# Mesh-free Neural PDE Solvers: Deep Ritz Method and Deep Galerkin Method

**Aman Jalan**
Department of Mechanical Engineering
Massachusetts Institute of Technology
`ajalan@mit.edu`

## Abstract

Traditional numerical techniques employ discretization schemes to obtain solutions of partial differential equations which are often sensitive to mesh parameters and are computationally expensive. Recent developments and successes in deep learning techniques have catapulted a new field of research in mesh-free PDE solvers. These solvers are mesh independent by virtue of generating solutions and optimizing themselves using gradient descent on a predefined loss function, thus obviating the need to feed in training data of any kind. While there do exist data-driven deep learning techniques, employing them often requires generating large amounts of training data by traditional numerical schemes which add significantly to the computational cost of using that method, besides necessitating additional optimization to get convergence guarantees that are mesh-independent for practical usage. In this project we look at two such mesh-free methods in detail, develop optimized implementations of them in Julia and showcase the accuracy of the code by benchmarking against known analytical solutions and results in their original publications `Report`.

## 1 Introduction

The idea of using neural networks to solve partial differential equations dates back to the 1990s. However, more active research has only started to surface in the past few years. These techniques can be broadly classified into two categories: data-driven or supervised learning techniques and generative methods. For the data-driven approach, a supervised learning framework is adopted in which data generated from other traditional numerical techniques. For the second class of approaches, solutions are generated at each training step of the model and the corresponding loss is calculated to determine the direction of the steepest fall in its value. Both these methods rely on the function approximation capabilities of neural networks.

A variety of architectures are often employed in either classes of methods. More recent works have focused on leveraging the generalization capabilities of complex network structures like convolutional long-short-term memory units, graphical neural networks and hypernets. [11] Introduces a ConvLSTM network which integrates with FDM/FVM schemes and solves PDEs with minimum simulation errors (against traditional solutions). It also proposes a training scheme that minimizes long-term error accumulation. [1] proposes a discretization learning ConvLSTM scheme that is capable of learning small scale dynamics from high resolution data, by learning equation-specific forms of the spatial derivative. For the loss function, the accuracy of the final time derivative is used instead of the spatial derivatives themselves.

With recent developments in scientific computing: automatic differentiation (AD) [2], the idea of defining partial differential equations utilizing AD, imposing physics through neural network losses together with easy access to backpropagation revolutionized PDE solving by using Physics- Informed

Neural Networks (PINN). Implementations of PINN on solving PDEs are investigated in [[9], [8], [7], [6]].

[12] proposes a CNN based architecture (similar to U-Net) that uses an encoder-decoder system to map input space to the approximate solution along with their mean and variances. [DEEP-O-NETS] Propose a DL scheme that uses stacked branch-nets to encode input functions and a trunk-net to encode spatial points to ultimately learn various operators (eg. integral, operator to solve Burger's equation etc.). It takes random instances of different functions as inputs. [5] - proposes a loss function based on the "variational form" of the PDE that is discretization-independent and highly parallelizable and demonstrates performance on AD-PDEs. Some methods have also dealt with boundary conditions in novel way. [13] proposes a novel data-driven NN architecture that handles Dirichlet BCs separately (decomposing into 4 Laplace problems) and solving the resulting Poisson homogeneous problem separately.

Traditional PDE solver technologies developed over the last few decades have primarily relied on solving discretized formulations of PDEs using methods such as, finite volume, finite element or finite difference. The exact or approximate forms of the linearized discrete equations are used, in combination with linear equation solvers, to improve the solutions iteratively. The discretization method allows access to higher order and advanced numerical approximations for partial derivatives which can be useful in resolving highly non-linear parts of the PDE solution and also add artificial dissipation to improve solver stability. Additionally, these schemes coupled with the iterative solution strategy have proved to be robust in terms of solver stability and convergence.

This project explores 2 deep learning PDE solvers in detail: Deep Ritz [3] and the Deep Galerkin Method [10]. These methods had been introduced in recent years and have become fixtures in the scientific machine learning community. Since there were no current implementations of these techniques in Julia to the best of the author's knowledge, this project aims to provide a ready-to-use optimized implementations of these models for certain classes of problems. With some changes, one can use this code to solve the differential equation of their choice and benchmark their own methods. This work thus aides in accelerated research while providing a first-hand understanding of these methods.

The rest of this paper contains 3 sections, with sections 2 and 3 discussing Deep Ritz and DGM respectively and the last section concluding the paper. Within each of the middle two sections, several sections explain each of these models in detail, including the mathematical foundations underpinning these models, problem formulation and network architecture employed. They then end with numerical experiments that demonstrate the efficacy of our Juila implementation.

## 2 Deep Ritz Method

### 2.1 The Ritz Method

In this section, we provide a basic overview of the The Ritz method [4]. This methods enables us to find an approximation to eigenvalue problems that are otherwise very difficult to solve analytically, if not impossible, by leveraging the idea of the principle of least action. This is very useful especially when it is used to approximate to solve partial differential equations in the function space with an objective function. For example, if a function $y(x)$ could be assumed to be approximated by a linear combination of independent functions:

$$y(x) = \varphi_0(x) + k_1\varphi_1(x) + ... + \varphi_N(x). \tag{1}$$

where $\varphi_i(x), i \in \{1, 2, ..., N\}$ span the function space of our interest. A common set of basis functions to use is

$$\begin{gathered} \cos x, \cos 2x, \cdots, \cos nx, \cdots \\ \sin x, \sin 2x, \cdots, \sin nx, \cdots \end{gathered} \tag{2}$$

In this case however, the function space is infinite-dimensional, which makes it very difficult to solve. This prompts us to reduce the dimensionality of the problem, which is where the Ritz method is very useful. Consider the Poisson equation with homogeneous Dirichlet boundary conditions:

$$\begin{cases} \Delta u = -f(x), & x \in \Omega, \\ u = 0, & x \in \partial\Omega. \end{cases} \tag{3}$$

The weak solution of (3) that corresponds with the principle of least action is

$$\begin{cases} \text{Find } \ u \in H_0(\Omega), \quad s.t. \\ I(u) = \min\limits_{v \in H_0(\Omega)} I(v), \end{cases} \tag{4}$$

where $H_0(\Omega)$ is the set of admissible functions. Using the variational method, one can prove that

$$I(v) = \int_\Omega \left( \frac{1}{2} |\nabla v(x)|^2 - f(x)v(x) \right) dx. \tag{5}$$

If it is assumed that $v$ is an n-dimensional function, then

$$\delta I(v) = \int_\Omega \left( \frac{\partial v}{\partial x_1} \cdot \delta \frac{\partial v}{\partial x_1} + ... + \frac{\partial v}{\partial x_n} \cdot \delta \frac{\partial v}{\partial x_n} - f(x)\delta v(x) \right) dx. \tag{6}$$

Employing the integration by parts method:

$$\int_\Omega \frac{\partial v}{\partial x_i} \cdot \delta \frac{\partial v}{\partial x_i} dx = \frac{\partial v}{\partial x_i} \cdot \delta v \Big|_{\partial\Omega} - \int_\Omega \delta v \cdot \frac{\partial^2 v}{\partial x_i^2} dx = - \int_\Omega \delta v \cdot \frac{\partial^2 v}{\partial x_i^2}. \qquad 1 \le i \le n \tag{7}$$

Substituting (7) into (6) we can obtain the following equation.

$$\delta I(v) = - \int_\Omega \delta v (\Delta v + f(x)) dx = 0 \tag{8}$$

Owing to the strong convexity of the function $I$ in (5), the solution of (4) is unique. Hence the solution of (4) is identical to (3).

