LEARNING NEURAL PDE SOLVERS WITH CONVERGENCE GUARANTEES
ICLR REPRODUCIBILITY CHALLENGE 2019

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper is part of the ICLR Reproducibility Challenge 2019. We tried to replicate the results and algorithm for training an iterative partial differential equation solver by interpreting the solver as a linear convolutional neural network and optimizing the weights of the convolutional kernels. We can replicate the results of the original paper Anonymous (2019)[1], obtaining a general solver, which generalizes well to a wide variety of geometries and boundary conditions, while achieving high speed ups compared to the baseline solver and guaranteeing convergence.

1 INTRODUCTION

Partial differential equations (PDEs) are differential equations which contain a-priori unkown multivariable functions and their partial derivatives. They are used to model various physical phenomena, such as heat, fluid dynamics or quantum mechanics. Traditionally PDEs are solved, meaning that a function is found, either by a specific hand-crafted or generic approach which iteratively updates the solution until convergence is reached. Among them a common numerical method is the finite-difference method (FDM), which approaches the differential equation by discretizing the problem space and converting the PDE to a system of linear equations, which can be solved using linear algebra.

The paper takes the idea of using machine learning in order to find high performing update rules instead of designing them by hand from (Anonymous, 2019, equation 8). The goal is to find a high performing, general solver while guaranteeing convergence. In order to fulfill these requirements the learned solver is an adapted existing standard solver, which is guaranteed to converge. The learned solver thus inherits the convergence properties from the base solver. We stress that the goal is not to find a new solver, but to optimize a pre-existing one. To be precise the learned part operates with the residuals after applying the standard solver. Thus convergence is guarantees. This construction further allows application to existing linear iterative solvers on general PDE problems.

This new solver is trained on a single, simple set of problem instances, and then tested on different geometries and boundary conditions with no observable loss of performance; generalization is thus reached. The learned solver is constructed by guaranteeing that a fixed point of the original solver is a fixed point in the learned solver as well.

Since a linear PDE solver can be expressed as a product of convolutional operations, it is not far fetched to use the similar techniques used in deep learning in order to find such an optimal operator. In order to test this approach a solver was trained on a 2D Poisson equation with a square-shaped Dirichlet boundary conditions. This solver is then tested on larger geometries of two shapes and different boundary values.

For general information we kindly refer to the original paper Anonymous (2019).

[1]We based our this report on the version from the 10 October 2018. We noted no differences to the newest version from the 23 November 2018 with regard to this report.

2 BACKGROUND

In this section, we give a short introduction to the Poisson problem and iterative solvers, which will help to understand the justification of using a convolutional neural network to obtain a solver.

2.1 POISSON EQUATION

The Poisson equation is a second order linear partial differential equation (PDE). In order to guarantee the existence and uniqueness appropriate boundary conditions needs to be prescribed [Gilbarg & Trudinger (2001)]. In this work we only consider Dirichlet boundary conditions. The Poisson problem hence reads:

$$Find \quad u\colon \overline{\Omega} = \Omega \cup \partial\Omega \to \mathbb{R} \quad s.t \begin{cases} \nabla^2 u = \sum_i \frac{\partial^2}{\partial x_i^2} = f(\boldsymbol{x}) & in\ \Omega \\ u = b(\boldsymbol{x}) & on\ \partial\Omega \end{cases} \tag{1}$$

Where $\Omega \subset \mathbb{R}^k$ is a bounded domain with boundary $\partial\Omega$. More specifically we consider $\overline{\Omega} = [0,1]^2$.

2.2 FINITE DIFFERENCE METHOD

In order to solve complex, real-world PDEs a numerical approach must be used, as analytic solutions can seldomly be found. As a first step the problem is discretized by transforming the solution space from $u\colon \mathbb{R}^k \to \mathbb{R}$ to $u_h\colon \mathbb{D}^k \to \mathbb{R}$, where $\mathbb{D}$ is a discrete subset of $\mathbb{R}$. In this paper $k = 2$ and denoting by N the domain size, we introduce a regular grid $\overline{\Omega}_h \subset \mathbb{D}^k$ on $\overline{\Omega}$:

