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). 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 equations. More formally, let $\mathcal{F} = \{u : \mathbb{R}^k \to \mathbb{R}\}$ be the space of candidate functions, and $\mathcal{L} : \mathcal{F} \to \mathcal{F}$ be a linear operator; the goal is to find a function $u \in \mathcal{F}$ that satisfies a given linear equation $\mathcal{L}u = f$. 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 $\mathcal{L}u = f$ 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 $G \subset \mathbb{R}^k$ and fix the values of the function on G,

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

where the function b is usually clear from the underlying physical problem. We refer to G as the boundary set of the problem. In this paper we only consider linear PDEs and boundary conditions that have unique solutions.

2.2 FINITE DIFFERENCE METHOD

Most real-world physical systems do not admit an analytic solution and must be solved numerically. The first step is to discretize the 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 $\mathbb{R}$. In the case of a compact 2D space, the space is often discretized into an $n \times n$ uniform Cartesian grid. Any function in $\mathcal{F}$ is approximated by its value on the n^2 grid points. We shall denote the discretized function as a vector in $\mathbb{R}^{n^2}$.

Based on this, we discretize all three terms in the equation $Lu = f$ and boundary condition 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. The linear operator $\mathcal{L}$ is approximated using the finite difference method. Finite difference is a method that approximates derivatives in a discretized space, and as $h \to 0$, the approximation approaches the true derivative. For example, the Laplace operator in 2D can be approximated as:

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

After discretization, we can rewrite $\mathcal{L}u = f$ 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 addition we also need to include the discretized boundary condition $u(x) = b(x)$ for all x in the boundary set G. If some (x_i, y_j) belongs to the boundary set G, we need to fix the value of $u_{i,j}$

to $b_{i,j}$. To achieve this we reset the values of $u_{i,j}$ to $b_{i,j}$ after each iteration of the iterative solver. We denote this "reset" as an operator $\mathcal{G} : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$. We will refer to G as the geometry of the problem, and f, b as the parameters of the problem. We make this distinction because in Section 4 they play different roles in the behavior of iterative solvers.

2.3 Iterative Solvers

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

$$u^{k+1} = \Psi(u^k) = Tu^k + c \tag{3}$$

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 Iterators)**.** *We say an iterator Ψ is valid if it satisfies the following:*

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

The first condition is satisfied if the spectral norm $\rho(T) < 1$ according to the following theorem:

Theorem 1. *(Kress, 1998) For any linear iterator $\Psi(u) = Tu + c$, if the spectral norm $\rho(T) < 1$, Ψ converges to a unique stable fixed point from any initialization.*

Proof. For completeness, a proof is reported in Appendix A. □

To satisfy the second condition, a standard approach is to design Ψ by matrix splitting: split the matrix A into $A = M - N$; rewrite $Au = f$ as $Mu = Nu + f$. This naturally suggests the iterative update

$$\begin{aligned} Mu^{k+1} &= Nu^k + f \\ u^{k+1} &= M^{-1}Nu^k + M^{-1}f \end{aligned} \tag{4}$$

Here, the update matrix $T = M^{-1}N$ and constant $c = M^{-1}f$. By design, the stationary point $u^* = Tu^* + c$ satisfies $Au^* = f$. Clearly, the choices of M and N are arbitrary but crucial. From Theorem 1, it is important that the spectral norm $\rho(M^{-1}N) < 1$. Moreover, the matrix M must be sufficiently simple such that Eq. (4) is much easier to solve than the original system. For example, M is often chosen to be a diagonal matrix. Most linear iterative solvers can be derived and analyzed with this matrix splitting technique.

2.3.1 Jacobi Method

A simple but effective way to split the matrix is the Jacobi method, which sets $M = I$. The Jacobi update is then $u^{k+1} = (I - A)u^k + f$.

For Poisson equations, this can also be derived by moving $u_{i,j}$ to the left-hand side in Eq. (1),

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k) - \frac{h^2}{4} f_{i,j} \tag{5}$$

It has been shown that the update matrix $T = I - A$ is symmetric, and has spectral radius $\rho(T) < 1$. In addition, Eq. (5) can be efficiently implemented as a convolution with kernel $\begin{pmatrix} 0 & 1/4 & 0 \\ 1/4 & 0 & 1/4 \\ 0 & 1/4 & 0 \end{pmatrix}$. Based on this observation, we can learn a more effective convolutional kernel in Section 4.