## 2.2 Deep Ritz Method with neural networks

Using the Ritz Method mentioned above, it is natural to think of solving partial differential equations with deep neural networks. Considering the partial differential equation

$$\Delta u(x) + f(x) = 0, \qquad x \in \Omega. \tag{9}$$

According to Ritz Method, what we need to do is to find the minimum of $I$, which is

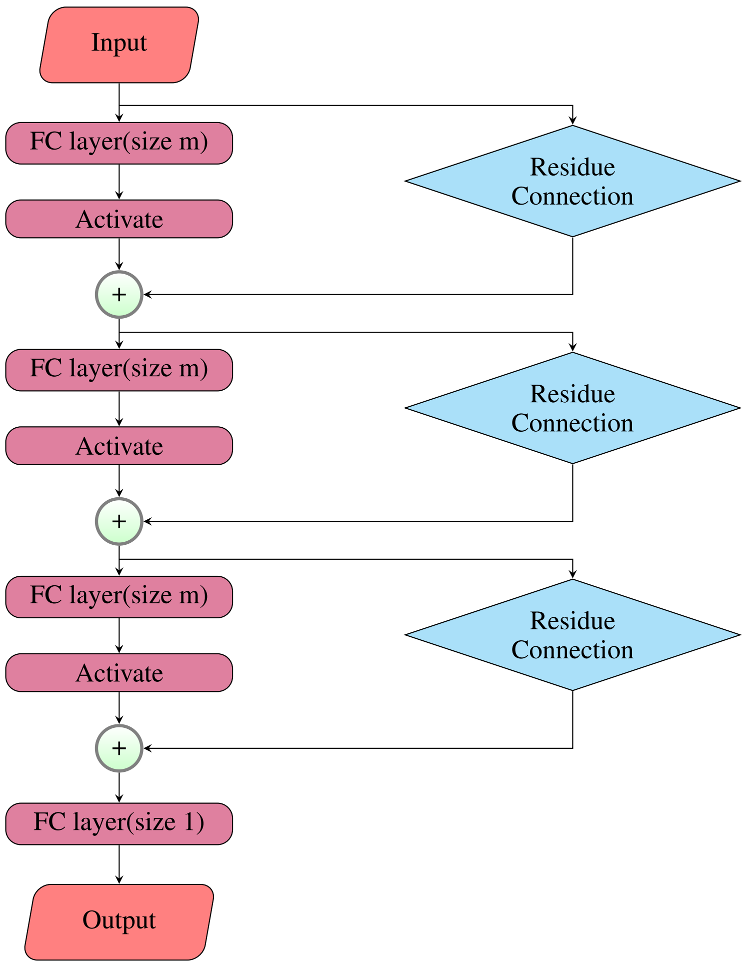
$$\min_{u \in H} I(u), \tag{10}$$

where

$$I(u) = \int_\Omega \left( \frac{1}{2} |\nabla u(x)|^2 - f(x)u(x) \right) dx, \tag{11}$$

and $H$ is the set of admissible function. Deep Ritz's main idea is to find an approximate solution by leveraging multi-layer neural networks' approximation property. The stochastic gradient descent algorithm is used to solve the optimization problem (10).

Figure 1: Deep Ritz Method Network Architecture



### 2.2.1 Building Trail Function

A nonlinear transformation $x \to u_\theta(x)$ defined by deep neural networks is used to approximate function $u(x)$. Here $\theta$ denotes network parameters in the model. Similar to the ResNet structure our network consists of several blocks, each block consisting of two linear transformations, two activation functions and one residual connection. The $i$-th block can be expressed as

$$s_i = \phi(W_{i2} \cdot \phi(W_{i1} \cdot s_{i_1} + b_{i1}) + b_{i2}) + s_{i-1}, \tag{12}$$

where $s_i$ is the output of the $i$-th layer, $W_{i1}, W_{i2} \in R^{m\times m}, b_{i1}, b_{i2} \in R^m$ and $\phi$ is the activation function.

Because our partial differential equation involves the Laplacian transform, we hope that the second derivative of the function $u(x)$ is not a constant. To ensure the smoothness of function $u_\theta$, we apply the non-linear function (13) as the activation function, instead of ReLU (rectified linear unit) function.

$$\phi(x) = \max\{x^3, 0\}. \tag{13}$$

The residual connection in [**?** ] helps to avoid gradient vanishing problems. After several blocks, we adopt a linear transform to the final result. The architecture of residual connection can be viewed in the figure below. And the whole network can be expressed as

$$u_\theta(x) = a \cdot f_n(x) \circ ... \circ f_1(x) + b, \tag{14}$$

where $f_i(x)$ is the $i$-th block and $a \in R^m, b \in R$. Note that the input vector $x$ is not necessarily $m$-dimensional. In order to handle this mismatch, we can pad $x$ with a vector of zeros. In our model, we always assume $d < m$.

After building our trial functions,we now need to minimize the $I(u)$ in (11)

### 2.2.2 Euler Numerical Integration Method

The first problem we need to deal with is to calculate the integral in (11) . For simplicity, we define:

$$g(x, \theta) = |\nabla u(x)|^2 - f(x)u(x), \tag{15}$$

then $I(u)$ can be expressed as

$$I(u) = \int_\Omega g(x, \theta) dx. \tag{16}$$

Obviously, it's impossible to calculate this integral directly. We use Euler integration method to approximate the integral.

$$I(u) = L(x, \theta) = \frac{1}{N} \sum_{i=1}^{N} g(x_i, \theta), \tag{17}$$

where $x_i$ is taken from the uniform grid on the domain with steps 0.001.

### 2.2.3 The Stochastic Gradient Descent Algorithm

During the training process of neural networks, stochastic gradient descent (SGD) is a commonly used method for optimization. In this problem, we also choose stochastic gradient descent method to minimize $I(u)$. This optimization process can be expressed as:

$$\theta^{k+1} = \theta^k - \eta \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} g(x_{i,k}, \theta^k), \tag{18}$$

where $\{x_{i,k}\}$ is the randomly selected from uniform grid points. In our settings, we use stochastic gradient descent algorithm with mini-batch and Adam [**?** ] optimizer to optimize.

