LEARNING NEURAL PDE SOLVERS WITH CONVERGENCE GUARANTEES

Anonymous authors
Paper under double-blind review

ABSTRACT

Partial differential equations (PDEs) are widely used across the physical and computational sciences. Decades of research and engineering went into designing fast iterative solution methods. Existing solvers are general purpose, but may be sub-optimal for specific classes of problems. In contrast to existing hand-crafted solutions, we propose an approach to learn a fast iterative solver tailored to a specific domain. We achieve this goal by learning to modify the updates of an existing solver using a deep neural network. Crucially, our approach is proven to preserve strong correctness and convergence guarantees. After training on a single geometry, our model generalizes to a wide variety of geometries and boundary conditions, and achieves 2-3 times speedup compared to state-of-the-art solvers.

1 INTRODUCTION

Partial differential equations (PDEs) are ubiquitous tools for modeling physical phenomena, such as heat, electrostatics, and quantum mechanics. Traditionally, PDEs are solved with hand-crafted approaches that iteratively update and improve a candidate solution until convergence. Decades of research and engineering went into designing update rules with fast convergence properties.

The performance of existing solvers varies greatly across application domains, with no method uniformly dominating the others. Generic solvers are typically effective, but could be far from optimal for specific domains. In addition, high performing update rules could be too complex to design by hand. In recent years, we have seen that for many classical problems, complex updates *learned* from data or experience can out-perform hand-crafted ones. For example, for Markov chain Monte Carlo, learned proposal distributions lead to orders of magnitude speedups compared to hand-designed ones (Song et al., 2017; Levy et al., 2017). Other domains that benefited significantly include learned optimizers (Andrychowicz et al., 2016) and learned data structures (Kraska et al., 2018). Our goal is to bring similar benefits to PDE solvers.

Hand-designed solvers are relatively simple to analyze and are guaranteed to be correct in a large class of problems. The main challenge is how to provide the same guarantees with a potentially much more complex learned solver. To achieve this goal, we build our learned iterator on top of an existing standard iterative solver to inherit its desirable properties. The iterative solver updates the solution at each step, and we learn a parameterized function to modify this update. This function class is chosen so that for any choice of parameters, the fixed point of the original iterator is preserved. This guarantees correctness, and training can be performed to enhance convergence speed. Because of this design, we only train on a single problem instance; our model correctly generalizes to a variety of different geometries and boundary conditions with no observable loss of performance. As a result, our approach provides: (i) theoretical guarantees of convergence to the correct stationary solution, (ii) faster convergence than existing solvers, and (iii) generalizes to geometries and boundary conditions very different from the ones seen at training time. This is in stark contrast with existing deep learning approaches for PDE solving (Tang et al., 2017; Farimani et al., 2017) that are limited to specific geometries and boundary conditions, and offer no guarantee of correctness.

Our approach applies to any PDE with existing linear iterative solvers. As an example application, we solve the 2D Poisson equations. Our method achieves a 2-3$\times$ speedup on number of flops when compared to standard iterative solvers, even on domains that are significantly different from our

training set. Moreover, compared with state-of-the-art solvers implemented in FEniCS (Logg et al., 2012), our method achieves faster performance in terms of wall clock CPU time. Our method is also simple as opposed to deeply optimized solvers such as our baseline in FEniCS (minimal residual method + algebraic multigrid preconditioner). Finally, since we utilize standard convolutional networks which can be easily parallelized on GPU, our approach leads to an additional $30\times$ speedup when run on GPU.

2 BACKGROUND

In this section, we give a brief introduction of linear PDEs and iterative solvers. We refer readers to LeVeque (2007) for a thorough review.

2.1 LINEAR PDES

Linear PDE solvers find functions that satisfy a (possibly infinite) set of linear differential equations. More formally, let $\mathcal{F} = \{u : \mathbb{R}^k \to \mathbb{R}\}$ be the space of candidate functions, and $\mathcal{A} : \mathcal{F} \to \mathcal{F}$ be a linear operator; the goal is to find a function $u \in \mathcal{F}$ that satisfies a given linear equation $\mathcal{A}u = f$. Many PDEs fall into this framework. For example, heat diffusion satisfies the Poisson equation $\nabla^2 u = f$, where $\nabla^2 = \frac{\partial^2}{\partial x_1^2} + \cdots + \frac{\partial^2}{\partial x_k^2}$ is the linear Laplace operator.

Usually the equation $\mathcal{A}u = f$ does not uniquely determine u. For example, $u = \text{constant}$ for any constant is a solution to the equation $\nabla^2 u = 0$. To ensure a unique solution we provide additional equations, called "boundary conditions". Several boundary conditions arise very naturally in physical problems. A very common one is the Dirichlet boundary condition, where we pick some subset $\mathcal{G} \subset \mathbb{R}^k$ and fix the values of the function on $\mathcal{G}$,

$$u(x) = b(x), \text{ for all } x \in \mathcal{G}$$

where the function b is usually clear from the underlying physical problem. We define $\mathcal{G}$ as the set of points in the boundary, and refer to it as the *geometry* of the problem. We will refer to b as the *boundary values*, and the pair $(\mathcal{G}, b)$ as the *boundary condition*. In this paper, we only consider linear PDEs and boundary conditions that have unique solutions.

