GRADIENT-BASED META-SOLVING AND ITS APPLICATIONS TO ITERATIVE METHODS FOR SOLVING DIFFERENTIAL EQUATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

In science and engineering applications, it is often required to solve similar computational problems repeatedly. In such cases, we can utilize the data from previously solved problem instances to improve efficiency of finding subsequent solutions. This offers a unique opportunity to combine machine learning (in particular, meta-learning) and scientific computing. To date, a variety of such domain-specific methods have been proposed in the literature, but a generic approach for designing these methods remains under-explored. In this paper, we tackle this issue by formulating a general framework to describe these problems, and propose a gradient-based algorithm to solve them in a unified way. As an illustration of this approach, we study the adaptive generation of initial guesses for iterative solvers to speed up the solution of differential equations. We demonstrate the performance and versatility of our method through theoretical analysis and numerical experiments.

1 INTRODUCTION

It is common and important in science and engineering to solve similar problems repeatedly. For example, in material science, a tremendous amount of physical and numerical experiments are conducted to discover and characterize new materials (Schmidt et al. (2019)). For another example, in computational fluid dynamics, many methods involve solving a Poisson equation to compute the pressure field in every time step of the simulation (Ajuria Illarramendi et al. (2020)). In these situations, we can utilize the data of the previously solved problems to solve the next similar but unseen problems more efficiently, and machine learning is a natural and effective approach for this.

Thus, in recent years, many learning-based methods have been proposed for repeated solutions of computational problems such as partial differential equations (PDEs) (Tang et al. (2017); Özbay et al. (2021); Tompson et al. (2017); Hsieh et al. (2018); Huang et al. (2020)). For example, in Tang et al. (2017), Ajuria Illarramendi et al. (2020), and Özbay et al. (2021), convolutional neural networks are used to predict the solution of Poisson equations, and Ajuria Illarramendi et al. (2020) and Özbay et al. (2021) propose to use the predicted solution as an initial guess of traditional numerical methods. Hsieh et al. (2018) combines a neural network and an iterative solver to accelerate it with maintaining the convergence guarantee. In addition to the regular supervised learning approaches, there are several works where meta-learning approaches are taken to solve computational problems (Feliu-Fabà et al. (2020); Chen et al. (2020); Psaros et al. (2021); Guo et al. (2021)). Meta-learning, or learning to learn, leverages previous learning experiences to improve future learning performance (Hospedales et al. (2021)), which fits the motivation utilizing the data from previously solved equations for the next one. For example, Chen et al. (2020) use meta-learning to generate a smoother of the Multi-grid Network for parametrized PDEs. Psaros et al. (2021) propose a meta-learning technique for offline discovery of physics-informed neural network loss functions.

Although many methods have been proposed in this direction, they are often problem-specific. In other words, there lacks both a unified framework to explain them and a general design pattern to adapt them to new problem settings. On the other hand, in the machine learning literature, there exists a general methodology - gradient-based meta-learning (Finn et al. (2017)) - that covers a variety of meta-learning problem settings. In this paper, we generalize this approach to yield a framework,

which we call gradient-based meta-solving (GBMS), that encompasses both learning and computational problems. This offers a general means to understand and develop learning-based algorithms to speed up computation. As an illustration of our approach, we apply GBMS to accelerate the solution of differential equations with iterative methods through learning. We show the advantage of the proposed algorithm over the baseline classical and learning-based approaches through theoretical analysis and numerical experiments. Finally, we incorporate the algorithm into a practical application and demonstrate its versatility and performance.

2 GRADIENT-BASED META-SOLVING

In this section, we introduce our core idea of gradient-based meta-solving. First, we formulate a class of problems, which we call meta-solving, that includes both ordinary meta-learning and learning-based computational problems. We then propose a general gradient-based algorithm for solving them. Under our formulation, many methods proposed in related works can be regarded as special cases of the GBMS algorithm.

2.1 GENERAL FORMULATION OF META-SOLVING

Let us now introduce the general formulation of meta-solving. We first fix the required notations. A task τ is a tuple $\tau = (D, U, L)$ consisting of a dataset D, a solution space U, and a loss function L. A solution space U is a set of parametric candidate solutions, which is usually a subset of $\mathbb{R}^N$ for some $N \in \mathbb{N}$. A loss function L is a function from U to $\mathbb{R}_{\geq 0}$ that measures the quality of solution candidates. To solve a task τ means to find an approximate solution $\hat{u} \in U$ which minimizes $L(\cdot)$. Meta-solving considers the solution of not one, but a distribution of tasks by a learnable solver. Thus, we consider a task space $(\mathcal{T}, P)$ as a probability space that consists of a set of tasks $\mathcal{T}$ and a task distribution P, which is a probability measure defined on a suitable σ-algebra on $\mathcal{T}$. A solver $\mathcal{A}$ is a function from $\mathcal{T}$ to U, where $U = \bigcup_{\tau \in \mathcal{T}} U_\tau$. $\theta \in \Theta$ is the parameter of $\mathcal{A}$, and Θ is its parameter space. Here, θ may or may not be trainable, depending on the problem. Then, solving a task $\tau \in \mathcal{T}$ by an algorithm $\mathcal{A}$ with a parameter θ is denoted by $\mathcal{A}(\tau; \theta) = \hat{u}$. A meta-solver $\mathcal{B}$ is a function from $\mathcal{T}$ to $\mathcal{A}$, where $\omega \in \Omega$ is a parameter of $\mathcal{B}$ and Ω is its parameter space. A meta-solver $\mathcal{B}$ parametrized by $\omega \in \Omega$ is expected to generate an appropriate parameter $\theta \in \Theta$ for solving a task $\tau \in \mathcal{T}$ with a solver $\mathcal{A}$, which is denoted by $\mathcal{B}(\tau; \omega) = \theta$. Then, by using the notations above, our meta-solving problem is defined as follows:

Definition 1 (Meta-solving problem)**.** *For a given task space $(\mathcal{T}, P)$, solver $\mathcal{A}$, and meta-solver $\mathcal{B}$, find $\omega \in \Omega$ which minimizes $\mathbb{E}_{\tau \sim P}[L(\mathcal{A}(\tau; \mathcal{B}(\tau; \omega)))]$.*

We present some familiar examples, which can be regarded as special cases of the meta-solving problem. First, we can see that conventional meta-learning problems fall in this formulation.

Example 1 (Few-shot learning)**.** As an example of the meta-learning problem, we take the few-shot regression problem with MAML (Finn et al. (2017)). The components of the problem are the following. The task τ is to learn a regression model from data. The dataset is $D = \{(x_i, y_i)\}_{i=1}^K$, which satisfies $y = f(x)$ for an unknown function f. D is divided into the training set D^{train} and validation set D^{val}. The solution parameter space U is a weights space of a neural network (NN) that models f. Note that $U = U_\tau$ because the architecture of NN is shared across all tasks. The approximate solution $\hat{u} \in U$ is the trained weights of NN, which is obtained by training on D^{train}. The loss function $L : U \to \mathbb{R}_{\geq 0}$ is the mean squared error (MSE) on D^{val}. The task distribution $(\mathcal{T}, P)$ is determined by the distribution of the target function f and distribution of samples (x, y). The solver $\mathcal{A} : \mathcal{T} \to U$ is the single step gradient descent to minimize $L^{\text{train}}(u) = \frac{1}{|D_\tau^{\text{train}}|} \sum_{(x,y) \in D_\tau^{\text{train}}} \|\text{NN}(x; u) - y\|^2$. Its parameter $\theta \in \Theta$ is initial weights $u^{(0)}$ of NN, so $\Theta = U$. Thus, $\mathcal{A}(\tau; \theta) = u^{(0)} - \alpha \nabla_{u^{(0)}} L^{\text{train}}(u^{(0)}) = \hat{u}$, where α is a learning rate. The meta-solver $\mathcal{B} : \mathcal{T} \to \mathcal{A}$ is considered as a constant function that returns its parameter $\omega \in \Omega$ for any task $\tau \in \mathcal{T}$. The parameter $\omega \in \Omega$ is expected to be an appropriate initial weights for all $\tau \in \mathcal{T}$ to be fine-tuned easily. Thus, $\Omega = \Theta = U$, and $\mathcal{B}(\tau; \omega) = \omega = \theta = u^{(0)}$. Note that the output of the meta-solver, the initial weights $u^{(0)}$, does not depend on τ. Then, the few-shot learning problem is defined as meta-solving problem, which is

$$\min_\omega \mathbb{E}_{\tau \sim P}[L(\mathcal{A}(\tau; \mathcal{B}(\tau; \omega)))] = \min_\omega \mathbb{E}_{\tau \sim P} \sum_{(x,y) \in D_\tau^{\text{val}}} \left\| \text{NN}(x; \omega - \alpha \nabla_\omega L^{\text{train}}(\omega)) - y \right\|^2. \quad (1)$$