## 2.3 Numerical Experiments

The model above was implemented in Julia from scratch to carry out numerical experiments and test its efficacy on 2 different classes of problems (shown in the next section). While each implementation is specific to the type of problem it solved, here are a few commonalities which were put in place to optimize the implementation:

- Activation functions and function which represented the right-hand side of the Poisson differential equation (hereby referred to as f(x)) were inlined to speed up all neural network computations (including boundary condition losses)
- @inbounds were used where-ever there was a possibility of an index bound-checking operation. This provided significant boosts to loss computations as they involved parsing through different regions in the problem domain which was set up as an array of random points.

### 2.3.1 Problem 1: 2D Poisson in a uniform square domain

$$\begin{cases} \Delta u = 1, & x \in \Omega, \\ u = 0, & x \in \partial\Omega, \end{cases} \tag{19}$$

where $\Omega = \{(x, y) \in (-1, 1) \times (-1, 1)\}$.

This is a standard case of the 2D Poisson equation. This was chosen as a first test case to check if our model implementation works as expected.

In order to implement the residual connections, two techniques were carried out and the faster one was chosen for the final submission. In the first method, a block (consisting of 2 dense layers and a residual connection) were implemented from scratch using a struct with explicitly defined weights and biases. This struct was further augmented by a function that produced an instance of this struct with the Xavier uniform initialization for the weights and zeros for biases and an overloaded call that defined the operation this struct performed on an input.

In the second method, the SkipConnection method in Flux was used to create the block structure. This method condensed the 50 lines of code for the former method down to just 1. It also turned out to be slightly faster than the from-scratch implementation and thus was chosen for all numerical experiments henceforth.

For calculating the inner loss (within the domain) in this problem, points were randomly sampled from a uniform distribution over the domain. Since the loss function required computing the magnitude of the gradient of the NN output, a finite difference quadrature rule was used. This quadrature was used as a result of poor performance of simple first order gradient approximations for this problem. The boundary condition losses were calculated in the same way as the inner losses. Here, concatenations were employed instead of using iterative techniques and array mutations to create an array of boundary points to significantly improve boundary-point sampling.

For this problem, the model was trained for 1000 epochs while sampling 2000 points for the inner loss computation and 1000 points (= 250 points on each boundary) for boundary loss. The following loss function was used:

$$I(u) = \int_{\Omega} \left( \frac{1}{2} |\nabla u(x)|^2 - u(x) \right) dx + \beta \int_{\partial\Omega} u(x)^2 dx, \tag{20}$$

where $\beta \int_{\partial\Omega} u(x)^2 dx$ is the penalty term. $\beta$ = 750 was chosen with a spacing of 0.01 in each dimension to calculate the gradients.

The network used to solve this problem included 4 block layers (= 8 dense layers + 4 skip connections) along with a dense layer at the start to bring the dimension of the input to 10 and a linear dense layer at the end to produce the output.

2 shows our model output for this problem. It can be seen that this is in agreement with the well known solution. This provides the first evidence that our model is implemented correctly.

### 2.3.2 Problem 2: 2D Poisson with boundary discontinuity

$$\begin{cases} \Delta u = 1, & x \in \Omega, \\ u = 0, & x \in \partial\Omega, \end{cases} \tag{21}$$

where $\Omega = \{(x, y) \in (-1, 1) \times (-1, 1) \backslash [0, 1) \times 0\}$.

It is shown in [XX] that this problem suffers from a corner singularity which is caused due to the discontinuous nature of its boundary. It is observed from asymptotic analysis that the solution follows

$$u(x) \equiv u(r, \theta) = r^{\frac{1}{2}} sin(\frac{\theta}{2}) \tag{22}$$

near the centre of the domain. Models of this type have been extensively used to help developing and testing adaptive finite element methods. The network used to solve this problem remains the same as the first one to test if the same model has the ability to solve this difficult boundary condition. The solution is presented below in comparison with the solution from the original publication of this method.

As can be observed from 3, our output agrees with that of the original paper, further reinforcing that our model implementation is accurate.

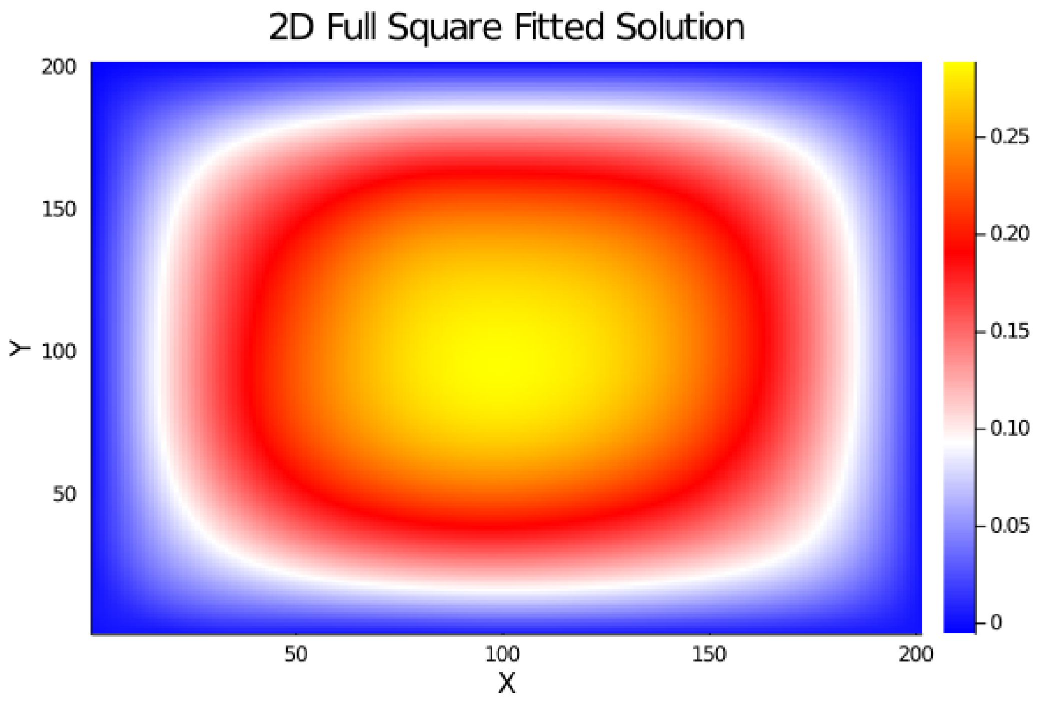


Figure 2: DeepRitz output for the solution of the 2D Poisson equation on a uniform square domain

#### 2.3.3 Problem 3: 2D Poisson with Neumann boundary condition

$$\begin{cases} -\Delta u + \pi^2 u = 2\pi^2 \sum cos(\pi x_k), & x \in \Omega, \\ \frac{\partial u}{\partial n}|_{\partial[0,1]^d} = 0, & x \in \partial\Omega, \end{cases} \tag{23}$$

where $\Omega = \{(x, y) \in [0, 1]^d\}$.