One practically important component we left out in the previous discussion is how to incorporate the geometry G and boundary condition b into the method. A simple approach is to apply the operator $\mathcal{G}$ after each Jacobi iterate: $\mathcal{G}$ "resets" the values for points (i, j) in G to the corresponding boundary values $b_{i,j}$. This modified iterator is guaranteed to converge to the solution of $Au = f$ under boundary condition G, b (Hackbusch, 1994).

2.3.2 MULTIGRID METHOD

The Jacobi method has very slow convergence rate (LeVeque, 2007). This is evident from the update rule of the Jacobi method, where the value at each grid point is only influenced by its 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 n downsampling layers followed by n upsampling layers, and multiple Jacobi updates are performed at each resolution. Even though theoretically it may not converge to the exact solution, it works well in practice. The advantages of the multigrid method is clear: for a downsampled grid (by a factor of 2), information propagation is twice as fast, and each iteration requires only 1/4 operations.

3 LEARNING FAST AND PROVABLY CORRECT ITERATIVE PDE SOLVERS

From Section 2.3.1 we observed that a convolutional operator can be used as a PDE solver. Given a set of geometries G and parameters f, b, this (hand designed) convolutional operator converges to the true solution. This suggests we might be able to train a different convolutional operator that converges to the same solution, only faster. Our key insight is that given a standard convolutional iterator $\Psi(u) = Tu + c$, we can improve upon it and design a new convolutional iterator that is guaranteed to be correct.

3.1 FORMULATION

We formulate our family of iterators $\Phi : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ as

$$\begin{aligned} w &= \Psi(u) - u \\ \Phi(u) &= \Psi(u) + Hw \end{aligned} \tag{6}$$

where H is a learned linear operator (which satisfies $H0 = 0$) represented as a circulant (convolution) matrix. The convolution matrix H can be parameterized by a deep linear network. We will discuss parameterization of H in detail in Section 3.4; we first prove some parameterization independent properties.

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

Lemma 1. *For any choice of H, if u^* is a fixed point of Ψ, it is a fixed point of Φ in Eq. (6).*

Proof. Based on the iterative rule in Eq. (6), if u^* satisfies $\Psi(u^*) = u^*$ then $w = \Psi(u^*) - u^* = \mathbf{0}$. Therefore, $\Phi(u^*) = \Psi(u^*) + H\mathbf{0} = u^*$. □

Moreover, the space of Φ subsumes the standard solver Ψ. If $H = 0$, then $\Phi = \Psi$. Furthermore, if $H = T$, then

$$\Phi(u) = \Psi(u) + T(\Psi(u) - u) = T\Psi(u) + b = \Psi^2(u) \tag{7}$$

which is equal to two iterations of Ψ. In this case, iterating with Φ is exactly equivalent to iterating with Ψ in terms of number of operations. This shows that with proper training, we can learn an H such that our iterator Φ is at least as good as the standard solver Ψ. In fact, we show in the following theorem that there is a convex open set of "valid" H that the learning algorithm can explore.

Theorem 2. *An H is valid if the corresponding iterator Φ in Eq. (6) is valid. In the normed vector space of matrices [1] , the spectral radius of Φ is a convex function of H, and the set of valid H is a convex open set.*

Proof. See Appendix A. □

[1] More technically, the vector space is $(R^{n^2 \times n^2}, +, \cdot)$ where $+$ and $\cdot$ are matrix addition and scalar multiplication. We assume the usual topology on $R^{n^2 \times n^2}$ (induced by Frobenius or any equivalent norm).

Therefore, the learning algorithm only has to explore a convex open set. The openness property implies that a valid H is robust against small perturbations, i.e., if H is valid, matrices in a small neighborhood will also be valid.

Note that in the actual iteration we must also take into consideration the geometry G and boundary condition b. Similar to Jacobi, we also apply $\mathcal{G}$ after each iteration to "reset" the values for points (i, j) in G to the corresponding boundary values $b_{i,j}$.

$$\begin{aligned} w &= \Psi(u) - u \\ \Phi(u) &= \mathcal{G}(\Psi(u) + Hw) \end{aligned} \tag{8}$$

This does not affect the correctness of fixed points because if u^* is a fixed point of Ψ, it is still a fixed point of Φ.