In addition to the conventional learning problems, we can regard other computational problems, such as solving a differential equation, as a task of the meta-solving problem. In constrast with MAML, the inner-loop learning is now replaced with a family of iterative solvers for differential equations. This necessitates the distinction of the meta-solver parameter space and the solution space U. Moreover, the meta-solver has to produce a task-specific parameter for the inner solver.

Example 2 (Solving differential equations)**.** Suppose that we need to repeatedly solve similar instances of a class of differential equations with a given numerical solver. The solver has a number of hyper-parameters, which sensitively affect accuracy and efficiency depending on the problem instance. Thus, finding a strategy to optimally select solver hyper-parameters given a problem instance can be viewed as a meta-solving problem. The components of the problem are the following. The task τ is to solve a differential equation. In this example, suppose that the target equation is the Poisson equation $u = f$ with Dirichlet boundary conditions $u = g$. The dataset D contains data of the differential equation, $D = \{f, g\}$. The solution parameter space U is $\mathbb{R}^{N_\tau}$. The loss function $L : U \to \mathbb{R}_0$ measures the accuracy of $\hat{u} \in U$. In this example, the ℓ^2-norm of the residuals obtained by substituting the approximate solution $\hat{u}$ into the equation can be used. The task distribution $(\mathcal{T}, P)$ is the joint distribution of f and g . The solver $\mathcal{A} : \mathcal{T} \to U$ is a numerical solver with a parameter $\theta \in \Theta$ for the differential equation. In this example, suppose that $\mathcal{A}$ is the Jacobi method (Saad (2003)) and θ is its initial guess. The meta-solver $\mathcal{S} : \mathcal{T} \to \Theta$ is a strategy characterized by $\omega \in \Omega$ to select a parameter, an initial guess, $\theta = \mathcal{S}(\tau) \in \Theta$ for each task $\tau \in \mathcal{T}$. Note that the output of the meta-solver depends on τ, which is different from the case of Example 1. Then, finding the strategy to select parameters of the numerical solver becomes meta-solving problem.

The above examples explain why we describe our problem as meta-solving instead of meta-learning. In Example 1, the task τ is learning from data, and the solver $\mathcal{A}$ is gradient descent. In Example 2, the task τ is solving a differential equation, and the solver $\mathcal{A}$ is an iterative differential equation solver. Regardless of the type of task and algorithm, in both cases, the solver $\mathcal{A}$ solves the task τ, and we do not distinguish whether $\mathcal{A}$ is a learning algorithm or other numerical solver. In other words, learning algorithms such as gradient descent are also a type of numerical solvers, and we regard learning as a special case of solving. It is also true for the outer learning algorithm to learn how to solve the task τ with the solver $\mathcal{A}$, so learning to solve is a special case of solving to solve. In this sense, we call it meta-solving.

2.2 Gradient-based meta-solving

In the previous section, we defined meta-solving problems as a generalization of meta-learning problems, but how can we solve them effectively? For meta-learning problems such as Example 1, general methodologies in the form of gradient-based meta-learning (Hospedales et al. (2021)), e.g. MAML (Finn et al. (2017)), have been proposed. In Example 1, MAML solves the minimization problem (1) by updating ω using the outer gradient descent algorithm $\omega \leftarrow \omega - \beta \nabla_\omega L(\hat{u})$. MAML is an algorithm to find good initial weights of the neural network for conventional meta-learning problems, but it can be generalized to the meta-solving problems. In meta-solving problems, tasks may not be learning problems, $\mathcal{A}$ may not be gradient descent, and its parameter θ may not be initial weights of a neural network. However, we can still employ the same update rule as MAML, as long as L , $\mathcal{A}$, and $\mathcal{S}$ are differentiable. Thus, for differentiable solvers $\mathcal{A}$ and $\mathcal{S}$, we propose gradient-based meta-solving algorithm (Algorithm 1) as a generalization of MAML. This is a form of data-driven algorithm design, and differs from previous works in this direction (e.g. Hutter et al. (2011), Mitzenmacher & Vassilvitskii (2021), and Balcan (2021)). These focus on discontinuous problems such as combinatorial optimization, while our framework focuses on differentiable problems.

2.3 Organizing related works

Owing to the general formulation presented above, we can organize several related works on learning-based methods for scientific computing and describe them in a unified way. We highlight the advantages of our method using the examples in this section. The detailed description of how the task, the solver and the meta-solver are defined in each case are found in Appendix A.

First, let us review Feliu-Fabà et al. (2020). This work proposes a neural network architecture inspired by the nonstantard wavelet form with meta-learning approach to solve the equations containing partial differential or integral operators. It can be regarded as a special case of GBMS, where

Algorithm 1: Gradient-based meta-solving algorithm

Require: (P, T): task space, , : differentiable solver, S: stopping criterion, α: learning rate
Result: ω
initialize ω;
while S *is not satisfied* **do**
  Sample task τ P;
  Generate parameter of using meta-solver: θ (ω) $(\tau; \omega)$;
  Solve task τ using solver: $\mathfrak{A}(\omega)$ $(\tau; \theta\ (\omega))$;
  Update meta-solver parameter: ω ω $\alpha\, \mathsf{r}_{\ !}\, L\ (\mathfrak{A}(\omega))$;
end

task τ is solving the equation, solver is the forward computation of the trained neural network, $\theta\ 2$ is its weights, and meta-solver is the constant function that returns the weights $\theta = \omega\ 2$. Note that θ does not depend on the task τ. We also note that other works where neural networks replace a whole or part of numerical methods (Tang et al. (2017); Özbay et al. (2021); Tompson et al. (2017); Hsieh et al. (2018)) can be organized in the same way. Thus, the meta-solving formulation includes many learning-based methods where meta-learning techniques are not explicitly employed.

We take Psaros et al. (2021) as another example. In this work, meta-learning is used to learn a loss function of the physics-informed neural network, shortly PINN (Raissi et al. (2019)), for solving PDEs. This also can be considered as a special case of GBMS, where τ is training the PINN, the solver is the gradient descent for training the PINN, $\theta\ 2$ is the weights of another neural network used as the loss function in the training, and the meta-solver is the constant function that returns the weights $\theta = \omega\ 2$. As in the previous example, θ does not depend on the task τ. We remark that this example and Example 1 are similar in the sense that is gradient descent and is the constant function returning neural network's weights in both examples, though the task τ is different. The unified framework sheds light on a similarity in various methods for various tasks, which enables us to apply a technique developed for one problem to another easily.

Lastly, let us consider Chen et al. (2020). In this work, meta-learning is used to generate a parameter of PDE-MgNet, a neural network representing the multigrid method, for solving parametrized PDEs. This also can be regarded as a special case of GBMS, where τ is solving a PDE, the solver is the PDE-MgNet, whose iterative function is implemented by a neural network ϕ with weights θ, and the meta-solver is another neural network with weights ω to generate θ depending on task τ. Note that the meta-solver is trained with single step of the solver but tested with multiple steps of . In other words, this work does not consider the solver itself but instead its iterative function ϕ in the training. We will show the importance of this difference in section 3.1.

