Deep Learning Applications on Solving PDEs

Chengze Shen
chengzes@andrew.cmu.edu

Senyu Tong
senyut@andrew.cmu.edu

Abstract

In this project, we present two deep learning approaches to solve first and second order Poisson equations with Dirichlet boundary conditions.

1 Introduction

Numerical or analytical solutions to different partial differential equations (PDEs) are required for a lot of fields including physics, mathematics, and most fields of engineering. There are existing traditional tools that can solve given PDEs by discretizing and solving in a domain (1). The results those tools output suffer from limited differentialbility, so in seek of a more general and efficient method, in this project, we present two deep learning methods, utilizing traditional multi-layer perceptron (MLP) and convolutional neural network (CNN), to solve first and second order PDEs. Our main focus in this project is second order Poisson equation with Dirichlet boundary conditions. Formally, given an equation with form:

$$G(x, u(x), \nabla u(x), \nabla^2 u(x)) = 0, \forall x \in D \tag{1}$$

and its boundary condition

$$u(x) = \hat{u}(x), \forall x \in \partial_d D \tag{2}$$

where $\partial_d D$ is portion of ∂D for essential boundary condition. We feed the discretized data points drwan from the domain into the network and train it, and output a diffrentiable, closed-form solution u_t to this PDE by

$$u_t(\vec{x}) = A(\vec{x}) + F(\vec{x})N(\vec{x}, \vec{p})$$

where $N(\vec{x}, \vec{p})$ was the output of the network, $A(\vec{x})$ and $F(\vec{x})$ are both designed to satisfies the boundary conditions for the trial solution in domain D. We believe the methods we provide could serve as baselines before applying more computationally heavy taksks, and could give engineers and mathematicians with potentially more efficient ways to calculate numerical results to PDEs.

2 Related Work

As the field of deep learning is thriving, we see only some research of deep learning applications on solving PDEs, focusing on very simple neural network structures (2). I. E. Lagaris *et al* present ANN, Artificial Neural Networks for Solving Differential Equations. This system preforms well on 2D domain, but with higher dimension, the number of parameters explodes and the model performance was unsatisfactory (1). Chiaramonte *et al.* applied one-layer feed-forward neural network to solve Laplace equation and conservation law by finding a trial solution that minimizes related error index within a given domain D (3). One group of students from Stanford University applied multi-layer neural network with Generative-Adversarial Network (GAN) style to some toy PDE examples (4), with the success on training the model to fit well-behaved PDEs. These research piqued our interests in exploring potential modifications to the existing architectures used, and finding novel applications of other deep learning architectures that can be applied to solving PDEs.

10807 Class Project (2019).

3 Background

As a baseline, we applied naive one-layer MLP without regularization or dropout to solve 2D Laplace equation $\nabla^2\Psi(x) = 0, \forall x \in D$. With $D = [0,1] \times [0,1]$, and boundary condition

$$u(x) = 0, \forall x \in \{(x_1, x_2) \in \partial D | x_1 = 0, x_1 = 1, or x_2 = 0\}$$

$$u(x) = sin\pi x_1, \forall x \in \{(x_1, x_2) \in \partial D | x_2 = 1\}$$

The trial solution contained numerical results to data points within domain $\hat{D}$ that were close to

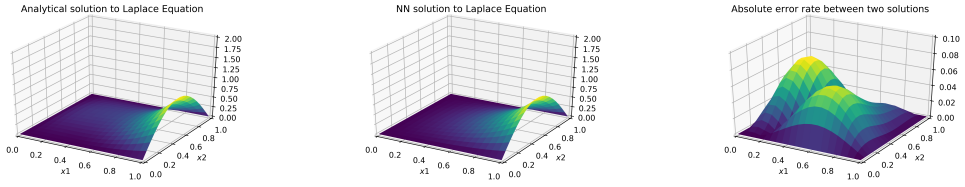

Figure 1: a-c. Midway results on solving Laplace equation. **a**, Analytical solution; **b**, MLP solution; **c**, absolute error between the two solutions.