2.2 FINITE DIFFERENCE METHOD

Most real-world PDEs do not admit an analytic solution and must be solved numerically. The first step is to discretize the solution space $\mathcal{F}$ from $\mathbb{R}^k \to \mathbb{R}$ into $\mathbb{D}^k \to \mathbb{R}$, where $\mathbb{D}$ is a discrete subset of $\mathbb{R}$. When the space is compact, it is discretized into an $n \times n \times n \cdots$ (k many) uniform Cartesian grid with mesh width h. Any function in $\mathcal{F}$ is approximated by its value on the n^k grid points. We denote the discretized function as a vector u in $\mathbb{R}^{n^k}$. In this paper, we focus on 2D problems ($k = 2$), but the strategy applies to any dimension.

We discretize all three terms in the equation $\mathcal{A}u = f$ and boundary values b. The PDE solution u is discretized such that $u_{i,j} = u(x_i, y_j)$ corresponds to the value of u at grid point (x_i, y_j). We can similarly discretize f and b. In linear PDEs, the linear operator $\mathcal{A}$ is a linear combination of partial derivative operators. For example, for the Poisson equation $\mathcal{A} = \nabla^2 = \sum_i \frac{\partial^2}{\partial x_i^2}$. Therefore we can first discretize each partial derivative, then linearly combine the discretized partial derivatives to obtain a discretized $\mathcal{A}$.

Finite difference is a method that approximates partial derivatives in a discretized space, and as mesh width $h \to 0$, the approximation approaches the true derivative. For example, $\frac{\partial^2}{\partial x^2} u$ can be discretized in 2D as $\frac{\partial^2}{\partial x^2} u \approx \frac{1}{h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j})$, the Laplace operator in 2D can be correspondingly approximated as:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \approx \frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) \tag{1}$$

After discretization, we can rewrite $\mathcal{A}u = f$ as a linear matrix equation

$$Au = f \tag{2}$$

where $u, f \in \mathbb{R}^{n^2}$, and A is a matrix in $\mathbb{R}^{n^2 \times n^2}$. In many PDEs such as the Poisson and Helmholtz equation, A is sparse, banded, and symmetric.

2.3 Boundary Condition

We also need to include the boundary condition $\mathcal{u}(x) = \mathcal{b}(x)$ for all x in the boundary set $\mathcal{G}$. If a discretized point (x_i, y_j) belongs to the boundary set $\mathcal{G}$, we need to fix the value of $u_{i,j}$ to $b_{i,j}$. To achieve this, we first define $e \in \{0, 1\}^{n^2}$ to be a vector of 0's and 1's, in which 0 indicates that the corresponding point is a boundary point. Then, we define a "reset" matrix $G = diag(e)$, a diagonal matrix $\mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ such that

$$(Gu)_{i,j} = \begin{cases} u_{i,j} & (x_i, y_j) \notin \mathcal{G} \\ 0 & (x_i, y_j) \in \mathcal{G} \end{cases} \tag{3}$$

G can be interpreted as a mask of the boundary points, which define the geometry of the domain. Note that the boundary values are fixed and do not need to satisfy $Au = f$. Thus, the solution u to the PDE in a geometry $\mathcal{G}$ should satisfy:

$$\begin{aligned} G(Au) &= Gf \\ (I - G)u &= (I - G)b \end{aligned} \tag{4}$$

The first equation ensures that the interior points satisfy $Au = f$, and the second ensures that the boundary values are correct. To satisfy the second condition, after each iteration, we "reset" the values of $u_{i,j}$ to $b_{i,j}$ by $Gu + (I - G)b$, masking out the boundary points to 0 and adding the boundary values back. G has some useful properties: $G^2 = G$ and $G(I - G) = 0$. This is discussed further in Appendix B.

To summarize, $(\mathcal{A}, \mathcal{G}, \mathcal{f}, \mathcal{b}, n)$ is our *PDE problem*, and we first discretize the problem on an $n \times n$ grid to obtain (A, G, f, b, n). Our objective is to obtain a solution u that satisfies Eq. (4), i.e. $Au = f$ for the interior points and boundary condition $u_{i,j} = b_{i,j}, \ \forall (x_i, y_j) \in \mathcal{G}$.

2.4 Iterative Solvers

A *linear* iterative solver $\Psi : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ can be expressed as

$$u' = \Psi(u) = Tu + c \tag{5}$$

where T is a constant update matrix and c is a constant vector. For each iterator Ψ there may be special vectors $u^* \in \mathbb{R}^{n^2}$ that satisfy $u^* = \Psi(u^*)$. These vectors are called fixed points.

Definition 1 (Valid Iterator)**.** *An iterator Ψ is valid w.r.t. a PDE problem (A, G, f, b, n) if it satisfies:*

a) There is a unique fixed point u^* such that Ψ converges to u^* from any initialization: $\forall u^0 \in \mathbb{R}^{n^2}, \lim_{k \to \infty} \Psi^k(u^0) = u^*$.

b) The fixed point u^* is the solution to the linear system $Au = f$ under boundary condition (G, b).

Condition (a) in Definition 1 is satisfied if the matrix T is *convergent*, i.e. $T^k \to 0$ as $k \to \infty$. It has been proven that T is convergent if and only if the spectral radius $\rho(T) < 1$ (Olver, 2008):

Theorem 1. *(Olver, 2008, Prop 7.25) For a linear iterator $\Psi(u) = Tu + c$, Ψ converges to a unique stable fixed point from any initialization if and only if the spectral radius $\rho(T) < 1$.*

Proof. See Appendix A. □

It is important to note that Condition (a) only depends on T and not the constant c.