This problem has the following exact solution:

$$u(x) = \sum cos(\pi x_k) \tag{24}$$

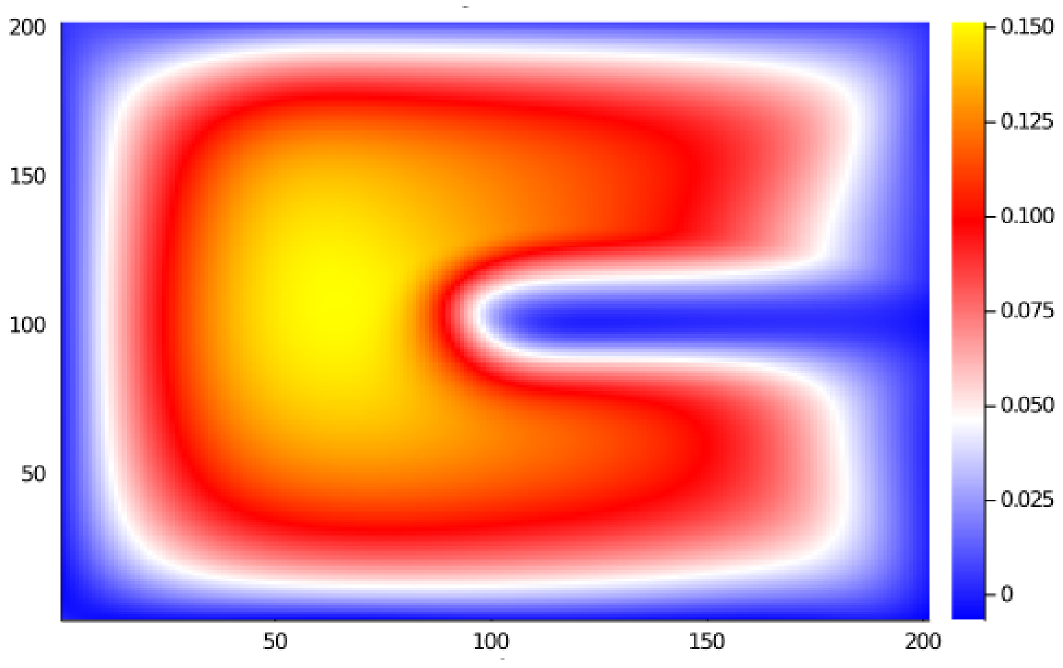
The network used to solve this problem involved using 3 blocks along with an input and an output linear layer. This problem poses another challenge in that the boundary condition itself requires computing a derivative. A simple finite difference scheme was employed to compute these derivatives, which were inbounded to improve performance. The following loss function was used:

$$I(u) = \int_{\Omega} \left( \frac{1}{2}(|\nabla u(x)|^2 + \pi^2 u(x)^2 - f(x)u(x))dx \right) + \beta \int_{\partial\Omega} u(x)^2 dx, \tag{25}$$

The following figure juxtaposes the model solution with the true solution and shows that they indeed agree with each other.

## 3 Deep Galerkin Method

The Deep Galerkin Method introduced by [Sirignano] is another mesh-free method that approximates the solution of a PDE with neural networks. Like the Deep Ritz method, the network uses randomly sampled points from the domain of the problem and uses stochastic gradient descent to optimize its weights and biases. This method is able to circumvent the curse of dimensionality by processing



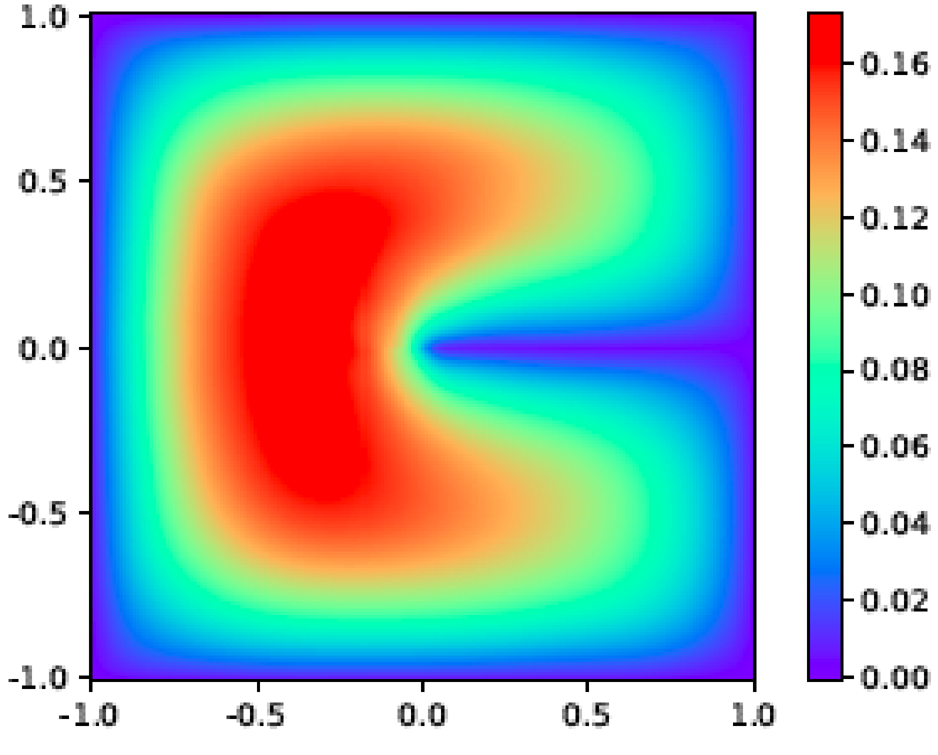
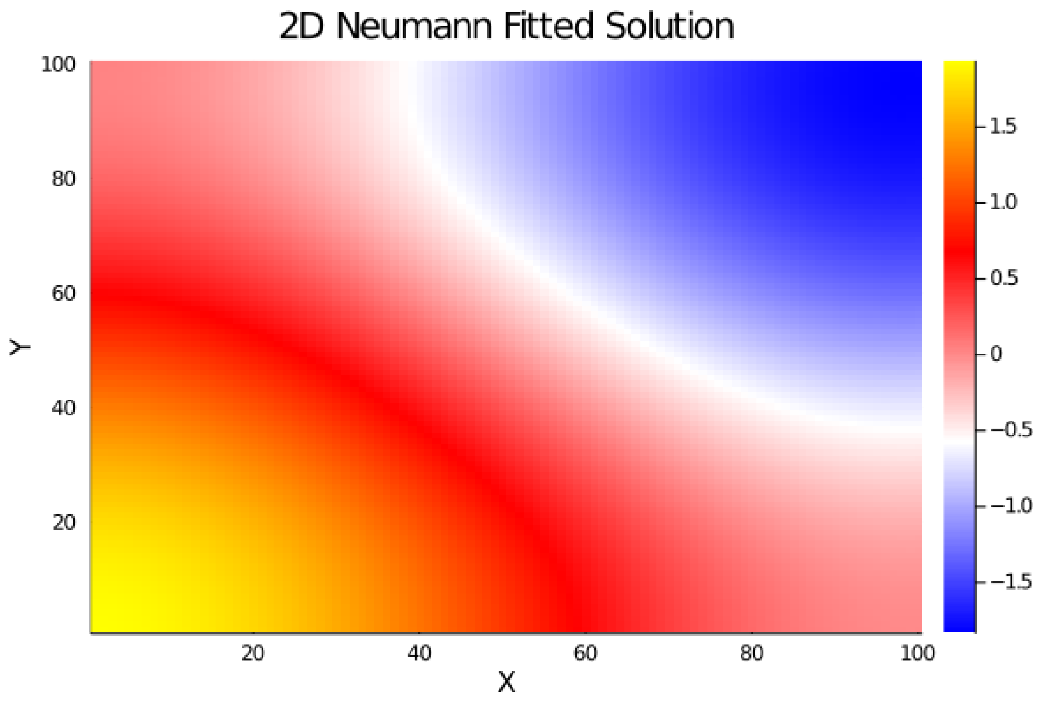


