UNDER REVIEW

RESCIENCE C

ICLR Reproducibility Challenge 2019 / Machine Learning

[Re] Learning Neural PDE Solvers with Convergence Guarantees

Francesco Bardi[1], Samuel von Baussnern[1], Emiljano Gjiriti[1],

[1] École polytechnique fédérale de Lausanne (EPFL), Lausanne, Switzerland

Edited by
Koustuv Sinha

Reviewed by
Anonymous
Anonymous

Received
10 April 2019

Published
–

DOI
–

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. There are several numerical methods to solve PDEs. A common one 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. The obtained linear system can be solved using an iterative procedure which updates the solution until convergence is reached.
The original paper proposes to use machine learning techniques in order to find high performing update rules instead of designing them by hand [1], while still guaranteeing convergence. In order to fulfill these requirements the learned solver is an adapted existing standard solver, from which the convergence property is inherited by enforcing that a fixed point of the original solver is a fixed point for the trained solver as well. We stress that the goal is not to find a new solver, but to optimize an existing one. To be precise the learned part operates with the residuals after applying the standard solver. This construction allows application to other existing linear iterative solvers of equivalent design.
Since a linear iterative 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 to solve a 2D Poisson equation on a square-shaped geometry with Dirichlet boundary conditions. This solver is then tested on larger geometries of two shapes and different boundary values. No significant loss of performance was observed; generalization is thus reached.
For more information we kindly refer to the original paper [1].

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.

Correspondence should be addressed to Francesco Bardi (francesco.bardi@epfl.ch)
The authors have declared that no competing interests exists.
Code is available at https://github.com/francescobardi/pde_solver_deep_learned.

2.1 Poisson Equation

The Poisson equation is a second order linear partial differential equation (PDE). In order to guarantee the existence and uniqueness of a solution, appropriate boundary conditions needs to be prescribed [[2]]. In this paper only Dirichlet boundary conditions were considered. 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 be seldom 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 $\mathbb{D}^k$ is a discrete subset of $\mathbb{R}^k$. 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 $h = 1/(N-1)$ and denoting by Ω_h the 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 ([3]). Introducing a matrix $\boldsymbol{A} \in \mathbb{R}^{N^2 \times N^2}$ and a vector $\boldsymbol{f} \in \mathbb{R}^{N^2}$ problem definition 2 can be written as a linear system:

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

With $\boldsymbol{A}$ being a pentadiagonal matrix:

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

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

$$f_i = \frac{h^2}{4} f(\boldsymbol{x}_{i^\star, j^\star})$$

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

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

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 to iterative methods solving linear systems equivalent to equation 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 finds the solution of a linear system by iteratively updating an initial solution guess $\boldsymbol{u}^0$. The updating step can be expressed as:

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

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 $\boldsymbol{M}\boldsymbol{u} = \boldsymbol{N}\boldsymbol{u} + \boldsymbol{f}$ 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 [4] or to [1].

Jacobi method – Setting $\boldsymbol{M} = diag(\boldsymbol{A})$ leads to the so called Jacobi method. In the case of the Poisson problem $\boldsymbol{M} = \boldsymbol{I}$ and $\boldsymbol{T} = \boldsymbol{I} - \boldsymbol{A}$, 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:

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

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

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

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:

$$\begin{aligned}\underline{\boldsymbol{u}}^{k+1} &= \underline{\Phi}_{\mathcal{H}}(\boldsymbol{u}^k) \\ &= \underline{\Psi}(\underline{\boldsymbol{u}}^k) + \mathcal{H}(\underline{\Psi}(\underline{\boldsymbol{u}}^k) - \underline{\boldsymbol{u}}^k)\end{aligned}$$

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

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

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 $\underline{\boldsymbol{w}}$. The operation 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:

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

$\boldsymbol{H}_i$ 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:

$$\rho(\boldsymbol{G}\boldsymbol{T} + \boldsymbol{H}(\boldsymbol{G}\boldsymbol{T} - \boldsymbol{I})) < 1 \tag{4}$$