For the attempt to utilize CNN architecture to solving PDEs, we apply filter W to the discretized domain $\hat{D}$ to capture the gradient features in small regions. We implemented CNN architecture on first order PDE to test out its validity. Formally, given equations of form:

$$G(x, u(x), \nabla u(x)) = 0, \forall x \in D \tag{3}$$

and the similar boundary condition, we solve the whole region $\hat{D}$ at once by approximating gradients at each position. This is a brand new attempt to apply CNN architecture to solve PDEs, and we name our proposed structure Gradient Approximate CNN (GACNN).

4 Methods

4.1 MLP

We apply Cybenko Theorem (5), stating that the feed-forward network can approximate continuous functions on compact subsets of R^n.
As stated in the introduction section, we try to output a closed-form solution that makes use of a trained neural network. Given a 2D Poisson equation and its Dirichlet boundary conditions,

$$\nabla^2 u(x, y) = f(x, y)$$

$$u(0, y) = f_0(y),\ u(1, y) = f_1(y)$$

$$u(x, 0) = g_0(x),\ u(x, 1) = g_1(x)$$

, we choose $A(x, y)$ to satisfies the bondary conditions, by

$$A(x,y) = (1-x)f_0(y)+xf_1(y)+(1-y)(g_0(x)-[(1-x)g_0(0)+xg_0(1)])+y(g_1(x)-[(1-x)g_1(0)+xg_1(1)])$$

And we set $F(x) = x(1-x)y(1-y)$ to make sure neural network output won't affect the boundary conditions.
Then we can write our trial solution as $u_t(x, y) = A(x, y) + F(x)N(x, y, \vec{w})$, where N is the neural network and $\vec{w}$ is the parameter learned in the neural network.
To train the network $N(x, y, \vec{w})$, we feed $\hat{D} = \{x^{(i)} \in D; i = 1, 2, ..m\}$, the discretized domain of D, to be the input layer of the neural network, and minimize

$$J(p) = \sum_{all\ x} G(x, u(x), \nabla u(x), \nabla^2 u(x))^2 \tag{4}$$

to learn the optimal p^* s.t. $\arg\ min_p J(p)$. For the simple example of Laplace equation in R^2,

$$G(x, u(x), \nabla u(x), \nabla^2 u(x)) = \frac{\partial^2 \Psi_t(x)}{\partial x_1^2} + \frac{\partial^2 \Psi_t(x)}{\partial x_2^2} \tag{5}$$

Then the backpropagation can be performed to update the model.
In our baseline, we only form the problem to solve the Laplace function, and now we extend the baseline one-layer neural network to include the options of adjusting hidden layers, number of neurons, momentum, weight decay and dropouts to see if there are any improvements with accuracy and convergence speed. For the input layer, we also further discretized the domain into two different sets, train and test sets, to check for the loss.

4.2 GACNN

Gradient approximation convolutional neural network (GACNN) is our attempt to apply CNN architecture to solve PDEs. The discretized domain $\hat{D}$ is passed in as input, and convolution operation is performed as $Conv(\hat{D}, f)$ to mimic the extraction of gradients within a small region. We trained two networks Net_X, Net_Y, each one trained for gradients in one dimension (input dimension $\in R^2$). The loss function is defined by the first order PDEs relationships of the target functions, and both Net_X, Net_Y gradients will be updated accordingly. For the simple example of a first order PDE

$$3\frac{\partial \Psi}{\partial x} + \frac{\partial \Psi}{\partial y} = 0$$

we have Net_X trained for $\frac{\partial \Psi}{\partial X}$ and Net_Y trained for $\frac{\partial \Psi}{\partial Y}$. The final prediction is made based on some Dirichlet boundary conditions $\Psi(X, a)$ or $\Psi(b, Y)$ and gradients at each position.

5 Results

5.1 MLP for Laplace Equation

Feed forward neural network with 2 hidden layers, each with 64 nodes was used for training Laplace Equation $\in R^2$. Learning rate = 0.001, batch size = 32. For the dropout the rate is 0.5. MSE is used as the loss function.