The above examples show the generality of our meta-solving formulation that organizes a variety of methods in the systematic way regardless of their type of algorithm. This overall approach allows us to easily design learning-based methods for new problem settings, and we will demonstrate it in the next section.

3 GBMS FOR ITERATIVE METHODS

In this section, we demonstrate how GBMS can be used to yield effective meta-solvers for new problem settings. In particular, we consider the problem of accelerating iterative methods by adaptively choosing initial conditions. Iterative methods are powerful tools to solve computational problems. For example, the Jacobi method and SOR method are used to solve PDEs (Saad (2003)). In iterative methods, a function ϕ , which depends on the task τ, is iteratively applied to the current approximate solution to update it closer to the true solution until it reaches a criterion, such as a certain error tolerance or number of iterations. These methods require an initial guess as the starting point of the iterations, and the performance of the method depends on this choice. Thus, adaptively choosing initial guesses for solvers is an effective way to speed up the computation process.

Here, we apply GBMS to derive a meta-solver that produces effective initial guesses for iterative solvers. The meta-solving problem here can be viewed as a generalized version of Example 2 beyond Jacobi solvers. The task τ is any computational problem which can be solved by iterative

methods. For example, τ is solving a PDE as described in Example 2. The task distribution $(\mathcal{T}, P)$ is defined according to each problem. The solver $: \mathcal{T} \to \mathcal{U}$ is an iterative method with an initial guess $\theta = u^{(0)} \in$, iterative function ϕ , and the number of iterations k, so $(\tau; u^{(0)}) = \phi^k(u^{(0)}) = u^{(k)}$ and $= \mathcal{U}$. The meta-solver $: \mathcal{T} \to$ is a function parametrized by $\omega \in$, which takes $\tau \in \mathcal{T}$ as an input and generates an initial guess $\theta = u^{(0)} \in$ for the solver . We implement by a neural network, so $\omega \in$ is its weights. Then, is trained to minimize the expectation of $L\ (u^{(k)})$ by the gradient descent. Since both and are implemented in a deep learning framework, $\nabla_\omega L\ (u^{(k)})$ can be computed by back-propagation. The entire process of the algorithm is presented in Algorithm 2.

Hereafter, we consider solving a PDE as a task of the meta-solving problem, but the method is applicable to other tasks, such as root finding. Note that although Huang et al. (2020), Ajuria Illarramendi et al. (2020), and Özbay et al. (2021) propose to use initial guesses generated by neural networks, these initial guesses are independent of the solvers. On the other hand, our initial guesses are optimized for each solver. We will show its advantage in the following sections.

Algorithm 2: Gradient-based meta-solving algorithm for iterative solvers

Require: $(P, \mathcal{T})$: task space, : differentiable solver, S: stopping criterion, k: number of iterations of the iterative solver, α: learning rate
Result: ω
initialize ω;
while S *is not satisfied* **do**
  Sample task τ P;
  Form iterative function ϕ depending on τ;
  Generate initial guess using meta-solver: $u^{(0)}(\omega)$ $(\tau; \omega)$;
  for i 1 *to* k **do**
    Update approximate solution by iterative function: $u^{(i)}(\omega)$ $\phi\ (u^{(i\ 1)}(\omega))$;
  end
  Update meta-solver parameter: ω ω $\alpha \nabla_\omega L\ (u^{(k)}(\omega))$;
end

3.1 Toy problem: 1D Poisson equations

To study the property of the proposed algorithm, we consider solving 1D Poisson equations as a toy problem. First, we show a theorem that guarantees the improvement of the proposed meta-solving approach for the Jacobi method and linear neural networks. Then, we demonstrate that the theorem numerically holds for another iterative method and practical nonlinear neural network.

3.1.1 Theoretical analysis

Let us recall Example 2 and set the domain of interest $\Omega = (0, 1)$. Then, the target equation becomes the following 1D Poisson equation with Dirichlet boundary condition:

$$\begin{aligned} \frac{d^2}{dx^2} u(x) &= f(x), \quad x \in (0, 1) \\ u(0) &= a, \quad u(1) = b. \end{aligned} \tag{2}$$

To solve this equation numerically, it is discretized with finite difference scheme, and we rewrite it as the matrix equation $Au = f$, where the domain $[0, 1]$ is discretized into N points, so $A \in \mathbb{R}^{N \times N}$ and $u, f \in \mathbb{R}^N$. Suppose that we solve the equation $Au = f$ with the Jacobi method under randomly sampled f . Then, the meta-solving problem for solving 1D Poisson equations is defined by the following. The task τ is to solve $Au = f$. The dataset is $D = \{f, u\}$, where u is the solution of $Au = f$. The solution parameter space is $\mathcal{U} = \mathbb{R}^N$. The loss function is $L\ (\hat{u}) = \|u \hat{u}\|^2$. The task distribution $(\mathcal{T}, P)$ is determined by the distribution of f , denoted by P_f. It is assumed to be centered and normalized, i.e. the mean of f is 0 and the covariance is the identity matrix. The solver $_k : \mathcal{T} \to \mathcal{U}$ is the Jacobi method with k iterations starting at an initial guess

$u^{(0)} = \theta\ 2$. Its iterative function is $\phi\ (u) = Mu + \frac{1}{2}f$, where $M = I \quad \frac{1}{2}A$. To summarize, $_k(\tau; u^{(0)}) = \phi^k(u^{(0)}) =$ ft. Note that $_0$ is the identity map. The meta-solver $: T \quad !$ is a linear neural network with weights $\omega\ 2$. It can be represented as matrix multiplication for some matrix W, so $(\tau; \omega) = Wf = u^{(0)}$. Here, we assume $\mathsf{rank}(W) < N$ to avoid the trivial solution $W = A^{-1}$, i.e. the meta-solver does not have the capacity to produce an exact solution for each task. Then, the meta-solving problem becomes

$$\min_{! 2} \ \mathsf{E}_P [L \ (\ (\tau; \ (\tau; \omega))] = \min_W \mathsf{E}_{f_\tau} \mathsf{E}_{P_f} \left\| u \quad \phi^k(Wf\ ) \right\|^2 . \tag{3}$$

As for this meta-solving problem, the following theorem holds. The proof is in Appendix B.

Theorem 1 (Guarantee of improvement by meta-solving)**.** *For any k 0, (3) has the unique minimizer W_k. Furthermore, if $k_1 < k_2$, then for all k k_2,*

$$\mathsf{E}_{f_\tau} \mathsf{E}_{P_f} \left\| u \quad \phi^{k}(W_{k_1} f\ ) \right\|^2 \quad \mathsf{E}_{f_\tau} \mathsf{E}_{P_f} \left\| u \quad \phi^{k}(W_{k_2} f\ ) \right\|^2 , \tag{4}$$

where the equality holds if and only if $W_{k_1} = W_{k_2}$.

Theorem 1 guarantees improvement of meta-solving for the considered setting. For example, suppose $k_1 = 0$ (regular supervised learning, i.e. directly predicting the solution without considering the solver) and $k_2 = 5$. Then, the theorem implies that the meta-solver trained with 5 Jacobi iterations is expected to give an better initial guess than the meta-solver obtained by regular supervised learning for any number of Jacobi iterations larger than 5. More generally, Theorem 1 shows that meta-learning with a higher number of Jacobi iterations in the inner solver improves the performance, provided one carries out at least a greater number of Jacobi iterations during inference. This shows the advantage of the meta-solving approach over regular supervised learning.

3.1.2 Numerical examples for more general cases

The key insight from the previous theoretical analysis is that meta-solving leverages the properties of the solver and adapts the selection of initial conditions to it. Here, we show using numerical examples that this is the case for more complex scenarios, involving different task distributions, different iterative solvers, and a practical nonlinear neural network.