$$\begin{aligned} \overline{\Omega}_h &= \{\boldsymbol{x}_{i,j} = (ih, jh) \qquad i,j = 0, ..., N-1\} \\ \Omega_h &= \{\boldsymbol{x}_{i,j} = (ih, jh) \qquad i,j = 1, ..., N-2\} \\ \partial\Omega_h &= \overline{\Omega}_h \setminus \Omega_h \end{aligned}$$

with $h = 1/N$ and denoting by the Ω_h interior points, and by $\partial\Omega_h$ the boundary points. Equation 1 can be approximated as follows, discretizing and approximating ∇^2:

$$Find \quad u_h\colon \overline{\Omega}_h \to \mathbb{R} \quad s.t. \begin{cases} \frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) = f_{i,j} & in\ \Omega_h \\ u_{i,j} = b_{i,j} & in\ \partial\Omega_h \end{cases} \tag{2}$$

It can be shown that the discrete approximation in equation 2 is stable and that $||u - u_h||_{L^2} \leq ch^2$ with c being a constant [Thomas (1995)]. Introducing a matrix $\boldsymbol{A} \in \mathbb{R}^{N^2 \times N^2}$ and a vector $\boldsymbol{f} \in \mathbb{R}^{N^2}$ problem defined in equation 2 can be written as a linear system:

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f} \tag{3}$$

From Problem definition 2 $\boldsymbol{A}$ can be written as a pentadiagonal matrix:

$$A_{i,j} = \begin{cases} 1 & \text{if } i = j \\ -\frac{1}{4} & \text{if } j \in \{i \pm 1, i \pm N\} \\ 0 & \text{else} \end{cases}$$

and defining $i^\star = \lfloor i/N \rfloor$, $j^\star = i \mod N$ we have:

$$f_i = h^2 f(\boldsymbol{x}_{i^\star, j^\star})$$

In order to prescribe the boundary conditions we introduce a reset operator $\mathcal{G}$:

$$\mathcal{G}(\boldsymbol{u}, \boldsymbol{b}) = \boldsymbol{G}\boldsymbol{u} + (\boldsymbol{I} - \boldsymbol{G})\boldsymbol{b} \tag{4}$$

where $\boldsymbol{G} \in \mathbb{R}^{N^2 \times N^2}$ is a diagonal matrix and $\boldsymbol{b} \in \mathbb{R}^{N^2}$ is the boundary values vector:

$$\begin{aligned} G_{i,i} &= 1, \quad b_i = 0 && \boldsymbol{x}_{i^\star,j^\star} \in \Omega_h \\ G_{i,i} &= 0, \quad b_i = b(\boldsymbol{x}_{i^\star,j^\star}) && \boldsymbol{x}_{i^\star,j^\star} \in \partial\Omega_h \end{aligned}$$

We note that the proposed approach to enforce boundary conditions is restricted iterative methods to solve the linear system 3 Moreover we have not investigated how this approach can be generalized to other type of boundary conditions other than Dirichlet or to different iterative methods such as the Gauss-Seidel method.

2.3 Iterative Solvers

A linear iterative solver can be expressed as:

$$\boldsymbol{u}^{k+1} = \boldsymbol{T}\boldsymbol{u}^k + \boldsymbol{c} \tag{5}$$

Where $\boldsymbol{T}$ is a constant update matrix and $\boldsymbol{c}$ is a constant vector. A common approach to build $\boldsymbol{T}$ and $\boldsymbol{c}$ is to split $\boldsymbol{A}$ into $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$ and by rewriting $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f}$ as $\boldsymbol{M} = \boldsymbol{N}\boldsymbol{u} + \boldsymbol{f}$ the following updating rule naturally arises:

$$\boldsymbol{u}^{k+1} = \boldsymbol{M}^{-1}\boldsymbol{N}\boldsymbol{u}^k + \boldsymbol{M}^{-1}\boldsymbol{f}$$

For more details we refer readers to LeVeque (2007) or to Anonymous (2019).

