissons equation solver based on deep learning technique. *arXiv preprint arXiv:1712.05559*, 2017.

Shiyin Wei, Xiaowei Jin, and Hui Li. General solutions for nonlinear differential equations: a rule-based self-learning approach using deep reinforcement learning. *Computational Mechanics,* 64(5):1361-1374, 2019.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. *arXiv preprint arXiv: 1806.07366*, 2019.

Bo Chang, Lili Meng, Eldad Haber, Frederick Tung, and David Begert. Multi-level residual networks from dynamical systems view. *arXiv preprint arXiv: 1710.10348*, 2018.