The meta-solving problem in this section is defined by the following. The task τ is the same as the previous section 3.1.1. Let the number of discretization points N be 512. The task distribution (T, P) is determined by the distribution of u. We consider two distributions P_s and P_h, where the solution u consists of sine functions and hyperbolic tangent functions respectively. Their details are listed in Appendix C.1. For each distribution, we prepare 30,000 tasks for training, 10,000 for validation, and 10,000 for test. To solve the tasks, we use the Jacobi method and the Red-Black ordering SOR method (Saad (2003)) with k iterations starting at an initial guess $u^{(0)} = \theta\ 2$, denoted by $_{\mathrm{Jac};k}$ and $_{\mathrm{SOR};k}$ respectively. Note that they are implemented by convolutional layers. We consider two meta-solvers. One is a heuristic initial guess generator $_{\mathsf{BL}}$ that takes τ as an input and gives the heuristic initial guess, which is the linear interpolation of the boundary condition. $_{\mathsf{BL}}$ does not have a parameter and is used as a baseline. The other is a variant of 1D U-Net (Ronneberger et al. (2015)) $_{\mathsf{NN}}$ with weights $\omega\ 2$, which takes f and the heuristic initial guess $_{\mathsf{BL}}(\tau)$ as inputs and generates an initial guess $\theta\ 2$ for the solver .

The meta-solver $_{\mathsf{NN}}$ is trained with solvers $_{\mathrm{Jac};k}$, $_{\mathrm{SOR};k}$ with $k = 0, 4, 16, 64$ by using the Algorithm 2. For each setting, $_{\mathsf{NN}}$ is trained six times with different random seeds. The details of the architecture and hyper-parameters of $_{\mathsf{NN}}$ are found in Appendix D.1. The trained meta-solver $_{\mathsf{NN}}$ and baseline $_{\mathsf{BL}}$ are tested with solvers $_{\mathrm{Jac};k}$, $_{\mathrm{SOR};k}$ with $k = 0, 4, 16, 64$. The performances of the meta-solvers are measured by the MSE on the test set and presented in Table 1. Figure 1 shows the comparison of the initial guesses $u^{(0)}$ obtained by $_{\mathsf{NN}}$ with $_{\mathrm{Jac};0}$ and $_{\mathrm{Jac};64}$. Figure 2 is the convergence plot of $_{\mathsf{NN}}$ trained with $_{\mathrm{SOR};k}$ on P_s.

The results show that the meta-solver $_{\mathsf{NN}}$ is optimized for each corresponding solver . In Table 1, the best performance for a given test solver is achieved at the diagonal where the training and test solvers match. We can explicitly investigate this adaptive phenomena by visualizing the meta-solver outputs for a representative problem instance. Figure 1 shows that the meta-solver trained with more iterations tends to ignore high frequencies and focus more on low frequencies. This is because the

Jacobi method converges fast in high frequencies but slow in low frequencies (Saad (2003)), and the trained meta-solver compensates the weakness of the solver. These results show that the meta-solver actively adapts to the nature of the inner-loop solver.

Furthermore, we can observe that Theorem 1 holds numerically for the results. Figure 2 illustrates that a meta-solver trained with more iterations is always better than those trained with fewer iterations, provided the number of iterations during testing is large. In addition, at 10,000 iterations, the error of NN trained with SOR;64 (MSE of 0.0054 and relative error of 0.013) remains approximately 8% better than NN trained with SOR;0, which shows that the GBMS keeps its advantage for a large number of iterations. These results support that Theorem 1 holds for various practical settings with significant improvement.

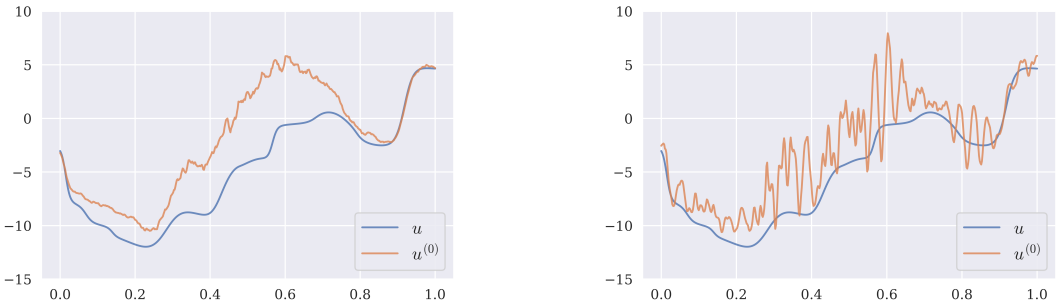
Table 1: Average MSE of GBMS for solving Poisson equations. Boldface indicates best performance for each column. Standard deviations are presented in Table 3 in Appendix E.

(a) MSE on P_s

	Trained with	Tested with Jac,0	Jac,4	Jac,16	Jac,64	SOR,4	SOR,16	SOR,64
NN	Jac,0 = SOR,0	**0.222**	0.220	0.219	0.214	0.218	0.212	0.193
	Jac,4	0.277	**0.212**	0.210	0.205	0.210	0.203	0.186
	Jac,16	1.169	0.231	**0.209**	0.204	0.221	0.203	0.185
	Jac,64	17.684	3.066	0.279	**0.198**	0.299	0.198	0.179
	SOR,4	1.140	0.982	0.753	0.496	**0.201**	0.195	0.179
	SOR,16	25.786	6.145	1.134	0.554	0.392	**0.185**	0.170
	SOR,64	22.451	8.306	2.459	0.744	2.536	0.224	**0.169**
BL	-	17.033	12.184	9.482	7.926	9.083	7.611	6.606

(b) MSE on P_h

	Trained with	Tested with Jac,0	Jac,4	Jac,16	Jac,64	SOR,4	SOR,16	SOR,64
NN	Jac,0 = SOR,0	**1.049**	1.041	1.030	1.005	1.028	0.994	0.922
	Jac,4	1.059	**1.023**	**1.006**	0.976	1.002	0.964	0.888
	Jac,16	1.202	1.049	**1.006**	0.971	1.000	0.958	0.885
	Jac,64	2.017	1.572	1.142	**0.940**	1.058	0.920	0.839
	SOR,4	1.632	1.479	1.383	1.333	**0.984**	0.938	0.862
	SOR,16	3.305	2.597	1.764	1.285	1.305	**0.928**	0.845
	SOR,64	7.683	6.009	3.580	1.417	3.029	1.116	**0.756**
BL	-	10.761	10.744	10.695	10.539	10.681	10.463	9.908



(a) $u^{(0)}$ generated by NN trained with Jac,0 (b) $u^{(0)}$ generated by NN trained with Jac,64

Figure 1: Comparison of initial guesses $u^{(0)}$ generated by NN

3.2 Application: Incompressible Flow Simulations

In this section, we introduce an application of GBMS for iterative solvers. GBMS is developed for repeatedly solving task τ sampled from a given task distribution P. One of the typical situations where GBMS is effective is that similar differential equations are repeatedly solved as a step of solving a time-dependent problem. For example, in many methods of incompressible flow simula-

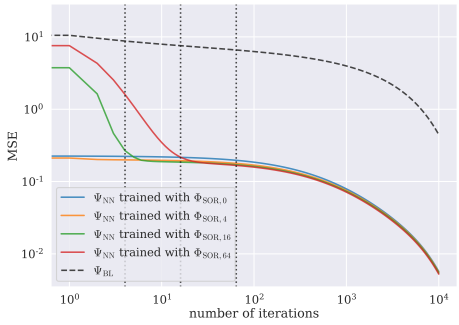


Figure 2: Convergence plot of trained with $_{\mathrm{SOR};k}$ on P_{s}. Vertical dotted lines are $k = 4, 16, 64$.

tions, Poisson equations are solved to compute the pressure field at every time step, and this process occupies a large part of the computation time for fluid simulations (Ajuria Illarramendi et al. (2020)).