To satisfy Condition (b) in Definition 1, a standard approach is to design Ψ by matrix splitting: split the matrix A into $A = M - N$; rewrite $Au = f$ as $Mu = Nu + f$ (LeVeque, 2007). This naturally suggests the iterative update

$$u' = M^{-1}Nu + M^{-1}f \tag{6}$$

By design, stationary points of Eq. (6) satisfy $Au^* = f$. Clearly, the choices of M and N are arbitrary but crucial. From Theorem 1, we must choose M such that the update converges and M^{-1} is easy to compute (e.g., diagonal). In addition, to satisfy boundary condition (G, b) we use the reset operator defined in Eq. (3) and obtain the final update rule

$$u' = G(M^{-1}Nu + M^{-1}f) + (I - G)b \tag{7}$$

This update rule is still linear. Comparing with Eq. (5), we have $T = GM^{-1}N$ and $c = GM^{-1}f + (1 - G)b$.

2.4.1 JACOBI METHOD

A simple but effective way to split the matrix is the Jacobi method, which sets $M = I$. The Jacobi update is then $u' = (I - A)u + f$. For Poisson equations, this update rule has the following form,

$$\hat{u}_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}) + \frac{h^2}{4} f_{i,j} \tag{8}$$

$$u' = G\hat{u} + (1 - G)b \tag{9}$$

For Poisson equations and any geometry G, the update matrix $T = G(I - A)$ has spectral radius $\rho(T) < 1$ (see Appendix B.1). In addition, each step of the Jacobi update can be implemented as a neural network layer, i.e., Eq. (8) can be efficiently implemented by convolving u with kernel $\begin{pmatrix} 0 & 1/4 & 0 \\ 1/4 & 0 & 1/4 \\ 0 & 1/4 & 0 \end{pmatrix}$ and adding $h^2 f/4$. The "reset" step in Eq. (9) can also be implemented as multiplying u with G and adding the boundary values $(1 - G)b$.

2.4.2 MULTIGRID METHOD

The Jacobi method has very slow convergence rate (LeVeque, 2007). This is evident from the update rule, where the value at each grid point is only influenced by its immediate neighbors. To propagate information from one grid point to another, we need as many iterations as their distance on the grid. The key insight of the Multigrid method is to perform Jacobi updates on a downsampled (coarser) grid and then upsample the results. A common structure is the V-cycle (Briggs et al., 2000). In each V-cycle, there are k downsampling layers followed by k upsampling layers, and multiple Jacobi updates are performed at each resolution. The downsampling and upsampling operations are also called *restriction* and *prolongation*, and are often implemented using weighted restriction and linear interpolation respectively. The advantage of the multigrid method is clear: on a downsampled grid (by a factor of 2) with mesh width $2h$, information propagation is twice as fast, and each iteration requires only 1/4 operations compared to the original grid with mesh width h.

3 LEARNING FAST AND PROVABLY CORRECT ITERATIVE PDE SOLVERS

A PDE problem consists of five components $(\mathcal{A}, \mathcal{G}, \mathcal{f}, \mathcal{b}, n)$. One is often interested in solving the same PDE class $\mathcal{A}$ under varying $\mathcal{f}$, discretization n, and boundary conditions $(\mathcal{G}, \mathcal{b})$. For example, solving the Poisson equation under different boundary conditions (e.g., corresponding to different mechanical systems governed by the same physics). In this paper, we fix $\mathcal{A}$ but vary $\mathcal{G}, \mathcal{f}, \mathcal{b}, n$, and learn an iterator that solves a class of PDE problems governed by the same $\mathcal{A}$. For a discretized PDE problem (A, G, f, b, n) and given a standard (hand designed) iterative solver Ψ, our goal is to improve upon Ψ and learn a solver $\Phi(u; G, f, b, n)$ that has (1) correct fixed point and (2) fast convergence (on average) on the class of problems of interest. We will proceed to parameterize a family of Φ that satisfies (1) by design, and achieve (2) by optimization.

In practice, we can only train $\Phi(u; G, f, b, n)$ on a small number of problems (A, f_i, G_i, b_i, n_i). To be useful, Φ must deliver good performance on every choice of G, f, b, and different grid sizes n. We show, theoretically and empirically, that our iterator family has good generalization properties: even if we train on a single problem (A, G, f, b, n), the iterator performs well on very different choices of G, f, b, and grid size n. For example, we train our iterator on a 64×64 square domain, and test on a 256×256 L-shaped domain (see Figure 1).

3.1 FORMULATION

For a fixed PDE problem (A, G, f, b, n), let Ψ be a standard linear iterative solver which is known to be valid, $\Psi(u) = Tu + c$. Note that $\Psi(u)$ should already satisfy the boundary conditions using the reset operator, for example, $T = G(I - A)$ and $c = Gf + (1 - G)b$ for Jacobi iteration in Eq. (9). We formulate our family of iterators $\Phi_H : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ as

$$\begin{aligned} w &= \Psi(u) - u \\ \Phi_H(u) &= \Psi(u) + GHw \end{aligned} \tag{10}$$

where H is a learned linear operator which satisfies $H0 = 0$. The term GHw can be interpreted as a correction term to $\Psi(u)$. Observe that the computation process of Ψ and Φ_H defined in Eq. (8,9,10) implicitly depend on f, G, b, n, so we formally denote them as $\Psi(u; f, G, b, n)$ and $\Phi_H(u; f, G, b, n)$. For simplicity, we neglect the dependence and denote as $\Psi(u)$ and $\Phi_H(u)$.

Φ_H should have similar computation complexity as Ψ. Therefore, we choose H to be a convolutional operator, which can be parameterized by a deep linear convolutional network (i.e., a sparse, circulant matrix). We will discuss the parameterization of H in detail in Section 3.4; we first prove some parameterization independent properties.