2.3.1 Jacobi method

Setting $\boldsymbol{M} = diag(\boldsymbol{A})$ leads to the so called Jacobi method. In the case of the Poisson problem $\boldsymbol{M} = \boldsymbol{I}$ hence relying on the previously introduced reset operator the Jacobi method reads:

$$\begin{aligned} \boldsymbol{u}^{k+1} &= \Psi(\boldsymbol{u}^k) \\ &= \mathcal{G}((\boldsymbol{I} - \boldsymbol{A})\boldsymbol{u}^k + \boldsymbol{f}, \boldsymbol{b}) \\ &= \boldsymbol{G}((\boldsymbol{I} - \boldsymbol{A})\boldsymbol{u}^k + \boldsymbol{f}) + (\boldsymbol{I} - \boldsymbol{G})\boldsymbol{b} \\ &= \boldsymbol{G}((\boldsymbol{I} - \boldsymbol{A})\boldsymbol{u}^k + \boldsymbol{f} - \boldsymbol{b}) + \boldsymbol{b} \end{aligned}$$

The Jacobi method can also be implemented by convolution and point-wise operations, as we explain in the following. We define by $\omega_J * \underline{\boldsymbol{u}}$ the 2D convolution with zero padding of the kernel ω_J and $\underline{\boldsymbol{u}} \in \mathbb{R}^{N\times N}$, with:

$$\omega_J = \begin{pmatrix} 0 & 1/4 & 0 \\ 1/4 & 0 & 1/4 \\ 0 & 1/4 & 0 \end{pmatrix} \tag{6}$$

We can also define a new reset operator $\underline{\mathcal{G}}$ denoting by $\circ$ the Hadamard product:

$$\underline{\mathcal{G}}(\underline{\boldsymbol{u}}, \underline{\boldsymbol{b}}) = \underline{\boldsymbol{G}} \circ \underline{\boldsymbol{u}} + \underline{\boldsymbol{b}} \tag{7}$$

where $\underline{\boldsymbol{G}}, \underline{\boldsymbol{b}} \in \mathbb{R}^{N\times N}$:

$$\begin{aligned} \underline{G}_{i,j} &= 1, \quad \underline{b}_{i,j} = 0 && \boldsymbol{x}_{i,j} \in \Omega_h \\ \underline{G}_{i,j} &= 0, \quad \underline{b}_{i,j} = b(\boldsymbol{x}_{i,j}) && \boldsymbol{x}_{i,j} \in \partial\Omega_h \end{aligned}$$

Finally the Jacobi method can be written as

$$\begin{aligned} \underline{\boldsymbol{u}}^{k+1} &= \underline{\Psi}(\underline{\boldsymbol{u}}^k) \\ &= \underline{\mathcal{G}}(\omega_J * \underline{\boldsymbol{u}}^k + \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}}) \\ &= \underline{\boldsymbol{G}} \circ (\omega_J * \underline{\boldsymbol{u}}^k + \underline{\boldsymbol{f}}) + \underline{\boldsymbol{b}} \end{aligned}$$

3 LEARNING PROCESS

We want to find an operator $\mathcal{H}$ to optimize the convergence of the Jacobi method for the Poisson problem of the form:

$$\underline{\boldsymbol{u}}^{k+1} = \underline{\Phi}^{k}_{\mathcal{H}} \tag{8}$$

$$= \underline{\Psi}(\underline{\boldsymbol{u}}^{k}) + \mathcal{H}(\underline{\Psi}(\underline{\boldsymbol{u}}^{k}) - \underline{\boldsymbol{u}}^{k}) \tag{9}$$

We define $\mathcal{H}$ as the composition of K operations:

$$\mathcal{H}(\underline{\boldsymbol{w}}) = \mathcal{H}_K ... (\mathcal{H}_3(\mathcal{H}_2(\mathcal{H}_1(\underline{\boldsymbol{w}})))) ...)$$
$$\mathcal{H}_i(\underline{\boldsymbol{w}}) = \underline{\boldsymbol{G}} \circ (\omega_i * \underline{\boldsymbol{w}})$$

As in the Jacobi method $\omega_i * \underline{\boldsymbol{w}}$ represents a 2D convolution with zero padding and no bias term of a 3×3 kernel ω_i with $\underline{\boldsymbol{w}}$ whereas the Hadamard product with $\underline{\boldsymbol{G}}$ ensures that the residuals are always zero at the boundary points.