Problem definition. Let us now describe the incompressible flow simulation setup. We consider 2D channel flow with a fluctuating inflow. The velocity field v and pressure field p follow the Navier-Stokes equations in the form:

$$\frac{\partial v}{\partial t} + (v \quad r)v = \quad \frac{1}{\rho} r\, p + \nu \quad v \tag{5}$$

$$p = \quad \rho\, r \quad ((v \quad r)v) \tag{6}$$

where ρ is the density, and ν is the viscosity. The equation (6) is called the pressure Poisson equation. Let $\rho = 1$, $\nu = 0.01$. Let the domain of interest D be $(0, 1)$ $(0, 1)$ and the time t 2 $[0, 1]$. Let $y = 0$ and $y = 1$ be the non-slip wall and fluctuating inflow v_{in} defined on $x = 0$. By using the finite difference scheme, we discretize the equations into 128 by 128 spacial grids and 1,000 time steps. Then, v and p are computed alternately by the discretized equations. Details of the numerical scheme for solving Navier-Stokes equations follow Barba & Forsyth (2018). GBMS for iterative method is used for solving the pressure Poisson equation (6).

Let us define the meta-solving problem for the above setting. The task τ is to solve a pressure Poisson equation. The dataset D is $\{f\ f\ , p\ , f\ _1, p\ _1, f\ _2, p\ _2\}$, where f is the right hand side and p is the solution of the pressure Poisson equation. $f\ _i, p\ _i$ are those at ith previous time step. Although p is determined by f theoretically, additional features of previous timesteps can provide useful information to determine a good guess for p . The solution parameter space U is $\mathbb{R}^{128\ \ 128}$. The loss function is the relative ℓ^2-error. The task distribution (T, P) is determined by the distribution of v_{in} and the computation process of the simulation. Let P_{in} be the distribution of v_{in}, whose detail is found in Appendix C.2. We prepare 20 inflows for training, 10 for validation, and 10 for test, which are denoted by $V_{\mathrm{in}}^{\mathrm{train}}$, $V_{\mathrm{in}}^{\mathrm{val}}$ and $V_{\mathrm{in}}^{\mathrm{test}}$ respectively. Then, we generate datasets under each inflow by solving the Navier-Stokes equations with the SOR method without GBMS, where the relative error tolerance is $10^{\ 9}$. The solver $_{\mathrm{SOR};k}$ is the SOR method with k iterations starting at an initial guess θ 2 . We consider two meta-solvers. One is a baseline $_{\mathrm{BL}}$ that takes τ as an input and gives the heuristic initial guess, which is the solution of the previous step Poisson equation $p\ _1$. This choice is based on the domain knowledge of fluid simulations and standard in the literature (Ferziger & Perić (2002)). The meta-solver $_{\mathrm{NN}}$ is a variant of 2D U-Net with weights ω 2 , which takes D $n\, f p\ g$ as an input and generates initial guess θ 2 for the solver $_{\mathrm{SOR};k}$. Note that $_{\mathrm{NN}}$ trained with $_{\mathrm{SOR};0}$ (i.e. solver-independent initial guess) is similar to the method in Ajuria Illarramendi et al. (2020) and is considered as a data-driven baseline in this paper. We also note that the choice of $_{\mathrm{NN}}$ is arbitrary, and any other problem-specific neural network architectures can be used as $_{\mathrm{NN}}$.

Training. We train the meta-solver $_{\mathrm{NN}}$ with the SOR solver $_{\mathrm{SOR};k}$ with $k = 0, 4, 16, 64$ by using the Algorithm 2. The details of the architecture and hyper-parameters of $_{\mathrm{NN}}$ are found in Appendix D.2. In addition, we use data augmentation during training to prevent overfitting to the high-accuracy training data. During the training, we augment data by proceeding 10 time steps with $_{\mathrm{SOR};k}$ and $_{\mathrm{NN}}$ being trained. Specifically, after computing the loss for p , we proceed one time step using $_{\mathrm{SOR};k}$ and $_{\mathrm{NN}}$, and compute the loss for next $p\ _{+1}$ using the computed previous step information.

Then, it is repeated 10 times for each mini batch. Without the data augmentation, the inputs of NN are from the prepared dataset and are always accurate during training. However, during inference, the inputs of NN are the outputs of the simulation at previous time steps, which can be of a slightly different distribution than encountered during training (i.e. distribution shift) due to numerical errors and generalization gaps. This then causes further accuracy problems for the simulation at the next step, and this issue compounds itself in time. This eventually degrades the performance if the shift in distribution is not dealt with. The data augmentation is used to prevent it. We remark that this training process does not require the whole of the simulation to be differentiable. This is practically important because we can employ GBMS with established solvers by only replacing the Poisson solver and can utilize the other parts without any modification.

Evaluation. We incorporate the trained meta-solvers into simulations and evaluate them by the relative ℓ^2-error of the velocity field on test inflows under a fixed number of the SOR iterations. More specifically, the performance metric is $\frac{1}{|V_{\text{in}}^{\text{test}}|}\sum_{v_{\text{in}}\in V_{\text{in}}^{\text{test}}}\frac{1}{900}\sum_{i=101}^{1000}\frac{\|v_i-\hat{v}_i\|}{\|\hat{v}_i\|}$, where v_i is the ground truth of ith step velocity under the inflow v_{in}, which is obtained by using BL and SOR;k with a large enough k to achieve the relative error tolerance of 10^{-9}, and $\hat{v}_i$ is the ith step velocity obtained by using the trained NN and SOR;k with a fixed k. To stabilize the simulation, the first 100 steps are excluded, and the simulation with the trained NN starts with 101 steps.

The results presented in Table 2 demonstrate the advantage of GBMS. Although all NN diverge for SOR;0 because of error accumulation, for the other solvers, the best performance with significant improvement is achieved at the diagonal where the training and test solvers match. For example, for the test solver SOR;4, the accuracy of NN trained with SOR;4 is approximately 50 times better than regular supervised learning (NN trained with SOR;0) and 130 times better than the classical baseline BL. Furthermore, GBMS keeps its advantage for a larger number of iterations than the number which it is trained with. For example, for SOR;64, the meta-solver trained with SOR;64 has the best performance, followed by one trained with SOR;16, SOR;4, and SOR;0 in that order. As for computation time, NN trained with SOR;4 and tested with SOR;4 (GBMS approach) takes 11 sec to simulate the flow for one second, while BL tested with SOR;128 takes 105 sec to achieve similar accuracy to the GBMS approach. Also, NN trained with SOR;0 and tested with SOR;64 takes 73 sec to achieve the similar accuracy. Since it takes approximately 7 hours to train NN with a GeForce RTX 3090, our approach is effective if we simulate the flow for more than 268 seconds, which can be easily satisfied in practical settings. To summarize, GBMS performs best for a fixed number of iterations and can achieve high accuracy within a smaller number of iterations and shorter computation time than the classical baseline and regular supervised learning.

Table 2: Relative ℓ^2-error of GBMS on incompressible flow simulations.

	Trained with	Tested with SOR,0	SOR,4	SOR,16	SOR,64
NN	SOR,0	NaN	0.054861	0.008543	0.001284
	SOR,4	NaN	**0.001171**	0.001625	0.000864
	SOR,16	NaN	0.017513	**0.000829**	0.000723
	SOR,64	NaN	NaN	0.005567	**0.000312**
BL	-	1.988177	0.152626	0.014112	0.002941

4 CONCLUSION

In this paper, we proposed a formulation of meta-solving and a general gradient-based algorithm (GBMS) to solve this class of problems. In the proposed framework, many related works that used neural networks for solving differential equations were organized in a unified way and regarded as variants of GBMS. Thus, the GBMS approach offers a general design pattern to develop solution algorithms that blends machine learning and scientific computing. As a concrete illustration, we applied GBMS to iterative methods for solving differential equations and showed its advantage over both classical numerical methods and regular supervised learning. In particular, the proposed method was tested in the incompressible flow simulation and the accuracy was improved by approximately 50 times compared to regular supervised learning and 130 times compared to a classical baseline for a fixed number of iterative solver iterations. We will study applications of GBMS to other types of problems such as nonlinear equations in future work.

5 REPRODUCIBILITY STATEMENT

To ensure the reproducibility, we provide the detailed proof of Theorem 1 in Appendix B. As for the experiments, we provide the details of dataset generation in Appendix C and the details of network architecture and training hyper-parameters in Appendix D. In addition, we conducted experiments with different random seeds to obtain robust results. Finally, we will make our source code public at a later date. Source code with limited documentation is available upon request.