Figure 3: Comparison of our model output with the paper's output for the discontinuous boundary case



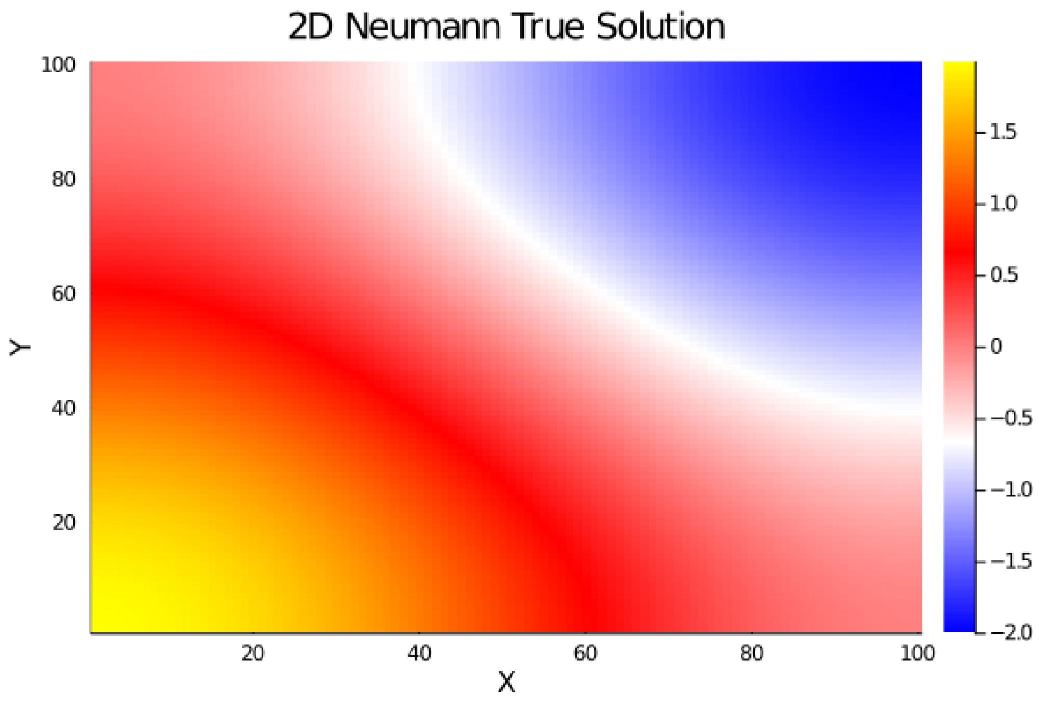


Figure 4: Comparison of our model output with the analytical solution for the Neumann boundary condition

small batches of points from different regions in the domain, also avoiding exorbitant computational costs associated with grid-based methods in the process.

## 3.1 Overview of the Algorithm

DGM primarily solves PDEs of the following form:

$$\begin{cases} (\partial_t + \mathcal{L})u(t,x) = 0, & (t,x) \in [0,T] \times \Omega \\ u(0,x) = u_0(x), & x \in \Omega \\ u(t,x) = g(t,x), & (t,x) \in [0,T] \times \partial\Omega \end{cases} \tag{26}$$

DGM approximates $u$ with a parameterized function $f(t,x;\theta)$ which is represented by a deep neural network with parameters $\theta$. To ensure that the resulting solution conforms to all of the constraints above, the loss function is carefully crafted as a sum of all these components as follows:

- The first item in PDE above describes the behaviour of the operator acting on $u$. Since it is being approximated by the $f$, a way to measure the extent to which the neural network approximates the operator is by the following expression, which is the first component of our loss function:

$$\left\|(\partial_t + \mathcal{L})f(t,x;\theta)\right\|^2_{[0,T]\times\Omega,\nu_1} \tag{27}$$

- In the same vein, the extent to which the solution satisfies the boundary condition is given by:

$$\left\|f(t,x;\theta) - g(t,x)\right\|^2_{[0,T]\times\Omega,\nu_2} \tag{28}$$

- Similarly for the initial condition:

$$\left\|f(0,x;\theta) - u_0(x)\right\|^2_{\Omega,\nu_2} \tag{29}$$

The total loss is thus the sum of all the three losses defined above as follows:

$$L(\theta) = \left\|(\partial_t + \mathcal{L})f(t,x;\theta)\right\|^2_{[0,T]\times\Omega,\nu_1} + \left\|f(t,x;\theta) - g(t,x)\right\|^2_{[0,T]\times\Omega,\nu_2} + \left\|f(0,x;\theta) - u_0(x)\right\|^2_{\Omega,\nu_2} \tag{30}$$

On the basis of the above, a general outline of algorithm is provided which can be tweaked for the specific problem of interest.

## 3.2 Neural network architecture

The DGM structure proposed by Sirignano and Spiliopoulos consists of 3 layers: an input layer, a set of hidden layers and an output layer. The hidden layers are sets of LSTM-like layers which take in not only the output of the previous layer but also the original input for computation. An outline of the whole algorithm is given below

Operations taking place within each hidden DGM layer are very similar to those in Highway Networks. The inner workings of a hidden layer are shown in below in a schematic and in the form of a set of equations.