3.1 INTERPRETATION OF $\mathcal{H}$

The operator $\mathcal{H}$ can also be expressed as a matrix vector multiplication. We call $\boldsymbol{H} \in \mathbb{R}^{N^2 \times N^2}$ the equivalent matrix:

$$\boldsymbol{H} = \boldsymbol{G}\boldsymbol{H}_K \boldsymbol{G}\boldsymbol{H}_{K-1} ... \boldsymbol{G}\boldsymbol{H}_1 \tag{10}$$

$\boldsymbol{H}_i$ is a banded matrix which is obtained from the corresponding 3×3 kernel ω_i as follows:

$$\begin{aligned} H_{i,i-N-1} &= \omega_{0,0} & H_{i,i-N} &= \omega_{0,1} & H_{i,i-N+1} &= \omega_{0,2} \\ H_{i,i-1} &= \omega_{1,0} & H_{i,i} &= \omega_{1,1} & H_{i,i+1} &= \omega_{1,2} \\ H_{i,i+N-1} &= \omega_{2,0} & H_{i,i+N} &= \omega_{2,1} & H_{i,i+N+1} &= \omega_{2,2} \end{aligned}$$

So the new method can be written using only matrix multiplications as:

$$\boldsymbol{u}^{k+1} = \Phi_{\boldsymbol{H}}(\boldsymbol{u}^k) = \Psi(\boldsymbol{u}^k) + \boldsymbol{H}(\Psi(\boldsymbol{u}^k) - \boldsymbol{u}^k)$$

This interpretation is useful because if the following holds:

$$\rho(\boldsymbol{G}\boldsymbol{T} + \boldsymbol{H}(\boldsymbol{G}\boldsymbol{T} - \boldsymbol{I})) < 1 \tag{11}$$

then the method is guaranteed to convergence to a fixed point. Which can be used during training time to enforce the convergence requirement.

4 TRAINING AND GENERALIZATION

4.1 TRAINING

In order to find the optimal operation $\boldsymbol{H}(\boldsymbol{u})$ a linear deep neural network of multiple convolutional layers with kernel size 3 and without any bias terms or activation function is created, trained on a small grid with a square boundary, e.g. a 16×16 grid, and tested on larger grids and different boundary shapes. The optimization objective is defined as:

$$\min_{\mathcal{H}} \sum_{\underline{\boldsymbol{G}}, \underline{\boldsymbol{b}}, \underline{\boldsymbol{f}} \in \mathcal{D}; k \in \mathcal{DU}(1,20)} \left\| \underline{\Phi}^{k}_{\mathcal{H}}(\underline{\boldsymbol{u}}^0, \underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}}) - \underline{\boldsymbol{u}}^{\star}(\underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}}) \right\|_2^2 \tag{12}$$

Intuitively: A network $\mathcal{H}$ is found by minimizing the error after a fixed k iterations of the learned solver $\phi_{\mathcal{H}}$ with the network $\mathcal{H}$. With $k \in \mathcal{DU}(1, 20)$ we denote the sampling of k from a discrete uniform distribution with interval $[1, 20]$. $\underline{\boldsymbol{u}}^0$ is sampled from a Gaussian distribution: $\underline{\boldsymbol{u}}^0 \sim \mathcal{N}(0, 1)$. We have not enforced the any constraint to guarantee that the obtained operator $\underline{\Phi}_{\mathcal{H}}$ converges to a fixed point. Since it is not possible to express analytically the spectral radius in Inequality 11 it is not clear how a regularization term could be added to the objective function. A possible solution would be to check the spectral radius at each iteration and if > 1 under-relax the weights of the convolutional kernels. However this technique is highly computationally expensive since it requires to compute the eigenvalues of a $N^2 \times N^2$ matrix at each iteration. Anyway even without explicitly enforcing the optimization yields an operator $\underline{\Phi}_{\mathcal{H}}$ which indeed converges for the tested problems.

Two different shapes were chosen: Square and L-shaped. The L-shaped domain is created by removing a smaller square from one of the edges. Each side exhibits a different but constant boundary value chosen from a uniform distribution on the interval [-1, 1]. Thus an L-shaped domain has 6 different boundary values.

4.1.1 Optimizer

We are using Adadelta as the optimizer of our model, because of its ability to adapt over time and its minimal computational overhead. The method requires no manual adjustment of a learning rate and is robust to various selection of hyperparameters. Adadelta adjusts the learning rate by slowing down learning around a local optima, when the accuracy changes by a small margin. Adadelta also uses the idea of momentum to accelerate progress along dimensions in which the gradient consistently point in the same direction. This idea is implementing by keeping track of the previous parameter update and applying an exponential decay with a decay factor of $\rho = 0.9$ Zeiler (2012).