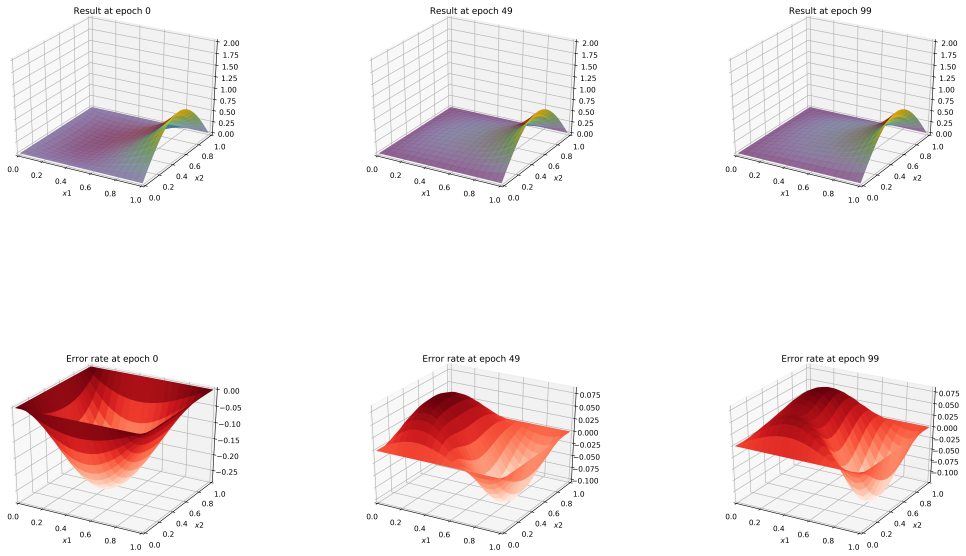

Figure 2: a-f: **Top (a-c)**: Results for neural network and analytical solution. **Bottom (d-f):** Error between the two types of results.

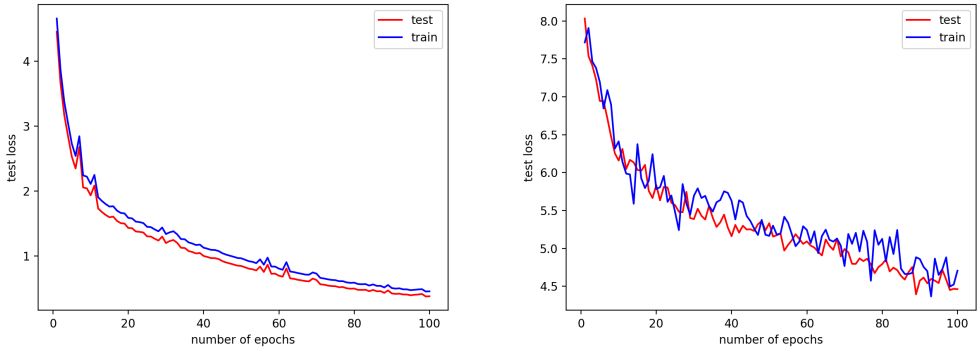

Figure 3: a-b. Loss vs. epochs for training on Laplace Equation. **a**, without dropout; **b**, with dropout.

We saw a quick convergence of the NN solutions towards the analytical solutions and a smooth loss curve with consistent decreases (Figure 3a). The test data were refined mesh grid points within domain D and they showed same consistency as of the training data. The error graphs showed that there were persistent regions where the NN solutions could not achieve the accuracy as of the rest region (Figure 2d-f). Overall the NN solutions were consistent to the analytical solutions within region D (Figure 2a-c). Using the same network architecture with dropout resulted in noisy learning and the slow decrease in the loss (Figure 3b). We also find that adding momentum and weight decay do not contribute much in this problem in terms of convergence speed and accuracy.

5.2 MLP for higher dimension problem

We apply our NN approach to higher dimension. We test on a 3D Poisson equation

$$\nabla^2 u(x, y, z) = -3 * \pi^2 * sin(\pi x) * sin(\pi y) * sin(\pi z)$$

with zero boundary condition. The exact solution is given by

$$u(x, y, z) = sin(\pi x) sin(\pi y) sin(\pi z)$$

, and we get $A(x) = 0$. As dimension increase, we suffer from the exponential growth of input parameters. From the domain $[0, 1]^3$ we still draw 20 points from each $[0, 1]$ interval and get 8000 discrete point.
To measure the accuracy, we compute the L2 norm of the residue, $||u - u_t||_2$. Figure 4 shows the results running on a single layer network with batch size 32, and figure 5 shows the results running on a 2 layer network with weight decay $1 - 10^{-9}$ and momentum 0.05.