REFERENCES

Ekhi Ajuria Illarramendi, Antonio Alguacil, Michaël Bauerheim, Antony Misdariis, Benedicte Cuenot, and Emmanuel Benazera. Towards an hybrid computational strategy based on Deep Learning for incompressible flows. In *AIAA AVIATION 2020 FORUM*, AIAA AVIATION Forum. American Institute of Aeronautics and Astronautics, June 2020.

Maria-Florina Balcan. Data-Driven Algorithm Design. In *Beyond the Worst-Case Analysis of Algorithms*, pp. 626–645. Cambridge University Press, January 2021.

Lorena Barba and Gilbert Forsyth. CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education*, 1(9):21, November 2018.

Yuyan Chen, Bin Dong, and Jinchao Xu. Meta-MgNet: Meta Multigrid Networks for Solving Parameterized Partial Differential Equations. *arXiv preprint arXiv:2010.14088*, October 2020.

Jordi Feliu-Fabà, Yuwei Fan, and Lexing Ying. Meta-learning pseudo-differential operators with deep neural networks. *Journal of computational physics*, 408:109309, May 2020.

Joel H Ferziger and Milovan Perić. *Computational Methods for Fluid Dynamics*. Springer, Berlin, Heidelberg, 2002.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 1126–1135. JMLR.org, August 2017.

Yue Guo, Felix Dietrich, Tom Bertalan, Danimir T Doncevic, Manuel Dahmen, Ioannis G Kevrekidis, and Qianxiao Li. Personalized Algorithm Generation: A Case Study in Meta-Learning ODE Integrators. *arXiv preprint arXiv:2105.01303*, May 2021.

Timothy M Hospedales, Antreas Antoniou, Paul Micaelli, and Amos J Storkey. Meta-Learning in Neural Networks: A Survey. *IEEE transactions on pattern analysis and machine intelligence*, PP, May 2021.

Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning Neural PDE Solvers with Convergence Guarantees. In *International Conference on Learning Representations*, September 2018.

Jianguo Huang, Haoqin Wang, and Haizhao Yang. Int-Deep: A deep learning initialized iterative method for nonlinear problems. *Journal of computational physics*, 419:109675, October 2020.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, pp. 507–523. Springer Berlin Heidelberg, 2011.

Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with Predictions. In *Beyond the Worst-Case Analysis of Algorithms*, pp. 646–662. Cambridge University Press, January 2021.

Ali Girayhan Özbay, Arash Hamzehloo, Sylvain Laizet, Panagiotis Tzirakis, Georgios Rizos, and Björn Schuller. Poisson CNN: Convolutional neural networks for the solution of the Poisson equation on a Cartesian mesh. *Data-Centric Engineering*, 2, 2021.

Apostolos F Psaros, Kenji Kawaguchi, and George Em Karniadakis. Meta-learning PINN loss functions. *arXiv preprint arXiv:2107.05544*, July 2021.

M Raissi, P Perdikaris, and G E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of computational physics*, 378:686–707, February 2019.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pp. 234–241. Springer International Publishing, 2015.

Yousef Saad. *Iterative Methods for Sparse Linear Systems: Second Edition*. Other Titles in Applied Mathematics. SIAM, April 2003.

Jonathan Schmidt, Mário R G Marques, Silvana Botti, and Miguel A L Marques. Recent advances and applications of machine learning in solid-state materials science. *npj Computational Materials*, 5(1):1–36, August 2019.

Wei Tang, Tao Shan, Xunwang Dang, Maokun Li, Fan Yang, Shenheng Xu, and Ji Wu. Study on a Poisson's equation solver based on deep learning technique. In *2017 IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*, pp. 1–3, December 2017.

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 3424–3433. JMLR.org, August 2017.

A DETAILS OF EXAMPLES

Example 3 (Feliu-Fabà et al. (2020))**.** Feliu-Fabà et al. (2020) propose the neural network architecture with meta-learning approach that solves the equations in the form $L\ u(x) = f(x)$ with appropriate boundary conditions, where L is a partial differential or integral operator parametrized by a parameter function $\eta(x)$. This work can be described as follows:

- Task τ: The task τ is to solve a $L\ u(x) = f(x)$ for $\eta = \eta\ :$
  - Dataset D : The dataset D is $D = f\eta\ , f\ , u\ g$, where $\eta\ , f\ , u\ 2\ \mathsf{R}^N$ are the [], right hand side, and solution respectively. The reference solution u is obtained by [].
  - Solution parameter space U : The solution parameter space U is a subset of R^N for $N\ 2\ \mathsf{N}$.
  - Loss function L : The loss function $L\ : U\ !\ \mathsf{R}_{\ 0}$ is the mean squared error with the reference solution, i.e. $L\ (\hat{u}) = ku\ \ \hat{u}k^2$.
- Task space (T, P): The task distribution (T, P) is determined by the distribution of η and f .
- Solver : The solver $: T\ \ !\ U$ is implemented by a neural network imitating the wavelet transform, which is composed by three modules with weights $\theta = (\theta_1, \theta_2, \theta_3)$. In detail, the three modules, $\phi_1(\ ;\theta_1)$, $\phi_2(\ ;\theta_2)$, and $\phi_3(\ ;\theta_3)$, represent forward wavelet transform, mapping η to coefficients matrix of the wavelet transform, and inverse wavelet transform respectively. Then, is represented by $(\tau;\theta) = \phi_3((\phi_2(\eta\ ;\theta_2)\phi_1(f\ ;\theta_1));\theta_3) = \hat{u}$.
- Meta-solver : The meta-solver $: T\ \ !$ is the constant function that returns its parameter ω, so $(\tau;\omega) = \omega = \theta$ and $=$. Note that θ does not depend on τ in this example.

Then, the weights $\omega = \theta$ is optimized by a gradient descent algorithm as described in Algorithm 1.

Example 4 (Chen et al. (2020))**.** The target equation in Chen et al. (2020) is a linear systems of equations $A\ u = f$ obtained by discretizing parameterized steady-state PDEs, where $u, f\ 2\ \mathsf{R}^N$ and $A\ \ 2\ \mathsf{R}^{N\ \ N}$ is determined by η, a parameter of the original equation. This work can be described as follows:

- Task τ: The task τ is to solve a linear system $A\ u = f$ for $\eta_{=}\eta\ :$
  - Dataset D : The dataset D is $f\eta\ , f\ g$.
  - Solution parameter space U : The solution parameter space U is R^N.
  - Loss function L : The loss function $L\ : U\ !\ \mathsf{R}_{\ 0}$ is an unsupervised loss based on the residual of the equation, $L\ (\hat{u}) = kf\ \ A_{\ \tau}\hat{u}k^2/kf\ k^2$.
- Task space (T, P): The task distribution (T, P) is determined by the distribution of η and f .
- Solver : The solver $: T\ \ !\ U$ is iterations of a function $\phi\ (\ ;\theta) : U\ !\ U$ that represents an update step of the multigrid method. ϕ is implemented using a convolutional neural network and its parameter θ is the weights corresponding to the smoother of the multigrid method. Note that weights of ϕ other than θ are naturally determined by η and the discretization scheme. In addition, ϕ takes f as part of its input at every step, but we write these dependencies as ϕ for simplicity. To summarize, $(\tau;\theta) = \phi^k(u^{(0)};\theta) = \hat{u}$, where k is the number of iterations of the multigrid method and $u^{(0)}$ is initial guess, which is $\mathbf{0}$ in the paper.
- Meta-solver : The meta-solver $: T\ \ !$ is implemented by a neural network with weights ω, which takes $A_{\ \tau}$ as its input and returns weights θ that is used for the smoother inspired by the subspace correction method.

Then, ω is optimized by a gradient decent algorithm with the number of multigrid iteration $k = 1$ as described in Algorithm 1.