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 operator $\mathcal{H}$ the corresponding linear neural network is created. Each 2D convolutional layer has a kernel size 3×3 and zero bias, without any activation function. The training phase is done on a set of Poisson problem instances $\mathcal{D}$. A problem instance is uniquely defined by $\underline{\boldsymbol{G}}$, $\underline{\boldsymbol{f}}$, and $\underline{\boldsymbol{b}}$. We set $\boldsymbol{f} = 0$ and we use a square domain with a 16×16 grid. Each side exhibits a different but constant boundary value chosen from a uniform distribution on the interval [-1, 1]. For each problem instance the error between the ground truth solution $\underline{\boldsymbol{u}}^\star(\underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}})$ and the computed solution using $\underline{\Phi}_{\mathcal{H}}$ with

k iterations contributes to the loss function. The ground truth solution is obtained using the Jacobi method operator $\underline{\Psi}$ with a sufficiently high number of iterations $k = 2000$. The optimization objective is then defined as:

$$\min_{\mathcal{H}} \sum_{\underline{\boldsymbol{G}},\underline{\boldsymbol{b}},\underline{\boldsymbol{f}} \in \mathcal{D}; k \in \mathcal{DU}(1,20)} \left\| \underline{\Phi}^k_{\mathcal{H}}(\underline{\boldsymbol{u}}^0, \underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}}) - \underline{\boldsymbol{u}}^\star(\underline{\boldsymbol{G}}, \underline{\boldsymbol{f}}, \underline{\boldsymbol{b}}) \right\|_2^2 \tag{5}$$

With $k \in \mathcal{DU}(1, 20)$ we denote the sampling of k from a discrete uniform distribution on the interval $[1, 20]$. The initial guess $\underline{\boldsymbol{u}}^0$ 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 4 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 ω_i. 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. We showed however that empirically, without explicitly enforcing this constraint, the optimization yields an operator $\underline{\Phi}_{\mathcal{H}}$ which indeed converges for the tested problems.

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 implemented by keeping track of the previous parameter update and applying an exponential decay with a decay factor of $\rho = 0.9$ ([5]).

Table 1. Parameters used in the training process.

Grid size $N \times N$	16×16
Number of problems $\lvert\mathcal{D}\rvert$	50
Batch size $\lvert\mathcal{B}\rvert$	10
Max epochs	1000
Tolerance	1e-6
Optimizer	Adadelta
ρ	0.9

The training was done with batch optimization of size $|\mathcal{B}| = 10$. At each epoch the set of problem instances $\mathcal{D}$ is randomly split in $\lceil |\mathcal{D}|/|\mathcal{B}| \rceil$ subsets. 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 ??.

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 γ small 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.

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
  $\mathcal{D}^\star \leftarrow$ randomly split $\mathcal{D}$ in $\lceil |\mathcal{D}|/|\mathcal{B}| \rceil$ subsets
  for $\mathcal{B} \in \mathcal{D}^\star$ do
    $\text{loss}_{batch} \leftarrow \sum_{p\in\mathcal{B}} \left\| \underline{\Phi}^k_{\mathcal{H}}(p) - \underline{\boldsymbol{u}}^\star(p) \right\|_2^2$
    Compute the gradient of the loss function
    Update weights of $\mathcal{H}$
  end
  $\text{loss}_{epoch} \leftarrow \sum_{p\in\mathcal{D}} \left\| \underline{\Phi}^k_{\mathcal{H}}(p) - \underline{\boldsymbol{u}}^\star(p) \right\|_2^2$
until $\|loss_{epoch-1} - loss_{epoch}\| < Tolerance$;

Algorithm 1: Training Process

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 process. The number of layers K chosen was from 1 to 5. Figure 2 compares the loss evolution for the different models. It is evident that the improvement on the total loss at convergence diminishes with K increasing, in particular it seems that there is not a substantial difference when $k > 3$.

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. The model was trained on 16×16 grid, and evaluated on grids of size 32×32 and 64×64 for both a square and an L-shaped domain. 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. The ground truth solution is obtained using the Jacobi method with a sufficient number of iterations $k = 5 \times 10^4$. 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 k for the obtained solvers ($K = \{1, 2, 3, 4, 5\}$) and the Jacobi method. The learned solvers clearly outperform 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 solvers exhibit $4 + 9K$ 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.