5.3 GACNN examples

Run naive GACNN on Laplace Equation, in which we solved the minimization with respect to ground truth calculated from analytical solution (Figure 6). The results were expected, that within 5000 epochs the model converged closely to the ground truth.
We further ran some toy first order PDE example on our standard GACNN, with equation

$$\frac{3\partial \Psi}{\partial x} + \frac{\partial \Psi}{\partial y} = 0, \Psi(x, 0) = xe^{-x^2}, x, y \in [0, 1] \times [0, 1] \tag{6}$$

in which we want to estimate $\Psi(x, y)$. Learning rate = 0.0001, momentum = 0.5. One layer CNN connects to a one layer fully connected neural network. L1 loss is used as the loss function.
The boundary conditions for the toy example was accurately achieved by GACNN. However, the overall shape did not resemble the original function (Figure 7a-d). The only parts that the GACNN model approximated with high accuracy were near both 0 and 1 of y (x_2 in the graph). This is off from our expectation, since we expected the model to achieve near MLP performance with correct implementation. The loss vs. epoch reflected that the learning was quick at the start, but it converged and the model won't be improved after certain point (Figure 8).

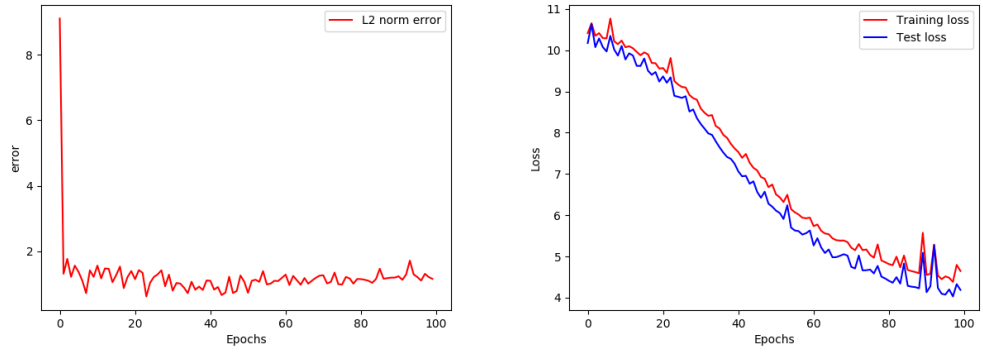

Figure 4: a-b. accuracy and loss for training on 3D problem with a single layer. **a**, the accuracy; **b**, the loss.

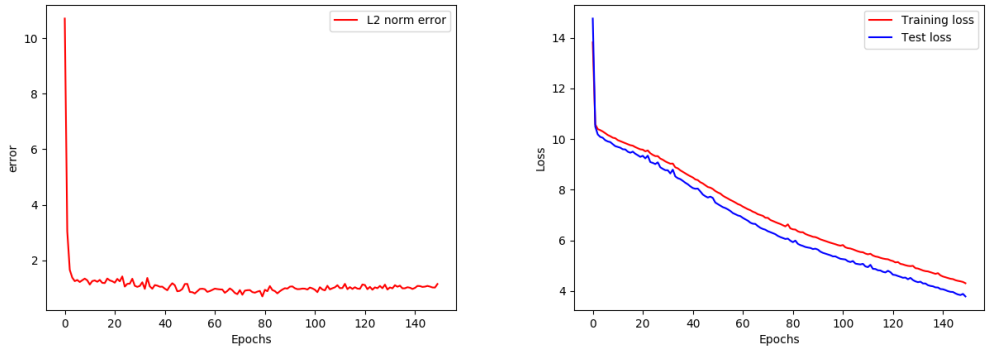

Figure 5: a-b. accuracy and loss for training on 3D problem with multiple layers. **a**, the accuracy; **b**, the loss.