Example 5 (Psaros et al. (2021))**.** In Psaros et al. (2021), meta-learning is used for learning a loss function of the physics-informed neural network, shortly PINN (Raissi et al. (2019)). The target equations are the following:

$$F\ [u](t,x) = 0, (t,x)\ 2\ [0,T] \quad D \tag{a}$$
$$B\ [u](t,x) = 0, (t,x)\ 2\ [0,T] \quad \partial D \tag{b}$$
$$u(0,x) = u_{0;}\ (x), x\ 2\ D, \tag{c}$$

where $D \quad \mathsf{R}^M$ is a bounded domain, $u : [0,T] \quad D \ ! \ \mathsf{R}^N$ is the solution, F is a nonlinear operator containing differential operators, B is a operator representing the boundary condition, $u_{0;} : D \ ! \ \mathsf{R}^N$ represents the initial condition, and λ is a parameter of the equations.

- Task τ: The task τ is to solve a differential equation by PINN:
  - Dataset D : The dataset D is the set of points $(t,x)\ 2\ [0,T] \quad D$ and the values of u at the points if applicable. In detail, $D = D_{f;} \ [\ D_{b;} \ [\ D_{u_0;} \ [\ D_{u;}$, where $D_{f;}$, $D_{b;}$, and $D_{u_0;}$ are sets of points corresponding to the equation (a), (b), and (c) respectively. $D_{u;}$ is the set of points (t,x) and observed values $u(t,x)$ at the points. In addition, each dataset $D_{\ ;}$ is divided into training set $D_{\ ;}^{\text{train}}$ and validation set $D_{\ ;}^{\text{val}}$.
  - Solution parameter space U : The solution parameter space U is the weights space of PINN.
  - Loss function L : The loss function $L : U \ ! \ \mathsf{R}_{\ 0}$ is based on the evaluations at the points in D^{val}. In detail,

$$L\ (\hat{u}) = L^{\text{val}}(\hat{u}) = L_{f;}^{\text{val}}\ (\hat{u}) + L_{b;}^{\text{val}}\ (\hat{u}) + L_{u_0;}^{\text{val}}\ (\hat{u}),$$

    where

$$L_{f;}^{\text{val}} = \frac{w_f}{jD_{f;}\ j} \sum_{(t;x)\ 2\ D_{f,\tau}} \ell\ (F\ [\hat{u}](t,x), \mathbf{0})$$
$$L_{b;}^{\text{val}} = \frac{w_b}{jD_{b;}\ j} \sum_{(t;x)\ 2\ D_{b,\tau}} \ell\ (B\ [\hat{u}](t,x), \mathbf{0})$$
$$L_{u_0;}^{\text{val}} = \frac{w_{u_0}}{jD_{u_0;}\ j} \sum_{(t;x)\ 2\ D_{u_0,\tau}} \ell\ (\hat{u}(0,x), u_{0;}\ (x))\ ,$$

    and $\ell : \mathsf{R}^N \quad \mathsf{R}^N \ ! \ \mathsf{R}_{\ 0}$ is a function. In the paper, the mean squared error is used as ℓ.

- Task space $(T\ , P)$: The task distribution $(T\ , P)$ is determined by the distribution of λ.

- Solver : The solver $: T \quad ! \ U$ is the gradient decent for training the PINN. The parameter $\theta\ 2$ controls the objective of the gradient decent, $L^{\text{train}}(\hat{u};\theta) = L_{f;}^{\text{train}}(\hat{u};\theta) + L_{b;}^{\text{train}}(\hat{u};\theta) + L_{u_0;}^{\text{train}}\ (\hat{u};\theta) + L_{u;}^{\text{train}}(\hat{u};\theta)$, where the difference from L^{val} is that parametrized loss ℓ is used in L^{train} instead of the MSE in L^{val}. Note that the loss weights w_f, w_b, w_{u_0}, w_u in L^{train} are also considered as part of the parameter θ. In the paper, two designs of ℓ are studied. One is using a neural network, and the other is using a learned adaptive loss function. In the former design, θ is the weights of the neural network, and in the latter design, θ is the parameter in the adaptive loss function.

- Meta-solver : The meta-solver $: T \quad !$ is the constant function that returns its parameter ω, so $(\tau;\omega) = \omega = \theta$ and $=$. Note that θ does not depend on τ in this example.

Then, the parameter $\omega = \theta$ is optimized by a gradient decent algorithm as described in Algorithm 1.

B PROOF OF THEOREM 1

Proof of Theorem 1. Equation (3) can be represented as follows:

$$\min_{W} \mathbb{E}_{f_\tau} \mathbb{E}_{P_f} \left\| u \quad \phi^k(Wf) \right\|^2 = \min_{W} \mathbb{E}_{f_\tau} \mathbb{E}_{P_f} \left\| M^k(u \quad Wf) \right\|^2 \tag{7}$$

$$= \min_{W} \mathbb{E}_{f_\tau} \mathbb{E}_{P_f} \left\| M^k(A^{-1} f \quad Wf) \right\|^2. \tag{8}$$

Since the mean of f is 0 and the covariance is the identity matrix, W is the minimizer of (8) if and only if W is the minimizer of the following:

$$\min_{W} \left\| M^k(A^{-1} \quad W) \right\|_F^2. \tag{9}$$

Let λ_i and v_i $(i = 1, 2, \ldots, N)$ be eigenvalues and eigenvectors of M. Note that $\lambda_i = \cos \frac{i}{N+1}$ and $1 > \lambda_1 > \lambda_2 > \quad > \lambda_N > \quad 1$. Since $A = 2I \quad 2M$, eigenvalues and eigenvectors of A are $2 \quad 2\lambda_i$ and v_i. By the eigenvalue decomposition, M and A can be written as

$$M = V \quad V^T \tag{10}$$

$$A = V(2I \quad 2 \quad)V^T, \tag{11}$$

where $= \text{diag}(\lambda_i)$ and $V = (v_1, v_2, \ldots, v_N)$. By the decompositions,

$$\left\| M^k(A^{-1} \quad W) \right\|_F^2 = \left\| V^{\ k} V^T (V(2I \quad 2 \quad)^{-1} V^T \quad W) \right\|_F^2 \tag{12}$$

$$= \left\| V^{\ k} (2I \quad 2 \quad)^{-1} V^T \quad V^{\ k} V^T W) \right\|_F^2 \tag{13}$$

Thus, the minimizer W_k is

$$W_k = \sum_{j=1}^{r} (2 \quad 2\lambda_{i(j;k)})^{-1} v_{i(j;k)} v_{i(j;k)}^T, \tag{14}$$

where $r = \text{rank}(W_k) < N$, and $i(j,k)$ is the index i where $\left| \frac{\lambda_i^k}{2 \quad 2\lambda_i} \right|$ takes the top jth value in $i \in \{1, 2, \ldots, N\}$. Then, the following lemma holds.

Lemma 2. *If $k_1 < k_2$, then for all $k \quad k_2$*

$$\left\| M^k(A^{-1} \quad W_{k_1}) \right\|_F^2 \quad \left\| M^k(A^{-1} \quad W_{k_2}) \right\|_F^2, \tag{15}$$

where the equality holds if and only if $W_{k_1} = W_{k_2}$.

Since (8) and (9) are equivalent, if Lemma 2 holds, then Theorem 1 holds.