$$S^1 = \sigma(w^1.x + b^1) \tag{32}$$

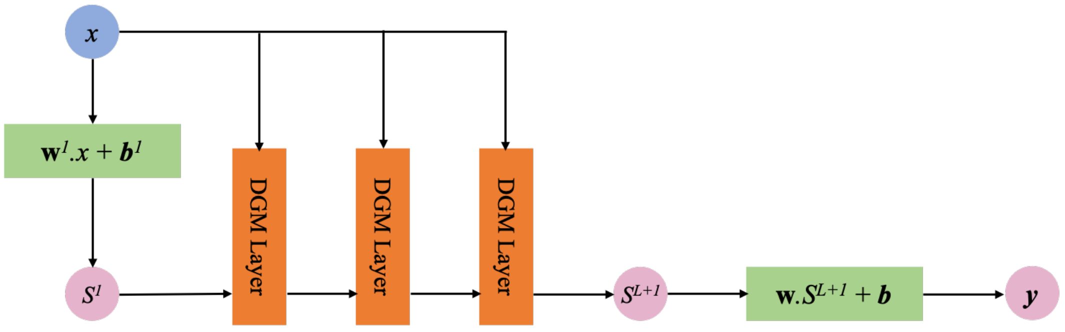
$$Z^l = \sigma(u^{z,l}.x + w^{z,l}.S^l + b^{z,l}), l = 1, \cdots, L \tag{33}$$

$$G^l = \sigma(u^{g,l}.x + w^{g,l}.S^l + b^{g,l}), l = 1, \cdots, L \tag{34}$$

$$R^l = \sigma(u^{r,l}.x + w^{r,l}.S^l + b^{r,l}), l = 1, \cdots, L \tag{35}$$

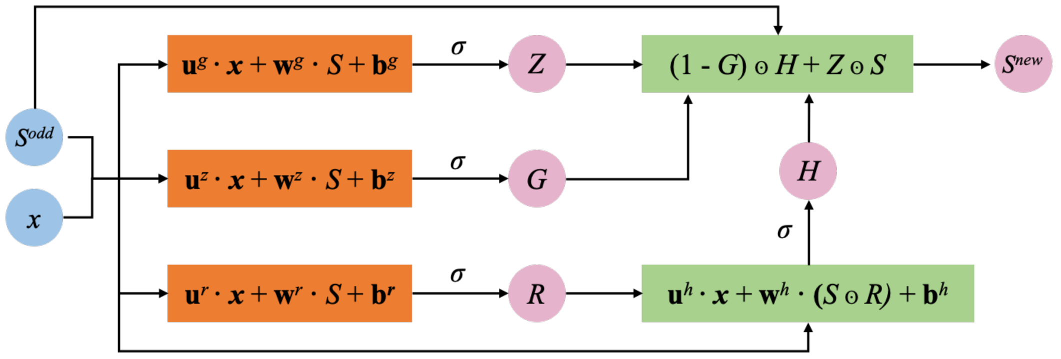
**Algorithm 1** DGM Model Training

```
1: Initialize parameters $\theta_0$ using Xavier uniform initialization
2: Generate samples for loss function computation from within the domain and the spatio-temporal
   boundaries
       • $(t_n, x_n)$ from $[0, T] \times \Omega$ according to $\nu_1$
       • $(\tau_n, z_n)$ from $[0, T] \times \partial\Omega$ according to $\nu_2$
       • $w_n$ from $\Omega$ according to $\nu_3$
3: Compute $L(\theta)$ for the samples collected above
       • $L_1(\theta_n; t_n, x_n) = \left\|(\partial_t + \mathcal{L})f(\theta_n; t_n, x_n)\right\|^2$
       • $L_2(\theta_n; \tau_n, z_n) = \left\|f(\tau_n, z_n) - g(\tau_n, z_n)\right\|^2$
       • $L_3(\theta_n; \tau_n, z_n) = \left\|f(0, w_n) - u_0(w_n)\right\|^2$
       • $L(\theta_n; s_n) = L_1(\theta_n; t_n, x_n) + L_2(\theta_n; \tau_n, z_n) + L_3(\theta_n; \tau_n, z_n)$
4: Perform a gradient descent step with the Adam optimization function
       $\theta_{n+1} = \theta n - \alpha_n \nabla_\theta L(\theta_n; s_n)$    (31)
5: Repeat the steps above till $\|\theta_{n+1} - \theta n\|$ comes below a tolerance level
```



$x$
$\mathbf{w}^1.x + \boldsymbol{b}^1$
$S^1$
DGM Layer
DGM Layer
DGM Layer
$S^{L+1}$
$\mathbf{w}.S^{L+1} + \boldsymbol{b}$
$y$

Figure 5: A schematic of the whole DGM architecture



$S^{odd}$
$x$
$\mathbf{u}^g \cdot x + \mathbf{w}^g \cdot S + \mathbf{b}^g$
$\mathbf{u}^z \cdot x + \mathbf{w}^z \cdot S + \mathbf{b}^z$
$\mathbf{u}^r \cdot x + \mathbf{w}^r \cdot S + \mathbf{b}^r$
$\sigma$
$Z$
$G$
$R$
$(1 - G) \circ H + Z \circ S$
$H$
$\mathbf{u}^h \cdot x + \mathbf{w}^h \cdot (S \circ R) + \mathbf{b}^h$
$S^{new}$

Figure 6: A schematic of a hidden layer in the DGM

$$H^l = \sigma(u^{h,l}.x + w^{h,l}.(S^l \odot R^l + b^{h,l}), l = 1, \cdots, L \tag{36}$$

$$S^{1+1} = (1 - G^l) \odot H^l + Z^l \odot S^l, l = 1, \cdots, L \tag{37}$$

$$f(t, x; \theta) = w.S^{L+1} + b \tag{38}$$

In the equations above, $\odot$ denotes the Hadamard or element-wise product. Like an LSTM, the weights of each subsequent layer decides how much of the information goes forward from the previous layer. According to the authors of DGM, the Hadamard products used above help represent more complicated functions. Skip connection employed by this architecture mitigate the vanishing gradient problem that is typical in very deep neural networks.

## 3.3 Numerical Experiments

The DGM implementation for Julia was optimized in the same way the Deep Ritz code was as described in the previous section.

### 3.3.1 Problem 1: European Call Option Pricing

$$\begin{cases} \partial_t g(t,x) + rx.\partial x g(t,x) + \frac{1}{2}\sigma^2 x^2.\partial_{xx} g(t,x) = r.g(t,x) \\ g(T,x) = G(x) \end{cases} \tag{39}$$

This second order PDE has the following exact solution inn the Black-Scholes equation:

$$g(t,x) = x\Phi(d_+) - Ke^{-r(T-t)}\Phi(d_-) \tag{40}$$

where

$$d_{+/-} = \frac{ln(x/K) + (r + / - \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}} \tag{41}$$

For the experiment, we chose the interest rate $r = 5\%$, volatility $\sigma = 25\%$, initial stock price $S_0 = 50$, time to maturity $T = 1$ and the strike price of the option $K = 50$.

Several experiments were carried out before we decided to sample uniformly for all loss computations. Since stock prices tended to follow the log-normal distribution, we tried using the same distribution for sampling. But this yielded poor results. Following the lead of the paper, we decided to sample uniformly for all loss computations, which still gave poor results due to an early in option prices before the terminal time point. In order to mitigate this effect, time-space pairs were now sampled from a longer space domain which finally matched the analytical solution. The figure below elucidates this point.