The correct PDE solution is a fixed point of Φ_H by the following lemma:

Lemma 1. *For any PDE problem (A, G, f, b, n) and choice of H, if u^* is a fixed point of Ψ, it is a fixed point of Φ_H in Eq. (10).*

Proof. Based on the iterative rule in Eq. (10), if u^* satisfies $\Psi(u^*) = u^*$ then $w = \Psi(u^*) - u^* = \mathbf{0}$. Therefore, $\Phi_H(u^*) = \Psi(u^*) + GH\mathbf{0} = u^*$. □

Moreover, the space of Φ_H subsumes the standard solver Ψ. If $H = 0$, then $\Phi_H = \Psi$. Furthermore, if $H = T$, then since $GT = T$ (see Eq. (7)),

$$\Phi_H(u) = \Psi(u) + GT(\Psi(u) - u) = T\Psi(u) + c = \Psi^2(u) \tag{11}$$

which is equal to two iterations of Ψ. Because computing Φ requires two convolutions: H and T, one iteration with Φ_H has the same number of convolution operations as two iterations of Ψ. This shows that we can learn an H such that our iterator Φ_H is at least as good as the standard solver Ψ.

3.2 TRAINING AND GENERALIZATION

We train our iterator $\Phi_H(u; G, f, b, n)$ to converge quickly to the ground truth solution on a set $\mathcal{D} = \{(G_l, f_l, b_l, n_l)\}_{l=1}^M$ of problem instances. For each instance, the ground truth solution u^* is obtained from the existing solver Ψ. The learning objective is then

$$\min_H \sum_{(G_l, f_l, b_l, n_l) \in \mathcal{D}} \mathbb{E}_{u^0 \sim \mathcal{N}(0,1)} \|\Phi_H^k(u^0; G_l, f_l, b_l, n_l) - u^*\|_2^2 \tag{12}$$

Intuitively, we look for a matrix H such that the corresponding iterator Φ_H will get us as close as possible to the solution in k steps, starting from a random initialization u^0 sampled from a white Gaussian. k in our experiments is uniformly chosen from $[1, 20]$, similar to the procedure in Song et al. (2017).

We show in the following theorem that there is a convex open set of H that the learning algorithm can explore. To simplify the statement of the theorem, for iterator $\Phi(u) = Tu + c$ we will refer to the spectral radius (norm) of Φ as the spectral radius (norm) of T.

Theorem 2. *For fixed G, f, b, n, the spectral norm of $\Phi_H(u; G, f, b, n)$ is a convex function of H, and the set of H such that the spectral norm of $\Phi_H(u; G, f, b, n) < 1$ is a convex open set.*

Proof. See Appendix A. □

Therefore, to find a iterator with small spectral norm, the learning algorithm only has to explore a convex open set. Note that Theorem 2 holds for *spectral norm*, whereas validity requires small

spectral radius in Theorem 1. Nonetheless, several important PDE problems (Poisson, Helmholtz, etc) are symmetric, so it is natural to use a symmetric iterator, which means that spectral norm is equal to spectral radius. In our experiments, we do not explicitly enforce symmetry, but we observe that the optimization finds symmetric iterators automatically.

For training, we use a single grid size n, a single geometry G, $f = 0$, and a restricted set of boundary conditions b. The geometry we use is a square domain shown in Figure 1a. Although we train on a single domain, the model has surprising generalization properties, which we show in the following:

Proposition 1. *For fixed A, G, n and fixed H, if for some f_0, b_0, $\Phi_H(u; G, f_0, b_0, n)$ is valid for the PDE problem (A, G, f_0, b_0, n), then for all f and b, the iterator $\Phi_H(u; G, f, b, n)$ is valid for the PDE problem (A, G, f, b, n).*

Proof. See Appendix A. □

The proposition states that we freely generalize to different f and b. There is no guarantee that we can generalize to different G and n. Generalization to different G and n has to be empirically verified: in our experiments, our learned iterator converges to the correct solution for a variety of grid sizes n and geometries G, even though it was only trained on one grid size and geometry.

Even when generalization fails, there is no risk of obtaining incorrect results. The iterator will simply diverge. This is because according to Lemma 1, fixed points of our new iterator is the same as the fixed point of hand designed iterator Ψ. Therefore if our iterator is convergent, it is valid.

3.3 Interpretation of H

What is H trying to approximate? From Eq. (10), we see that GHw is an additional term added to the original update $\Psi(u) = Tu + c$. Let the unknown ground truth solution be u^* satisfying $u^* = Tu^* + c$. We would like GHw to approximate the error $e = u^* - \Psi(u)$, given $w = \Psi(u) - u$. Observe that

$$e = u^* - \Psi(u) = u^* - (Tu + c) = T(u^* - u) = Tr \tag{13}$$

where $r = u^* - u$ is current error. Moreover, we derive the expression of w,

$$w = \Psi(u) - u = Tu + c - u + (u^* - Tu^* - c) = T(u - u^*) + (u^* - u) = (I - T)r \tag{14}$$

Combining Eq. (13) and 14, we obtain

$$e = T(I - T)^{-1}w \tag{15}$$

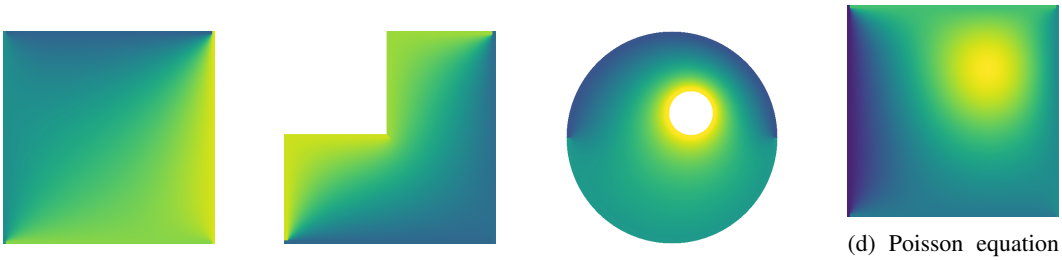
where $(I - T)^{-1}$ exists because $\rho(T) < 1$, so all eigenvalues of $I - T$ must be strictly positive. Therefore, we would like our linear function GH to approximate $GH \approx T(I - T)^{-1}$. Note that $(I - T)^{-1}$ is usually a dense matrix, meaning that it is impossible to obtain the exact error with a convolutional operator H. However, the better GHw is able to approximate Eq. (15), the faster our iterator converges to the solution u^*.