Proof of Lemma 2. Let

$$I_k := \{1, 2, \ldots, N\} \cap \{i(j,k) : j = 1, 2, \ldots, r\}. \tag{16}$$

I_k is the set of $N \quad r$ indices corresponding to the least $N \quad r$ values of $\left| \frac{\lambda_i^k}{2 \quad 2\lambda_i} \right|$. We have

$$\left\| M^k(A^{-1} \quad W_{k_1}) \right\|_F^2 \quad \left\| M^k(A^{-1} \quad W_{k_2}) \right\|_F^2 \tag{17}$$

$$= \sum_{i \in I_{k_1}} \left(\frac{\lambda_i^k}{2 \quad 2\lambda_i} \right)^2 \quad \sum_{i \in I_{k_2}} \left(\frac{\lambda_i^k}{2 \quad 2\lambda_i} \right)^2 \tag{18}$$

$$= \sum_{i \in I_{k_1} \cap I_{k_2}} \left(\frac{\lambda_i^k}{2 \quad 2\lambda_i} \right)^2 \quad \sum_{i \in I_{k_2} \cap I_{k_1}} \left(\frac{\lambda_i^k}{2 \quad 2\lambda_i} \right)^2. \tag{19}$$

If $W_{k_1} = W_{k_2}$, then (19) equals 0. Assume $W_{k_1} \neq W_{k_2}$, so $I_{k_1} \neq I_{k_2}$. Note that $I_k = \{\min I_k, \min I_k + 1, \ldots, \min I_k + (N - r) - 1 = \max I_k\}$, and for any $i \in I_k$, we have $|\lambda_i| \geq |\lambda_{\max I_k}|$. In addition, for $k_1 < k_2$, we have $\max I_{k_1} \geq \max I_{k_2}$. Let

$$i_{k_1} := \arg\min_{i \in I_{k_1} \cap I_{k_2}} \left| \frac{\lambda_i^k}{2 - 2\lambda_i} \right| \tag{20}$$

$$i_{k_2} := \arg\max_{i \in I_{k_2} \cap I_{k_1}} \left| \frac{\lambda_i^k}{2 - 2\lambda_i} \right| \tag{21}$$

$$C := |I_{k_1} \cap I_{k_2}| = |I_{k_2} \cap I_{k_1}|. \tag{22}$$

Then, we have

$$(19) \quad C \left(\frac{\lambda_{i_{k_1}}^k}{2 - 2\lambda_{i_{k_1}}} \right)^2 - C \left(\frac{\lambda_{i_{k_2}}^k}{2 - 2\lambda_{i_{k_2}}} \right)^2 \tag{23}$$

$$= C \left(\left(\frac{\lambda_{i_{k_1}}^{k_2 + (k - k_2)}}{2 - 2\lambda_{i_{k_1}}} \right)^2 - \left(\frac{\lambda_{i_{k_2}}^{k_2 + (k - k_2)}}{2 - 2\lambda_{i_{k_2}}} \right)^2 \right) \tag{24}$$

$$> 0. \tag{25}$$

The last inequality (25) is because for any $i \in I_{k_2}$,

$$\left| \frac{\lambda_{i_{k_1}}^{k_2}}{2 - 2\lambda_{i_{k_1}}} \right| > \left| \frac{\lambda_i^{k_2}}{2 - 2\lambda_i} \right| \tag{26}$$

and

$$|\lambda_{i_{k_1}}| > |\lambda_{\max I_{k_2}}| \geq |\lambda_i|. \tag{27}$$

This completes the proof of Lemma 2. □

Lemma 2 holds. Thus, Theorem 1 holds. □

C Details of task distributions

C.1 Distributions of 1D Poisson equation

In P_s, u is represented by

$$u(x) = \sum_{i=1}^{20} a_i \sin(b_i \pi (x - c_i)), \tag{28}$$

where

$$a_i \quad N(0, 1) \tag{29}$$

$$b_i \quad \text{Unif}(0, 128) \tag{30}$$

$$c_i \quad \text{Unif}(0, 1). \tag{31}$$

In P_h, u is represented by

$$u(x) = \sum_{i=1}^{20} a_i \tanh(b_i \pi (x - c_i)), \tag{32}$$

where

$$a_i \quad N(0, 10) \tag{33}$$

$$b_i \quad \text{Unif}(0, 30) \tag{34}$$

$$c_i \quad \text{Unif}(0, 1). \tag{35}$$

C.2 DISTRIBUTION OF INFLOW

In P_{in}, $v_{\text{in}} = (v_1, v_2)^T$ is represented by

$$v_1(y,t) = w(y,t)\cos z(t) \tag{36}$$
$$v_2(y,t) = w(y,t)\sin z(t), \tag{37}$$

where

$$w(y,t) = 0.5 + 0.5\sin(ay \quad bt)\pi \sin y\pi \sin t\pi \tag{38}$$
$$z(t) = e\sin(ct \quad d)\pi \tag{39}$$

and

$$a \quad \text{Unif}(0,10) \tag{40}$$
$$b \quad \text{Unif}(\quad 5,5) \tag{41}$$
$$c \quad \text{Unif}(0,5) \tag{42}$$
$$d \quad \text{Unif}(0,2) \tag{43}$$
$$e \quad \text{Unif}(0,\frac{\pi}{2}). \tag{44}$$

D DETAILS OF NETWORK ARCHITECTURE AND HYPER-PARAMETERS

D.1 DETAILS OF SECTION 3.1.2

In section 3.1.2, $_{\text{NN}}$ is a variant of 1D U-Net with a residual connection to leverage the heuristic initial guess $u_h = \ _{\text{BL}}(\tau)$, so $_{\text{NN}}(\tau) = \ _{\text{NN}}(f \ , u_h) = u_h + \text{1DUNet}(f \ , u_h)$, where 1DUNet consists of four stages with halved resolutions. Each stage has two convolutional layers with kernel size 11 and the activation function tanh. In the first stage, the number of channels is 8, and it is doubled as the resolution is halved. By utilizing the linearity of the Poisson equation, inputs are normalized by $kf \ k$ before feeding them into $_{\text{NN}}$, and the final output $\hat{u}$ is denormalized by $kf \ k$. The model is trained for 2000 epochs by Adam with the learning rate 0.0005 and the batch-size 512. The learning rate is decreased by $1/5$ times when the validation loss does not improve for 200 epochs, and the training is terminated when the validation loss does not improve for 250 epochs. The model with the best validation loss is used for evaluation.

D.2 DETAILS OF SECTION 3.2

In section 3.2, $_{\text{NN}}$ is a variant of 2D U-Net. The differences from $_{\text{NN}}$ in section 3.1.2 are its dimension, kernel size 5, and the starting number of channels 16. The model is trained for 50 epochs by Adam with the learning rate 0.0005 and the batch-size 64, and the learning rate is decreased by $1/5$ at 30 epoch. Note that the meta-solver is trained for 10 time steps for each mini batch by the data augmentation. The model with the best validation loss is used for evaluation.

E DETAILS OF EXPERIMENT RESULTS

Table 3: Standard deviation of MSE of GBMS for solving Poisson equations.

(a) Standard deviation of MSE on P_s

	Trained with	Tested with Jac,0	Jac,4	Jac,16	Jac,64	SOR,4	SOR,16	SOR,64
NN	Jac,0 = SOR,0	0.0052	0.0052	0.0052	0.0050	0.0051	0.0049	0.0043
	Jac,4	0.0097	0.0101	0.0098	0.0093	0.0098	0.0091	0.0074
	Jac,16	0.5548	0.0038	0.0036	0.0033	0.0095	0.0024	0.0020
	Jac,64	4.8458	0.7034	0.0167	0.0037	0.0381	0.0040	0.0029
	SOR,4	0.1211	0.1005	0.0739	0.0734	0.0085	0.0080	0.0067
	SOR,16	10.4207	1.9210	0.2923	0.2396	0.0917	0.0072	0.0056
	SOR,64	4.8430	0.9012	0.3301	0.2788	0.1937	0.0095	0.0078

(b) Standard deviation of MSE on P_h

	Trained with	Tested with Jac,0	Jac,4	Jac,16	Jac,64	SOR,4	SOR,16	SOR,64
NN	Jac,0 = SOR,0	0.0194	0.0192	0.0189	0.0184	0.0187	0.0181	0.0186
	Jac,4	0.0273	0.0261	0.0258	0.0251	0.0257	0.0248	0.0223
	Jac,16	0.0588	0.0212	0.0195	0.0208	0.0198	0.0213	0.0240
	Jac,64	0.2278	0.1431	0.0513	0.0279	0.0369	0.0281	0.0288
	SOR,4	0.6228	0.6871	0.7175	0.7154	0.0286	0.0308	0.0324
	SOR,16	0.3757	0.3590	0.6168	0.7796	0.0309	0.0218	0.0224
	SOR,64	1.2871	0.9696	0.4896	0.0872	0.3847	0.0492	0.0309