3.2 Training and Generalization

We train our model (by optimizing over H) to converge quickly to the ground truth solution on a set $\mathcal{D} = \{(G_l, f_l, b_l)\}_{l=1}^{M}$ of problem instances (with potentially different geometries G and parameters f, b). For each instance, the ground truth solution u^* is obtained from the existing solver Ψ. The learning objective is then

$$\min_{H} \sum_{(G_l, f_l, b_l) \in \mathcal{D}} \mathbb{E}_{u^0 \sim \mathcal{N}(0,1)} \|\Phi^k(u^0) - u^*\|_2^2 \tag{9}$$

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

For training, we use 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 theorem:

Theorem 3. *Under any geometry G, if H is valid for one parameter f and b, it is valid for all parameters f and b; H is valid for all parameters f and b if the iterator Φ converges to a fixed point under random initialization sampled from any continuous distribution.*

The theorem states that we freely generalize to different problem parameters f and b. However, the theorem is dependent on the geometry G. Across different geometries, even though correct fixed point is guaranteed, stable convergence is not. Despite the lack of theoretical guarantees, we verify in our experiments that after training only on one geometry, the learned iterator is valid for a variety of geometries. In addition, we can easily check for correctness because convergence implies correctness.

3.3 Interpretation of H

What is H trying to approximate? From Eq. (6), we see that $H(w)$ is an additional term added to the original update $\Psi(u)$. Let the unknown ground truth solution be u^*. We would like H to approximate the error $e = u^* - \Psi(u)$, given $w = \Psi(u) - u$. Observe that

$$\begin{aligned} e = u^* - \Psi(u) = u^* - (Tu + c) = T(u^* - u) = TA^{-1}(Au^* - Au) \\ = TA^{-1}(-Au + f) = M^{-1}NA^{-1}r \end{aligned} \tag{10}$$

where $r = -Au + f$ is the residual of the equation $Au = f$, and $T = M^{-1}N$ from Eq. (4). Moreover, from $A = M - N$, we derive the expression of w,

$$w = (Tu + c) - u = (M^{-1}Nu + M^{-1}f) - u = M^{-1}(-Au + f) = M^{-1}r \tag{11}$$

Combining Eq. (10) and 11, we obtain

$$H(w) \approx u^* - \Psi(u) = M^{-1}NA^{-1}Mw \tag{12}$$

Therefore, Eq. (12) is what we would like our linear function H to approximate. Note that A^{-1} is usually a dense matrix, meaning that it is impossible to obtain the exact error with a scalable function

H. However, the better $H(w)$ is able to approximate Eq. (12), the faster our iterator converges to the solution u^*.

The most important observation from Eq. (12) is that it does not depend on f (or b). Recall that f also includes the boundary conditions. Therefore, we can learn an H from a small family of f and boundary conditions, and it can naturally generalize to any f and boundaries.

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. (9) 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 require $O(n^2)$ computation and have a constant number of parameters, while a general linear function require $O(n^4)$ computation and have $O(n^4)$ parameters. Stacking d convolution layers allow 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:

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.

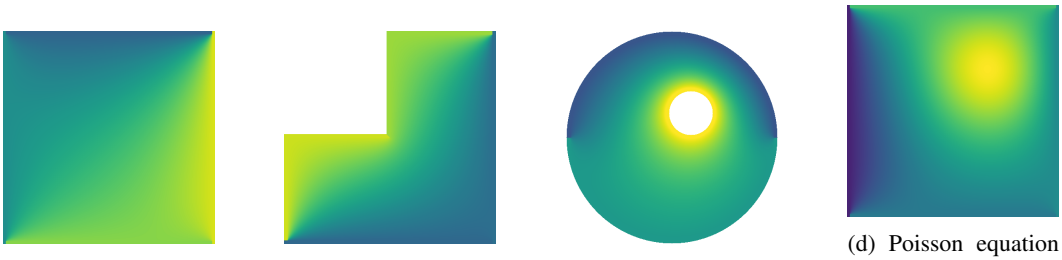
4 Experiments

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 dimensions than the training domain, for example, test on 256×256 grid 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 dimensions, (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 set f to be a 2D Gaussian function . Examples of the 4 settings are shown in Figure 1.



(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.

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

4.2 Evaluation

As discussed in Section 2.3, 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.

The Jacobi iterator and our model can both be efficiently implemented as convolutional layers on the GPU. Thus, we can set the computation cost as the number of convolutional layers. However, one might argue that on CPU, each Jacobi iteration only requires 4 operations, while a 3×3 convolutional kernel requires 9 operations. In the CPU case, a fair metric is the number of flops.