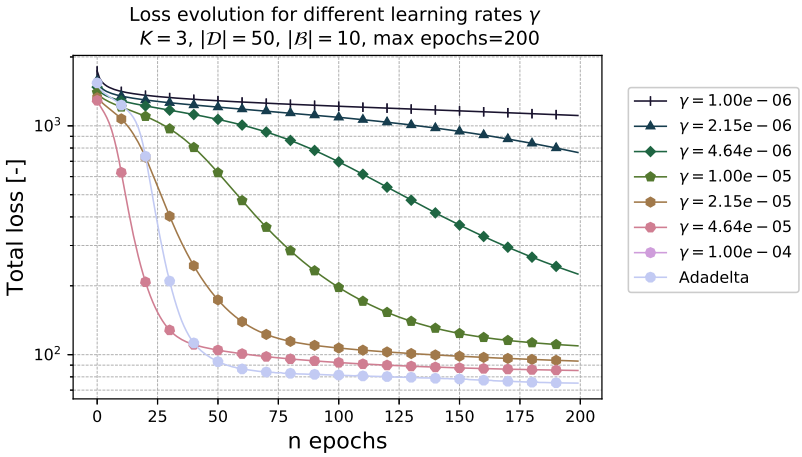

Figure 1. Loss evolution for different learning rates

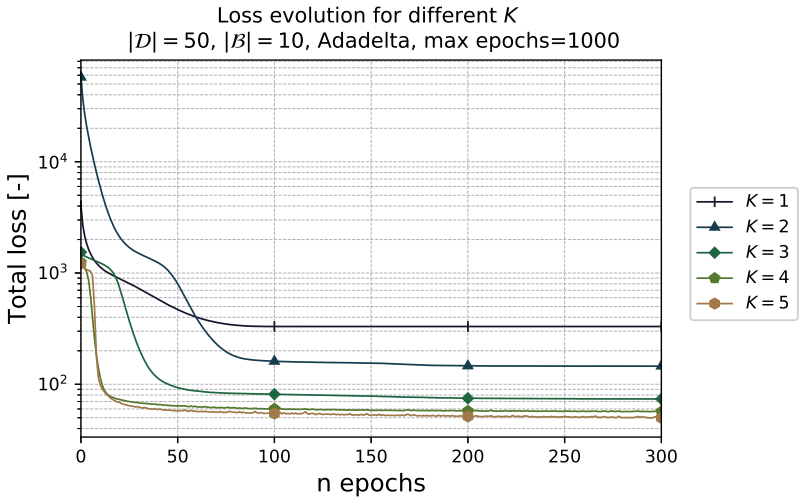

Figure 2. Loss evolution for different number of layers K

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. The highest speed-up is achieved by the 5-layer network.

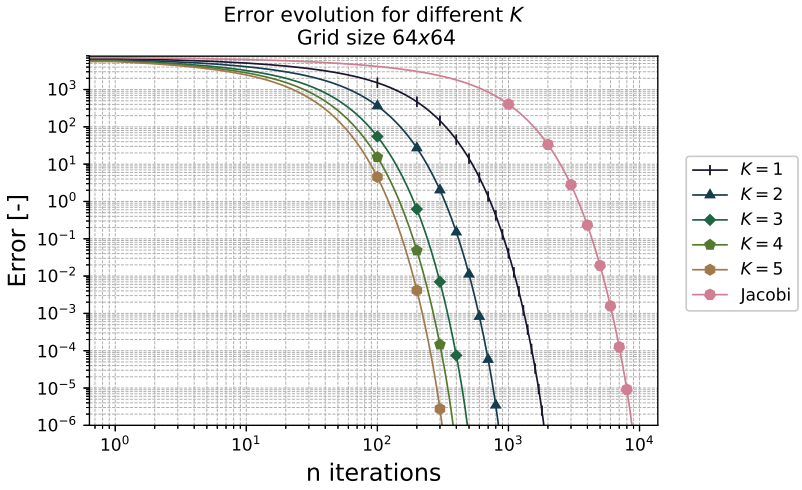

Figure 3. Error evolution w.r.t. solver iterations for different number of layers K