The most important observation from Eq. (15) is that it does not depend on f or b. Therefore, we can learn an H from a small family of f_0, b_0, and it can naturally generalize to any f, b.

3.4 Linear Deep Networks

In our iterator design, H is a linear function parameterized by a linear deep network without non-linearity or bias terms. Even though our objective in Eq. (12) is a non-linear function of the parameters of the deep network, this is not an issue in practice. In particular, Arora et al. (2018) observes that when modeling linear functions, deep networks can be faster to optimize with gradient descent compared to linear ones, despite non-convexity.

Even though a linear deep network can only represent a linear function, it has several advantages. Each convolution layer only requires $O(n^2)$ computation and have a constant number of parameters, while a general linear function requires $O(n^4)$ computation and have $O(n^4)$ parameters. Stacking d convolution layers allows us to parameterize complex linear functions with large receptive fields, while only requiring $O(dn^2)$ computation and $O(d)$ parameters. We experiment on two types of linear deep networks:



(a) Square domain. (b) L-shape domain. (c) Cylinders domain. (d) Poisson equation in the square domain.

Figure 1: The ground truth solutions of examples in different settings. We only train our models on the square domain, and we test on all 4 settings.

Conv model. We model H as a network with 3×3 convolutional layers without non-linearity or bias. We will refer to a model with k layers as "Conv$k$", e.g. Conv3 has 3 convolutional layers.

U-Net model. The Conv models suffer from the same problem as Jacobi: the receptive field grows only by 1 for each additional layer. To resolve this problem, we design the deep network counterpart of the Multigrid method. Instead of manually designing the sub-sampling / super-sampling functions, we use a U-Net architecture (Ronneberger et al., 2015) to learn them from data. Because each layer reduces the grid size by half, and the i-th layer of the U-Net only operates on $(2^{-i}n)$-sized grids, the total computation is only increased by a factor of

$$1 + 1/4 + 1/16 + \cdots < 4/3$$

compared to a two-layer convolution. The minimal overhead provides a very large improvement of convergence speed in our experiments. We will refer to Multigrid and U-Net models with k sub-sampling layers as Multigrid$k$ and U-Net$k$, e.g. U-Net2 is a model with 2 sub-sampling layers.

4 Experiments

4.1 Setting

We evaluate our method on the 2D Poisson equation with Dirichlet boundary conditions, $\nabla^2 u = f$. There exist several iterative solvers for the Poisson equation, including Jacobi, Gauss-Seidel, conjugate-gradient, and multigrid methods. We select the Jacobi method as our standard solver Ψ.

To reemphasize, our goal is to train a model on simple domains where the ground truth solutions can be easily obtained, and then evaluate its performance on different geometries and boundary conditions. Therefore, we select the simplest Laplace equation, $\nabla^2 u = 0$, on a square domain with boundary conditions such that each side is a random fixed value. Figure 1a shows an example of our training domain and its ground truth solution. This setting is also used in Farimani et al. (2017) and Sharma et al. (2018).

For testing, we set the testing domains to be in larger grid sizes than the training domain, for example, test on 256×256 grid for a model trained on 64×64 grids. Moreover, we designed challenging geometries to test the generalization of our models. We test generalization on 4 different settings: (i) same geometry but larger grid, (ii) L-shape geometry, (iii) Cylinders geometry, and (iv) Poisson equation in same geometry, where $f \neq 0$. The two geometries are designed because the models were trained on square domains and have never seen sharp or curved boundaries. For Poisson equation, we pick f to be a function shown in Figure 1d. Examples of the 4 settings are shown in Figure 1.

4.2 Evaluation

As discussed in Section 2.4, the convergence rate of any linear iterator can be determined from the spectral radius $\rho(T)$, which provides guarantees on convergence and convergence rate. However, a fair comparison should also consider the computation cost of H. Thus, we evaluate the convergence rate by calculating the computation cost required for the error to drop below a certain threshold.

Table 1: Comparisons between our models and the baseline solvers. The Conv models are compared with Jacobi, and the U-Net models are compared with Multigrid. The numbers are the ratio between the computation costs of our models and the baselines. None of the values are greater than 1, which means that all of our models achieve a speed up on every problem and every performance metric (layers and flops).