Table 1: Mandatory function results for primary model.

domain size $\lvert\boldsymbol{D}\rvert$	50
batch size $\lvert\boldsymbol{B}\rvert$	10
max epochs	1000
Tolerance	1e-6
Optimizer	Adadelta
ρ	0.9

The training was done with *batch optimization* of size $|\mathcal{B}| = 30$. A random sample of this size was chosen from the problem instances. The loss for these batches is defined as the sum over all losses in the batch. The pseudo code for our training process is given in Algorithm 1.

4.2 Hyper Parameter Search

In order to find the optimal number of layers and learning rate a simple grid search is performed. As a first step we fix the number of layers $K = 3$ and compare the loss evolution for different learning rates γ From Figure 1 it is evident the the loss decay is highly dependent on the choice of the learning rate. For small γs the loss tends to converge to what probably is a local minimum while for high values it can lead to divergence problems, note that in Figure 1 the loss for $\gamma = 1e - 4$ is not displayed since the optimization diverged.

We hence decided to use the Adadelta optimization method for its ability to adapt to the specific problem. We report in Table 1 the parameters used for the training with The number of layers K chosen was from 1 to 5.

5 Experiments & Results

The hypothesis of the original paper is that a general solver can be found by training on simple domains. The simplest Laplace equation $\nabla^2 u = 0$ on a square boundary shape was therefore chosen as training data. For results we refer to table **??** and figure 1. The model was trained on a

Parameter: ConvNet $\mathcal{H}$
Data : $\underline{\boldsymbol{G}}$,$\underline{\boldsymbol{b}}$,$\underline{\boldsymbol{f}}$
Result : Optimal ConvNet $\mathcal{H}$

for $\{\underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}}\} \in D$ **do**
  Compute $\underline{\boldsymbol{u}}^{\star}(\underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}})$
  Randomly sample k_i from $\mathcal{DU}(1, 20)$
  Sample $\underline{\boldsymbol{u}}^0$ from a Gaussian with $\mu = 0$ and $\sigma = 1$
end
repeat
  batch $\leftarrow$ randomly sample a subset of $\mathcal{D}$
  $\text{loss}_{batch} \leftarrow \sum_{p \in batch} \left\| \underline{\Phi}^k_{\mathcal{H}}(p) - \underline{\boldsymbol{u}}^{\star}(p) \right\|_2^2$
  Compute the gradient of the loss function
  Update weights of $\mathcal{H}$
  $\text{loss}_{epoch} \leftarrow \sum_{p \in \mathcal{D}} \left\| \underline{\Phi}^k_{\mathcal{H}}(p) - \underline{\boldsymbol{u}}^{\star}(p) \right\|_2^2$
until $\|loss_{epoch-1} - loss_{epoch}\| < threshold$;

Algorithm 1: Training Process

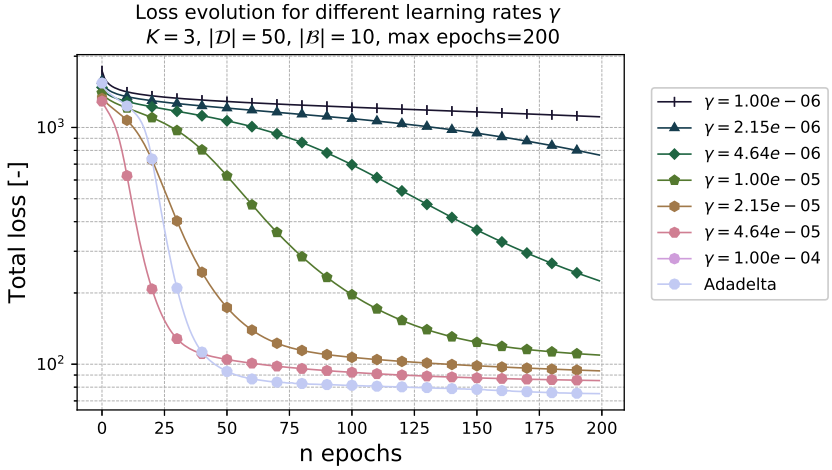

Figure 1: Loss evolution for different learning rates

16×16 grid and evaluated on grids of size 32×32 and 64×64 for both the square and the L-shaped domain. See Figure 4 for an example solution.