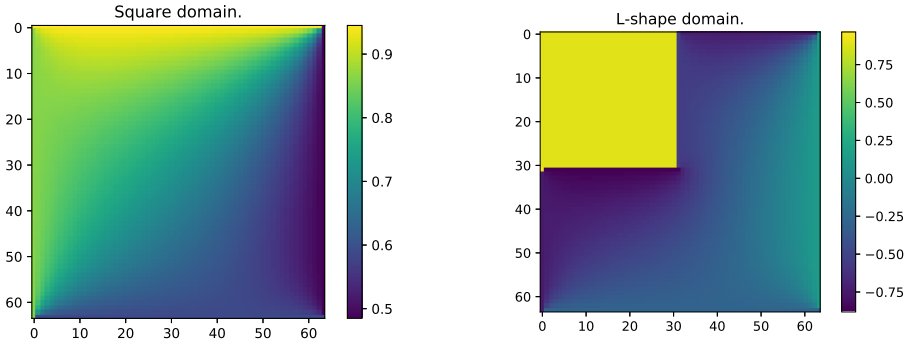

Figure 4. Example solutions for the two domains.

Table 2. Test results of solver trained on a 16×16 grid with Adadelta as optimizer. Sample size for each test of 20, number of iterations was counted until a difference to the ground truth solution of $1e-6$ or a convergence difference of $1e-12$ were achieved. Shown are the mean and the standard deviation.

			ratios: $\frac{\text{trained solver}}{\text{existent solver}}$		
K	Grid size	Geometry	FLOPS	CPU time [s]	#iterations k
1	32	l_shape	0.688 ± 0.001	0.308 ± 0.064	0.212 ± 0.000
		square	0.688 ± 0.000	0.332 ± 0.097	0.212 ± 0.000
	64	l_shape	0.689 ± 0.001	0.351 ± 0.053	0.212 ± 0.000
		square	0.686 ± 0.011	0.327 ± 0.025	0.211 ± 0.003
2	32	l_shape	0.518 ± 0.001	0.161 ± 0.032	0.094 ± 0.000
		square	0.518 ± 0.001	0.171 ± 0.051	0.094 ± 0.000
	64	l_shape	0.521 ± 0.001	0.165 ± 0.026	0.095 ± 0.000
		square	0.519 ± 0.008	0.183 ± 0.031	0.094 ± 0.001
3	32	l_shape	0.421 ± 0.001	0.106 ± 0.012	0.054 ± 0.000
		square	0.421 ± 0.001	0.101 ± 0.013	0.054 ± 0.000
	64	l_shape	0.426 ± 0.001	0.115 ± 0.017	0.055 ± 0.000
		square	0.425 ± 0.007	0.116 ± 0.019	0.055 ± 0.001
4	32	l_shape	0.401 ± 0.002	0.109 ± 0.028	0.040 ± 0.000
		square	0.401 ± 0.002	0.095 ± 0.022	0.040 ± 0.000
	64	l_shape	0.408 ± 0.001	0.097 ± 0.013	0.041 ± 0.000
		square	0.407 ± 0.007	0.098 ± 0.016	0.041 ± 0.001
5	32	l_shape	0.402 ± 0.002	0.078 ± 0.016	0.033 ± 0.000
		square	0.402 ± 0.002	0.083 ± 0.016	0.033 ± 0.000
	64	l_shape	0.412 ± 0.001	0.088 ± 0.012	0.034 ± 0.000
		square	0.410 ± 0.007	0.091 ± 0.023	0.033 ± 0.001

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 [1], and on empirically proving the generalization of their model to arbitrary shapes and grid sizes.

7 Conclusion & Future work

We could partially confirm the results reported in the original paper, not every result was reproducible either through lack of time or certainty in how these results were 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.

References

1. J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon. “Learning Neural PDE Solvers with Convergence Guarantees.” In: International Conference on Learning Representations. 2019.
2. D. Gilbarg and N. Trudinger. Elliptic Partial Differential Equations of Second Order. Springer, 2001.
3. J. Thomas. Numeric Partial Differential Equations: Finite Difference Methods. Springer, 1995.
4. R. LeVeque. Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-state and Time-dependent Problems. 2007.
5. M. D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method.” In: CoRR abs/1212.5701 (2012). arXiv: 1212.5701.