### 3.3.2 Problem 2: American Put Option Pricing

$$\begin{cases} \partial_t g(t,x) + rx.\partial x g(t,x) + \frac{1}{2}\sigma^2 x^2.\partial_{xx} g(t,x) - r.g(t,x) = 0, & (t,x) : g(t,x) > G(x) \\ g(t,x) \leq G(x), & (t,x) \in [0,T] \times R \\ g(T,x) = G(x), & x \in R \end{cases} \tag{42}$$

where $G(x) = (K - x)_+$

Unlike the European Call Option PDE, this PDE lacks an analytical solution due its free boundary condition. The parameters used here are the same as above.

The free boundary condition is handled by incorporating it into our loss function as follows:

$$\left\| max(-f(t,x;\theta) - (K-x)_+), 0 \right\|^2_{[0,T]\times\Omega,\nu_1} \tag{43}$$

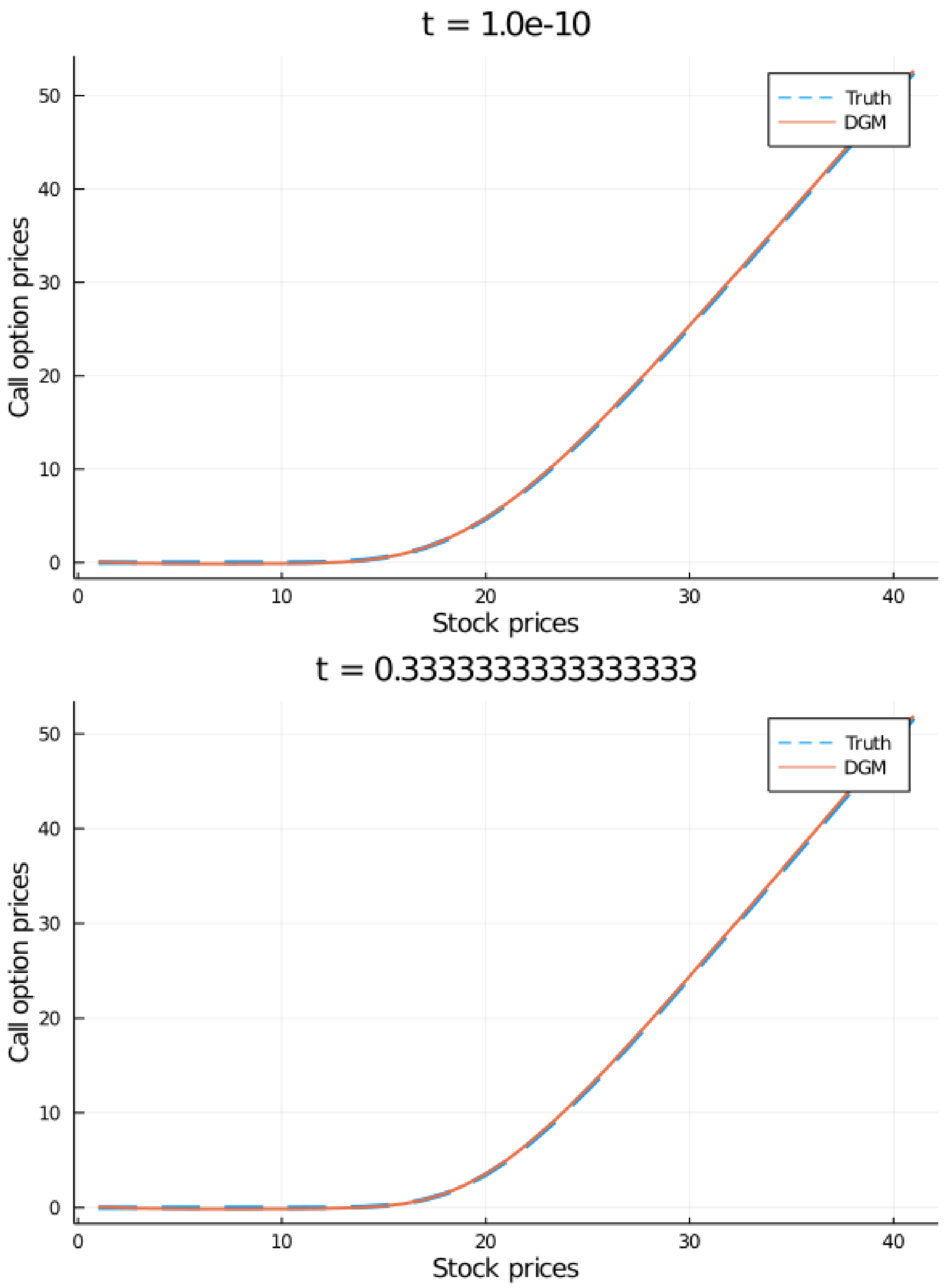


Figure 7: DGM vs. Black Scholes for European Call Option Pricing problem for t = $0, \frac{T}{3}$