In Figure 3 we shows how the error w.r.t the ground truth solution evolves with the number of iterations for $K = \{1, 2, 3, 4, 5\}$ and the Jacobi method. However we need better metrics in order to fairly compare the different models.

Both solvers were evaluated on three metrics: the number of iterations, ratio of FLOPS and ratio of CPU-time until required tolerance is reached. The number of flops were calculated assuming both solvers would be implemented using convolutional operators. This results in 4 *multiply-add* operations for each element in the grid for the Jacobi iteration, whereas the learned solver exhibits 9 *multiply-add* operations. This is the same measurement as reported in the original paper, which is an estimation of the FLOPS taken. In addition to the paper we measured the CPU-time. Which deemed us to be a less error-prone and more reliable measure, nevertheless both ratios gave comparable results.

As can be seen in table 2 the trained solver was considerably faster than the existent solver, showing a much quicker conversion than the baseline model. Thus replicating the given results in the original paper. It is interesting to note that while the ratio of flops worsens for the learned solver, the CPU

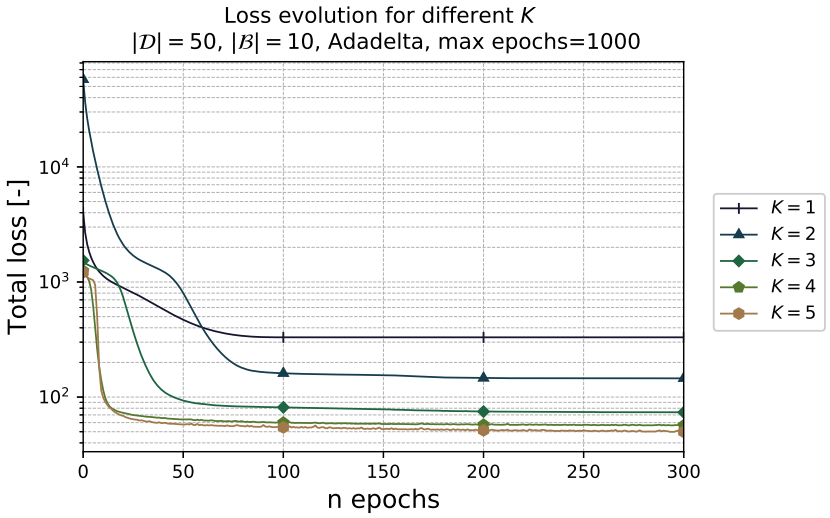

Figure 2: Loss evolution for different number of layers K

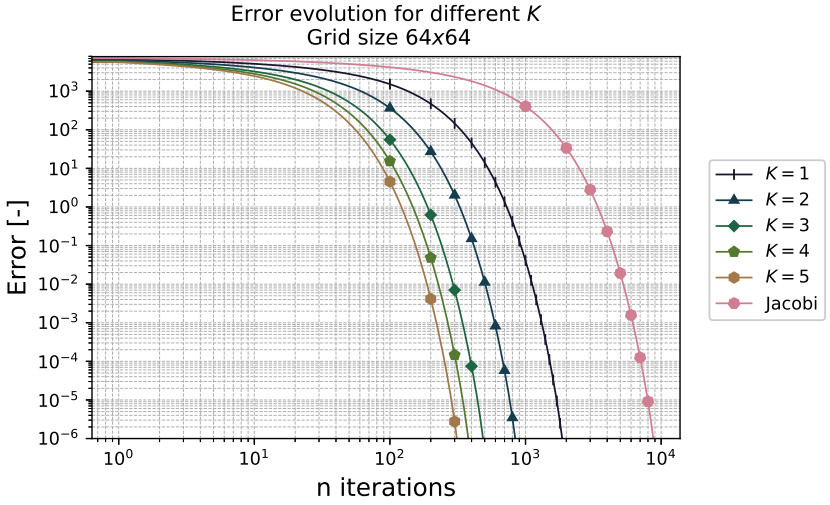

Figure 3: Error evolution w.r.t. solver iterations for different number of layers K

time ratio increases, which corresponds to the increased complexity of the model. The results show a clear signal, that the learned solver outperforms the standard one.