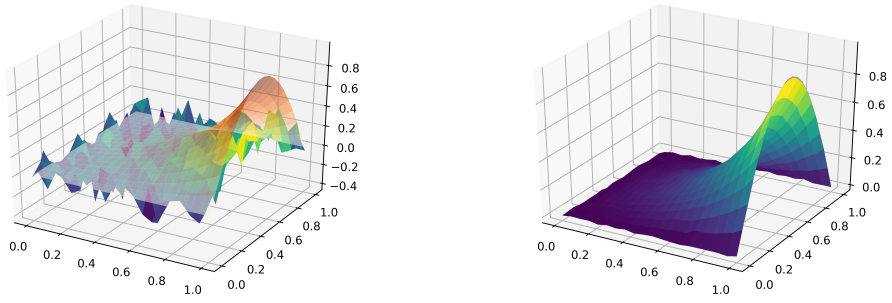

Figure 6: a-b. GACNN solution on Laplace equation. **a**: at epoch 1000, **b**: at epoch 5000

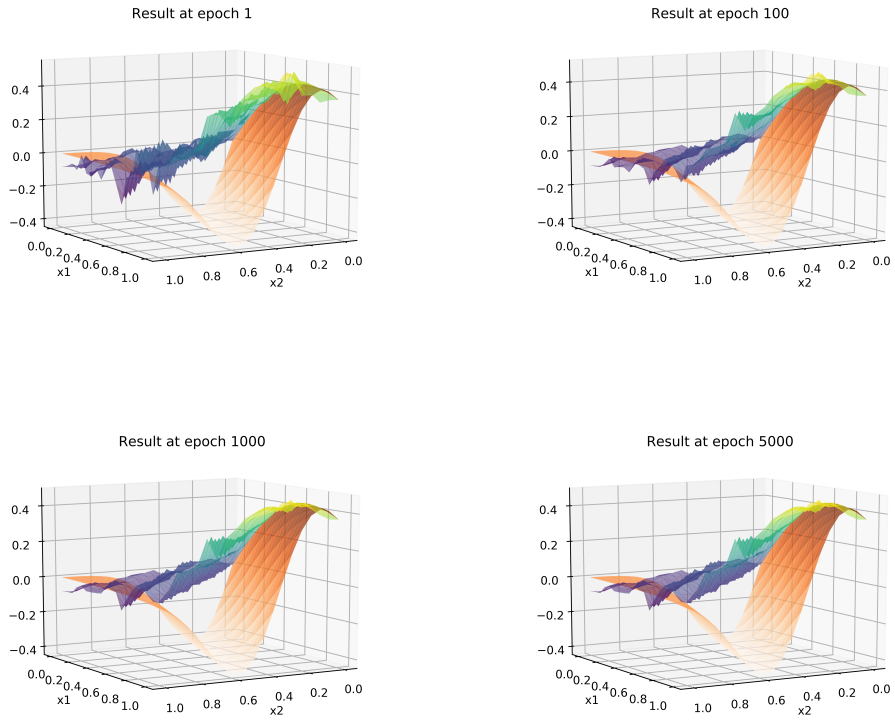

Figure 7: a-d. GACNN results comparing to the analytical solutions at different epochs. The orange surface is the analytical solution; the green-blue surface is the GACNN solution.