Model	**Baseline**	**Square** layers / flops	**L-shape** layers / flops	**Cylinders** layers / flops	**Square-Poisson** layers / flops
Conv1	Jacobi	0.432 / 0.702	0.432 / 0.702	0.432 / 0.702	0.431 / 0.701
Conv2	Jacobi	0.286 / 0.524	0.286 / 0.524	0.286 / 0.524	0.285 / 0.522
Conv3	Jacobi	**0.219 / 0.424**	**0.219 / 0.423**	**0.220 / 0.426**	**0.217 / 0.421**
Conv4	Jacobi	0.224 / 0.449	0.224 / 0.449	0.224 / 0.448	0.222 / 0.444
U-Net2	Multigrid2	**0.091 / 0.205**	**0.090 / 0.203**	**0.091 / 0.204**	**0.079 / 0.178**
U-Net3	Multigrid3	0.220 / 0.494	0.213 / 0.479	0.201 / 0.453	0.185 / 0.417

On GPU, the Jacobi iterator and our model can both be efficiently implemented as convolutional layers. Thus, we measure the computation cost by the number of convolutional layers. On CPU, each Jacobi iteration $u'_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})$ has 4 multiply-add operations, while a 3×3 convolutional kernel requires 9 operations, so we measure the computation cost by the number of multiply-add operations. This metric is biased in favor of Jacobi because there is little practical reason to implement convolutions on CPU. Nonetheless, we report both metrics in our experiments.

4.3 Conv Model

Table 1 shows results of the Conv model. The model is trained on a 16×16 square domain, and tested on 64×64. For all settings, our models converge to the correct solution, and require less computation than Jacobi. The best model, Conv3, is $\sim 5\times$ faster than Jacobi in terms of layers, and $\sim 2.5\times$ faster in terms of flops.

As discussed in Section 3.2, if our iterator converges for a geometry, then it is guaranteed to converge to the correct solution for any f and boundary values b. The experiment results show that our model not only converges but also converges faster than the standard solver, even though it is only trained on a smaller square domain.

4.4 U-Net Model

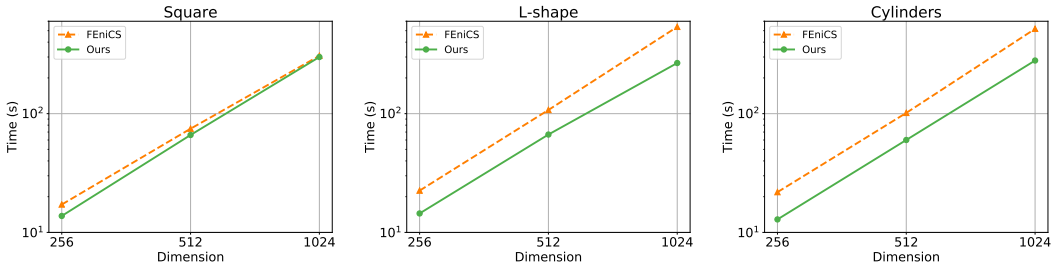
For the U-Net models, we compare them against Multigrid models with the same number of subsampling and smoothing layers. Therefore, our models have the same number of convolutional layers, and roughly $9/4$ times the number of flops compared to Multigrid. The model is trained on a 64×64 square domain, and tested on 256×256.

The bottom part of Table 1 shows the results of the U-Net model. Similar to the results of Conv models, our models outperforms Multigrid in all settings. Note that U-Net2 has a lower cost ratio than U-Net3 since the iterator benefits more from our learned model for a slower baseline. However, note that U-Net3 still converges faster than U-Net2 due to its larger receptive field.

4.5 Comparison with FEniCS

The FEniCS package (Logg et al., 2012) provides a collection of tools with high-level Python and C++ interfaces to solve differential equations. The open-source project is developed and maintained by a global community of scientists and software developers. Its extensive optimization over the years, including the support for parallel computation, has led to its widespread adaption in industry and academia (Alnæs et al., 2015).

We measure the wall clock time of the FEniCS model and our model, run on the same hardware. The FEniCS model is set to be the minimal residual method with algebraic multigrid preconditioner, which we measure to be the fastest compared to other methods such as Jacobi or Incomplete LU



(a) Square domain. (b) L-shape domain. (c) Cylinders domain.

Figure 2: CPU runtime comparisons of our model with the FEniCS model. Our method is comparable or faster than the best solver in FEniCS in all cases. When run on GPU, our solver provides an additional $30\times$ speedup.

factorization preconditioner. We ignore the time it takes to set up geometry and boundary conditions, and only consider the time the solver takes to solve the problem. We set the error threshold to be 1 percent of the initial error. For the square domain, we use a quadrilateral mesh. For the L-shape and cylinder domains, however, we let FEniCS generate the mesh automatically, while ensuring the number of mesh points to be similar.

Figure 2 shows that our model is comparable or faster than FEniCS in wall clock time. Moreover, the time scales linearly with the number of vertices, as expected. These experiments are all done on CPU. Our model can be efficiently run on GPU, and we measure an additional $30\times$ speedup (on Tesla K80 GPU, compared with a 64-core CPU).

5 Related Work

Recently, there have been several works on applying deep learning to solve the Poisson equation. However, to the best of our knowledge, previous works used deep networks to directly generate the solution; they have no correctness guarantees and are not generalizable to arbitrary grid sizes and boundary conditions. Most related to our work are Farimani et al. (2017) and Sharma et al. (2018), which learn deep networks to output the solution of the 2D Laplace equation (a special case where $f = 0$). Farimani et al. (2017) trained a U-Net model that takes in the boundary condition as a 2D image and outputs the solution, using the L1 loss to the ground truth solution and an adversarial loss from a discriminator. Sharma et al. (2018) also trained a U-net model but used a weakly-supervised loss. There are other related works that solved the Poisson equation in various physical problems. Tang et al. (2017) solved the electric potential in 2D/3D space with 2D/3D CNN; Tompson et al. (2017) used a CNN to solve the pressure fields determined by the Poisson equation for fluid simulation; Zhang et al. (2018) solved the Poisson equation in particle simulation of a PN Junction.