6 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. This is the reason why our work was focused on reproducing the results of Anonymous (2019), and on empirically proving the generalization of their model to arbitrary shapes and grid sizes. For more information on the the theoretical aspects of our paper, we refer to the original paper.

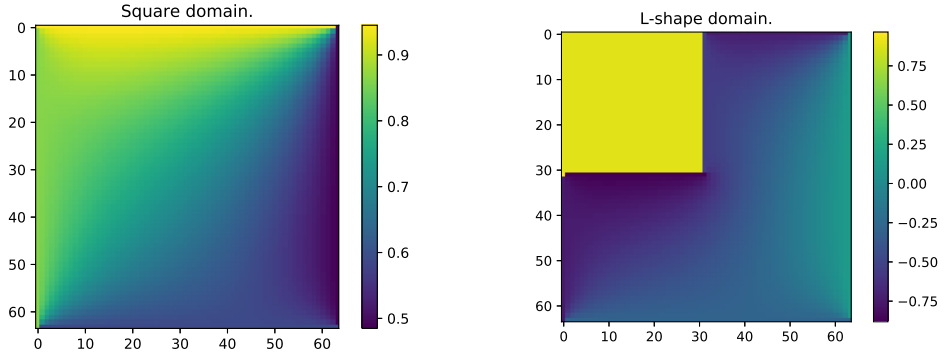

Figure 4: Example solutions for the two domains.

Table 2: Test results of solver trained on a 16×16 grid with Adadelta as optimizer.

# layers	grid size	shape	FLOPS ratio	CPU time ratio	# iterations existent solver	# iterations trained solver
1	32	l-shape	0.17	0.34	1978.15	439.15
	32	square	0.17	0.33	1970.50	437.40
	64	l-shape	0.17	0.34	1978.15	439.15
	64	square	0.17	0.36	8768.60	1937.30
2	32	l-shape	0.30	0.17	1978.15	196.20
	32	square	0.30	0.18	1970.50	195.55
	64	l-shape	0.30	0.17	1978.15	196.20
	64	square	0.30	0.19	8768.60	868.60
3	32	l-shape	0.39	0.12	1978.15	113.30
	32	square	0.39	0.11	1970.50	112.80
	64	l-shape	0.39	0.12	1978.15	113.30
	64	square	0.39	0.11	8768.60	504.50
4	32	l-shape	0.50	0.09	1959.04	82.42
	32	square	0.51	0.09	1960.46	82.56
	64	l-shape	0.50	0.09	1959.04	82.42
	64	square	0.51	0.10	8745.48	372.78

7 Conclusion & Future work

We could partially confirm the results reported in the original paper, not every result was reproducible either through lacking time or lack of certainty in how these results where achieved or measured. The trained solver was able to generalize well to the presented different sizes, geometries and boundary values, while using less resources compared to the standard solver.

In the future work we would like to improve the design of the solver and the experiments in order to gain more confidence in the presented approach. For example $\mathcal{H}$ is fixed for each iteration, one could imagine a solver with different $\mathcal{H}$ for different iterations up to a certain threshold. We did not have the opportunity to test the solver using the MultiGrid method, nor the square-Poisson problem. It is not clear how the cylinder domain was implemented in a finite difference framework, whether radial coordinates or a non uniform grid were used.

We estimate that investigating how this approach can be generalized to other type of boundary conditions other than Dirichlet or to different iterative methods such as the Gauss-Seidel method would lead to interesting results and a more applicable approach in general, as well as trying to solve different PDEs.

We find that counting the number of flops might not be the best measure of comparing the two models in this case, because the convolutional network can be implemented much more efficiently on a GPU, and might take fewer flops to converge to the ground truth solution.

REFERENCES

Anonymous. Learning neural pde solvers with convergence guarantees. 2019. URL https://openreview.net/forum?id=rklaWn0qK7. under review.

David Gilbarg and Neil Trudinger. *Elliptic Partial Differential Equations of Second Order*. Springer, 2001. URL https://www.jstor.org/stable/2132758.

Randall LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-state and Time-dependent Problems*. 2007.

J.W. Thomas. *Numeric Partial Differential Equations: Finite Difference Methods*. Springer, 1995. URL https://www.springer.com/book/9780387979991.

Mathew D. Zeiler. Adadelta: An adaptive learning rate method. 2012. URL https://arxiv.org/pdf/1212.5701.pdf.