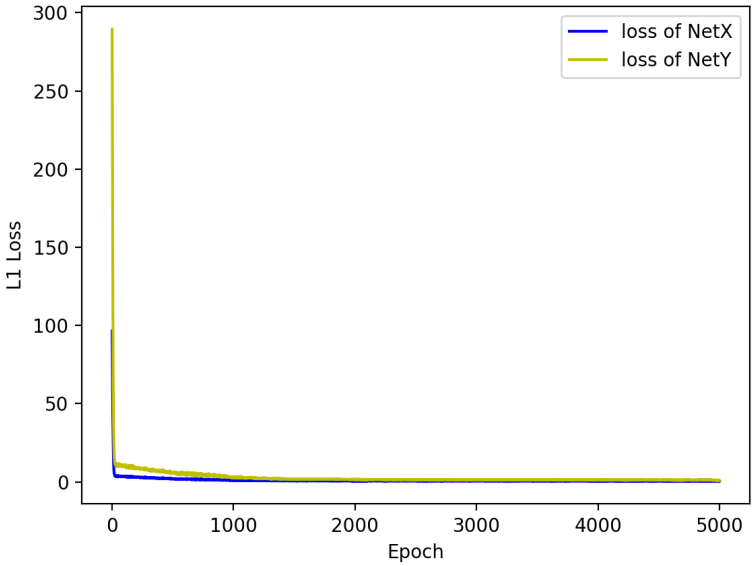

Figure 8: GACNN training loss on the toy example from equation (6).

6 Discussion and Analysis

6.1 MLP

Extending from the baseline of Chiaramonte *et al*'s one layer network, we have checked the performance of the algorithm on other Poisson equations, and find out that this approach performs well if we limited the domain to 2D. From section 5.1, for the simple Laplace equation, we find that increasing the number of layers won't have a significant improvement on the final prediction accuracy, and in addition, adding dropouts to the model worsened the performance.

As shown in section 5.2, if we increase the dimension of the domain, we find that increasing the layer and adding regularization term make both the loss graph and accuracy graph more smooth, but there is no significant improvement regarding the convergence rate. Again, if we add dropouts, the result becomes worse.

We then proposed a hypothesis–that all the neurons are important for making the near optimal prediction, since they are responsible for transferring the information about the gradient computation, and hence removing any node will lead to loss of information and worsen the performance.

For dimension higher than 3D, we find the computational power needed for training the network becomes exceptionally high, and the loss converges extremely slowly. So we have the incentives to deploy CNN, in the sense that instead of computing the gradient on each single point in the domain, we split the domain to relatively larger subspaces, and get the gradient change in a whole neighboring region to avoid the explosion of parameters. But its clear to see that this method won't be as accurate as the MLP method, and is useful only for higher dimensions to get a sense of the gradient flows in a particular area.

6.2 CNN

We also came up with a new model utilizing CNN architecture to solve some specific PDEs. We built the GACNN model and tested its performance on first order PDEs, and it was under-performing. This might be the result of several potential issues: 1) the loss function did not utilize the boundary conditions in the correct way, leading to ambiguity for gradients; 2) the predictions did not use the trained gradients correctly, leading to errors in the function calculation. These issues will be naturally addressed in the future work on the topics.

6.3 Limitations

There are several limitations to our models. For the MLP model, it is only able to reliably train on low-dimensional data; higher dimension leads to exponential increase in training time. In addition, it might not be intuitive at the first sight to choose the most fitted boundary functions. For the GACNN model, we only was able to construct the model to train on first order PDEs. For second order and other PDEs, it might be tricky to find the correct way to update the gradient at each data point. Nevertheless, the GACNN approach is the first attempt we saw to utilize CNN architecture to solve PDEs and it is very intuitive to train for gradients within the target region. We believe it to have potentials to become a solution at least as good as the MLP model.

Bibliography

[1] I. E. Lagaris, A. Likas and D. I. Fotiadis. *Artificial Neural Networks for Solving Ordinary and Partial Differential Equations*. Physics, 1997.

[2] Alexandr Honchar. *Neural Networks for Solving Differential Equations*. Media, May 2017.

[3] M. M. Chiaramonte and M. Kiener. *Solving Differential Equations Using Neural Networks*. cs229 Stanford University, 2013.

[4] Kailai Xu, Bella Shi, Shuyi Yin. *Deep Learning for Partial Differential Equations (PDEs)*. cs230 Stanford University, 2018.

[5] G. Cybenko. *Approximation by Superpositions of a Sigmoidal Function*. Math. Control Signals Systems, 1989, 2:303-314.

[6] J.David Logan *Applied Partial Differential Equations*.

[7] Modjtaba Baymani, Asghar Kerayechian, Sohrab Effati. *Artificial Neural Networks Approach for Solving Stokes Problem*. Applied Mathematics, 2010, 1, 288-292.