There are other works that solve other types of PDEs. For example, many studies aimed to use deep learning to accelerate and approximate fluid dynamics, governed by the Euler equation or the Navier-Stokes equations (Guo et al., 2016; Yang et al., 2016; Chu & Thuerey, 2017; Kutz, 2017). Other applications include solving the Schrodinger equation (Mills et al., 2017), turbulence modeling (Singh et al., 2017), and the American options and Black Scholes PDE (Sirignano & Spiliopoulos, 2018). A lot of these PDEs are nonlinear and may not have a standard linear iterative solver, which is a limitation to our current method since our model must be built on top of an existing linear solver to ensure correctness. We consider the extension to different PDEs as future work.

6 Conclusion

We presented a method to learn an iterative solver for PDEs that improves on an existing standard solver. The correct solution is theoretically guaranteed to be the fixed point of our iterator. We show that our model, trained on simple domains, can generalize to different grid sizes, geometries and boundary conditions. It converges correctly and achieves significant speedups compared to standard solvers, including highly optimized ones implemented in FEniCS.

REFERENCES

Martin S Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100):9–23, 2015.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, 2016.

Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. *arXiv preprint arXiv:1802.06509*, 2018.

William L Briggs, Steve F McCormick, et al. *A multigrid tutorial*, volume 72. Siam, 2000.

Mengyu Chu and Nils Thuerey. Data-driven synthesis of smoke flows with cnn-based feature descriptors. *ACM Transactions on Graphics (TOG)*, 36(4):69, 2017.

Amir Barati Farimani, Joseph Gomes, and Vijay S Pande. Deep learning the physics of transport phenomena. *arXiv preprint arXiv:1709.02432*, 2017.

Stanley P Frankel. Convergence rates of iterative treatments of partial differential equations. *Mathematical Tables and Other Aids to Computation*, 4(30):65–75, 1950.

Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 481–490, 2016.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 489–504. ACM, 2018.

J Nathan Kutz. Deep learning in fluid dynamics. *Journal of Fluid Mechanics*, 814:1–4, 2017.

Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, volume 98. Siam, 2007.

Daniel Levy, Matthew D Hoffman, and Jascha Sohl-Dickstein. Generalizing hamiltonian monte carlo with neural networks. *arXiv preprint arXiv:1711.09268*, 2017.

Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*. Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8.

Kyle Mills, Michael Spanner, and Isaac Tamblyn. Deep learning and the schrödinger equation. *Physical Review A*, 96(4):042113, 2017.

Peter J Olver. Numerical solution of ordinary differential equations. *Numerical Analysis Lecture Notes*, 2008.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

Rishi Sharma, Amir Barati Farimani, Joe Gomes, Peter Eastman, and Vijay Pande. Weakly-supervised deep learning of heat transport via physics informed loss. *arXiv preprint arXiv:1807.11374*, 2018.

Anand Pratap Singh, Shivaji Medida, and Karthik Duraisamy. Machine-learning-augmented predictive modeling of turbulent separated flows over airfoils. *AIAA Journal*, pp. 2215–2227, 2017.

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

Jiaming Song, Shengjia Zhao, and Stefano Ermon. A-nice-mc: Adversarial training for mcmc. In *Advances in Neural Information Processing Systems*, 2017.

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 *IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*. IEEE, 2017.

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *International Conference on Machine Learning*, 2017.

Cheng Yang, Xubo Yang, and Xiangyun Xiao. Data-driven projection method in fluid simulation. *Computer Animation and Virtual Worlds*, 27(3-4):415–424, 2016.

Zhongyang Zhang, Ling Zhang, Ze Sun, Nicholas Erickson, Ryan From, and Jun Fan. Solving poisson's equation using deep learning in particle simulation of pn junction. *arXiv preprint arXiv:1810.10192*, 2018.

A PROOFS

Theorem 1. *For a linear iterator $\Psi(u) = Tu + c$, Ψ converges to a unique stable fixed point from any initialization if and only if the spectral radius $\rho(T) < 1$.*

Proof. Suppose $\rho(T) < 1$, then $(I-T)^{-1}$ must exist because all eigenvalues of $I-T$ must be strictly positive. Let $u^* = (I-T)^{-1}c$; this u^* is a stationary point of the iterator Ψ, i.e. $u^* = Tu^* + c$. For any initialization u^0, let $u^k = \Psi^k(u^0)$. The error $e^k = u^* - u^k$ satisfies

$$Te^k = (Tu^* + c) - (Tu^k + c) = u^* - u^{k+1} = e^{k+1} \Rightarrow e^k = T^k e^0 \tag{16}$$

Since $\rho(T) < 1$, we know $T^k \to 0$ as $k \to \infty$ (LeVeque, 2007), which means the error $e^k \to 0$. Therefore, Ψ converges to u^* from any u^0. This also implies stability, as any small deviation from u^* will converge to u^*.

Now suppose $\rho(T) \geq 1$. Let λ_1 be the largest absolute eigenvalue where $\rho(T) = |\lambda_1| \geq 1$, and v_1 be its corresponding eigenvector. Consider $e^0 = av_1$ for some constant a. Then, $T^k e^0 = a\lambda_1^k v_1$. If $\lambda_1 > 1$ or $\lambda_1 < -1$, then $a\lambda_1^k v_1$ diverges. If $\lambda_1 = -1$, then $a\lambda_1^k v_1 = a(-1)^k v_1$ oscillates with respect to k, which does not converge. Finally, if $\lambda_1 = 1$, then $a\lambda_1^k v_1 = av_1$. For another vector $\tilde{e}^0 = bv_1$ where $a \neq b$, they will converge but to different fixed points, $T^k e^0 = av_1 \neq bv_1 = T^k \tilde{e}^0$. This means that they will converge but not to the same point, and thus the fixed points are not unique.