Clearly, number of flops is a more challenging metric for our model. For example, for a Conv1 model comparing against Jacobi, each iteration requires twice as many layers, but requires $(4+9)/4$ times as many flops. We report both metrics in our experiments.

4.3 Conv Model

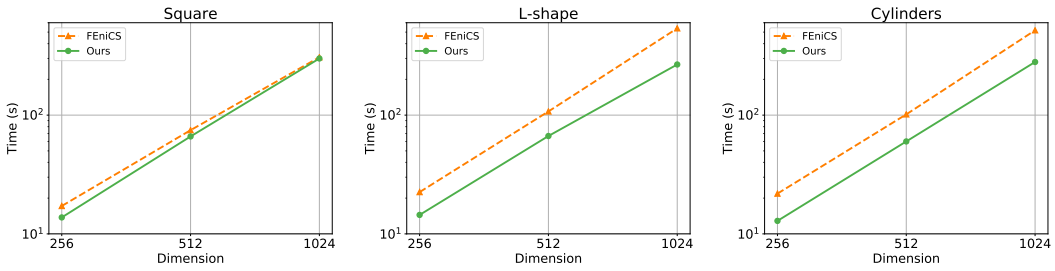
Table 1 shows results of the Conv model. The model is trained on 16×16 square domain, and the test dimensions are 64×64. For all settings, our models converge to the correct solution, and require less computation than Jacobi. The best model, Conv3, is $\sim 5\times$ faster than Jacobi in terms of layers,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



(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.

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 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 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 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.

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

Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations*, volume 95. Springer, 1994.

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.

Rainer Kress. *Numerical analysis, volume 181 of Graduate Texts in Mathematics*. Springer-Verlag, London, UK, 1998.

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

Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

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.

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.

A PROOFS

Proof of Theorem 1. Let u^* be a stationary point, i.e. $u^* = Tu^* + b$. For any initialization u^0 and $u^k = \Psi^k(u^0)$, the error $e^k = u^* - u^k$ satisfies

$$Te^k = (Tu^* + b) - (Tu^k + b) = u^* - u^{k+1} = e^{k+1} \Rightarrow e^k = T^k e^0 \tag{13}$$

Since $\rho(T) < 1$, we know $T^k \to 0$ as $k \to \infty$, which means the error $e^k \to 0$. Therefore, Ψ converges to u^* from any u^0.

□

Proof of Theorem 2. Observe that

$$\Phi(u) = Tu + c + H(Tu + c - u) = (T + HT - H)u + Hc + c \tag{14}$$

From Thereom 1, an H is valid if and only if $\rho(T + HT - H) < 1$. Because the spectral norm ρ is convex, and $T + HT - H$ is linear in H, so $\rho(T + HT - H)$ is convex in H as well. Thus, the set of valid H must be convex because it is a sub-level set of the convex function $\rho(T + HT - H)$.

To prove that it is open, observe that ρ is a continuous function, so $\rho(T + H(T - I))$ is continuous in H. Therefore if for some H we have $\rho(T + H(T - I)) < 1$, there exists sufficiently small $\epsilon > 0$, for any $\Delta H \in \mathbb{R}^{n^2 \times n^2}$ with $\|\Delta H\|_2 \leq \epsilon$,

$$\rho(T + (H + \Delta H)(T - I)) < 1 \tag{15}$$

This means that $H + \Delta H$ is valid for all ΔH such that $\|\Delta H\|_2 \leq \epsilon$. Therefore, the set of valid H is open under $\|\cdot\|_2$. In addition all norms are topologically equivalent in finite dimensions so the conclusion is independent of choice of norm. □

Proof of Theorem 3. From Eq. (14), Φ converges to a stable fixed point if $\rho(T + HT - H) < 1$. In addition, if $\rho(T + HT - H) \geq 1$, then after k iterations we have

$$u^k = (T + HT - H)^k u^0 + \cdots$$

where $\cdots$ are terms that do not contain u^0. Therefore it cannot converge if u^0 has non-zero components on eigenvectors of $T + HT - H$ with eigenvalues ≥ 1. Such vectors have measure 1 under Borel measure. This means that with probability 1 an initialization sampled from a continuous distribution does not converge. Therefore, if Φ converges, then with probability 1 we must have $\rho(T + HT - H) < 1$. This proves that it is valid. □