□

Theorem 2. *For fixed G, f, b, n, the spectral norm of $\Phi_H(u; G, f, b, n)$ is a convex function of H, and the set of H such that the spectral norm of $\Phi_H(u; G, f, b, n) < 1$ is a convex open set.*

Proof. Observe that

$$\Phi_H(u; G, f, b, n) = Tu + c + GH(Tu + c - u) = (T + GHT - GH)u + GHc + c \tag{17}$$

The spectral norm $\|\cdot\|_2$ is convex from the sub-additive property, and $(T + GHT - GH)$ is linear in H. Thus, $\|T + GHT - GH\|_2$ is convex in H as well. Thus, under the condition that $\|T + GHT - GH\|_2 < 1$, the set of H must be convex because it is a sub-level set of the convex function $\|T + GHT - GH\|_2$.

To prove that it is open, observe that $\|\cdot\|_2$ is a continuous function, so $\|T + GHT - GH\|_2$ is continuous in H. Given $\|T+GHT-GH\|_2 < 1$, the set of H is the preimage under this continuous function of $(0, 1-\epsilon)$ for some $\epsilon > 0$, and the inverse image of open set $(0, 1-\epsilon)$ must be open.

□

Proposition 1. *For fixed A, G, n and fixed H, if for some f_0, b_0, $\Phi_H(u; G, f_0, b_0, n)$ is valid for the PDE problem (A, G, f_0, b_0, n), then for all f and b, the iterator $\Phi_H(u; G, f, b, n)$ is valid for the PDE problem (A, G, f, b, n).*

Proof. From Theorem 1 and Lemma 1, our iterator is valid if and only if $\rho(T + GHT - GH) < 1$. The iterator $T + GHT - GH$ only depends on A, G, and is independent with the constant c in Eq. (17). Thus, the validity of the iterator is independent with f and b. Thus, if the iterator is valid for some f_0 and b_0, then it is valid for any choice of f and b.

□

B GEOMETRY

We define $\mathcal{G}$ as the geometry, or the set of boundary points, of the domain. For Dirichlet boundary conditions, $\mathcal{u}(x) = \mathcal{b}(x)$ must be satisfied for all $x \in \mathcal{G}$. To enforce this constraint, a straightforward implementation is to treat each point as an interior point, and after each iteration of the solver, we "reset" the boundary points $u_{i,j}$ to $b_{i,j}$. Recall that on an $n \times n$ grid, u, b, and f are vectors in

$\mathbb{R}^{n^2}$. We first define $e \in \{0,1\}^{n^2}$ to be a vector of 0's and 1's, where 0 and 1 indicate that the corresponding point is a boundary point and an interior point, respectively. Then, we define $G = diag(e)$ to be a diagonal matrix $\mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ such that

$$(Gu)_{i,j} = \begin{cases} u_{i,j} & (x_i, y_j) \notin \mathcal{G} \\ 0 & (x_i, y_j) \in \mathcal{G} \end{cases} \tag{18}$$

After each iteration we reset the boundary values by $Gu + (I - G)b$. The first term Gu corresponds to applying a mask to set all boundary points to 0, and the second term $(I - G)b$ adds the boundary values back. The reason of doing this is that since we treat every point as an interior point during each iteration, the values of the boundary points may be modified, and thus we need to reset them to be the correct boundary values. Note that there are some other update rules, such as Gauss-Seidel (LeVeque, 2007), that cannot be implemented as such.

There are several important properties of the operator G.

- $G^2 = G$ and thus $G(I - G) = 0$.
- For any u that satisfies the boundary conditions (not necessarily satisfying the PDE), $u = Gu + (I - G)b$.
- For an iterator $\Psi(u) = Tu + c$, if T is already multiplied by G, such as the Jacobi iterator $T = G(I - A)$, then $GT = T$.

Practically, G is simply a mask of 0's and 1's indicating where the interior points are, and mathematically G is represented as a diagonal matrix with 0's and 1's.

B.1 Jacobi Method for Poisson Equation

In Section 2.4.1, we show that for Poisson equation, the update matrix $T = G(I - A)$. We now formally prove that $\rho(G(I - A)) < 1$ for any G.

It has been proven that $\rho(I - A) < 1$ (Frankel, 1950). Since G is a diagonal matrix and A is symmetric, $G(I - A)$ is also a symmetric matrix, meaning that the spectral radius is equal to the spectral norm, $\rho(T) = \|T\|_2$. Moreover, $\rho(G) = 1$. Matrix norms are sub-multiplicative, so

$$\rho(T) = \|G(I - A)\|_2 \leq \|G\|_2 \|I - A\|_2 = \rho(I - A) < 1 \tag{19}$$

$\rho(T) < 1$ is true for any G. Thus, the standard Jacobi method is valid for the Poisson equation under any geometry.

B.2 Generalization

We showed in Proposition 1 that our model freely generalizes to different parameters f and b in the same geometry. However, given a new geometry G', even though the iterator is guaranteed to have a correct fixed point solution, there is no theoretical guarantee that the iterator will converge. In other words, if for a geometry G, the spectral radius $\rho(T + GHT - GH) < 1$, there is no guarantee that $\rho(T + G'HT + G'H) < 1$.

However, we can empirically check for convergence, as described in Section 3.2. In this new geometry, if the iterator converges for some (possibly random) parameter f and b under random initialization u^0, then with probability close to 1, the iterator is valid, i.e. spectral radius < 1 and guaranteed to converge to the correct solution for any parameters f and b. We also verify in our experiments that after training only on one geometry, the learned iterator is valid for a variety of geometries.