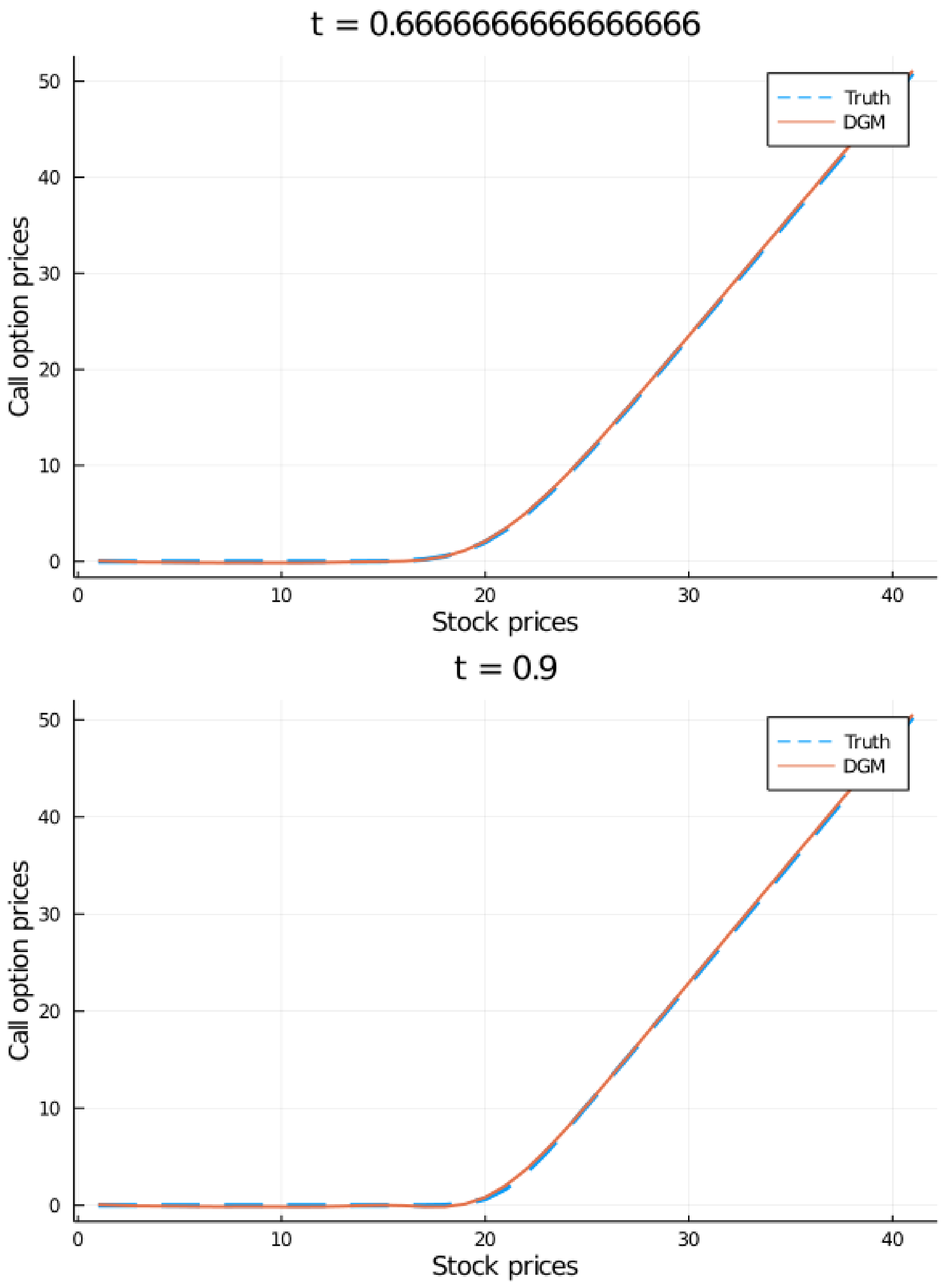
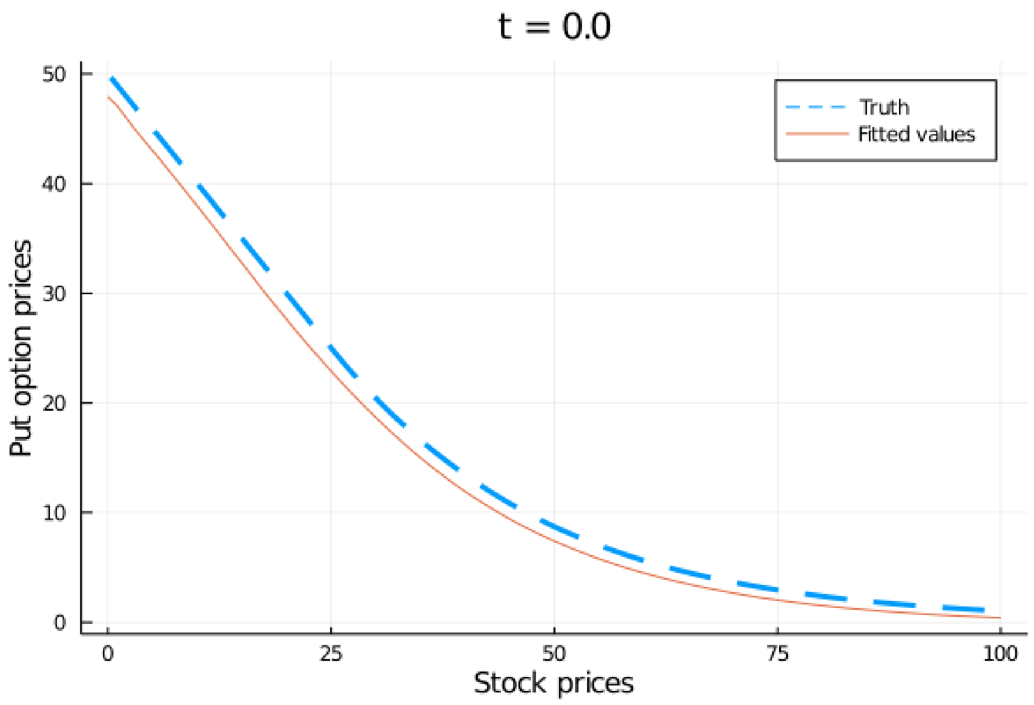


Figure 8: DGM vs. Black Scholes for European Call Option Pricing problem for t = $\frac{2T}{3}, T$



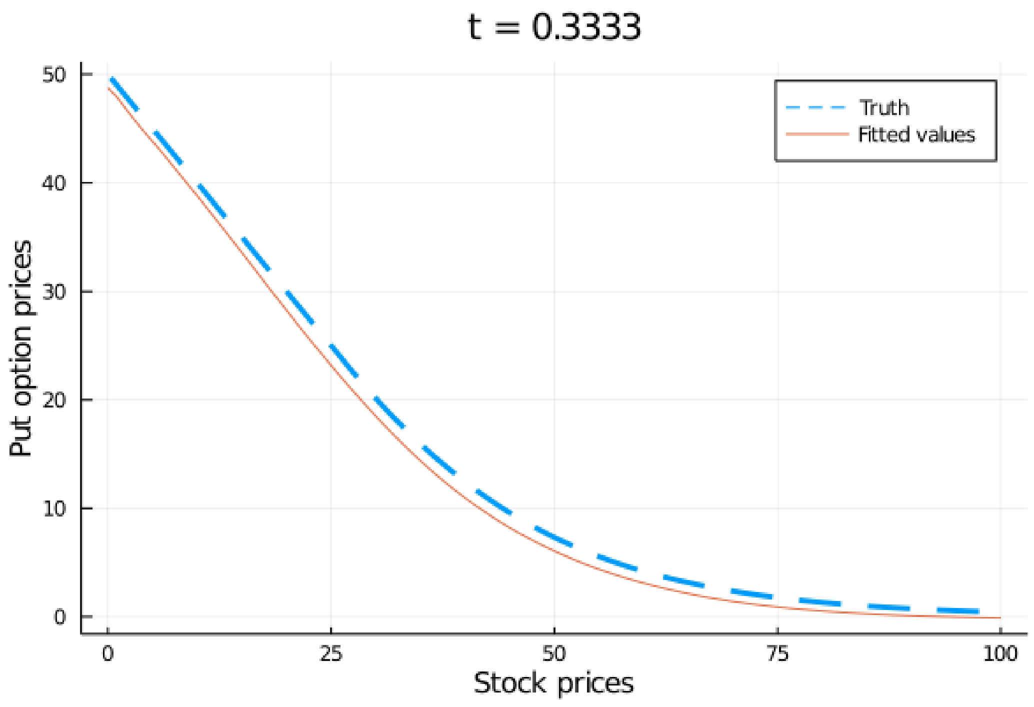


Figure 9: DGM vs. Black Scholes for American Put Option Pricing problem for t = $0, \frac{T}{3}$

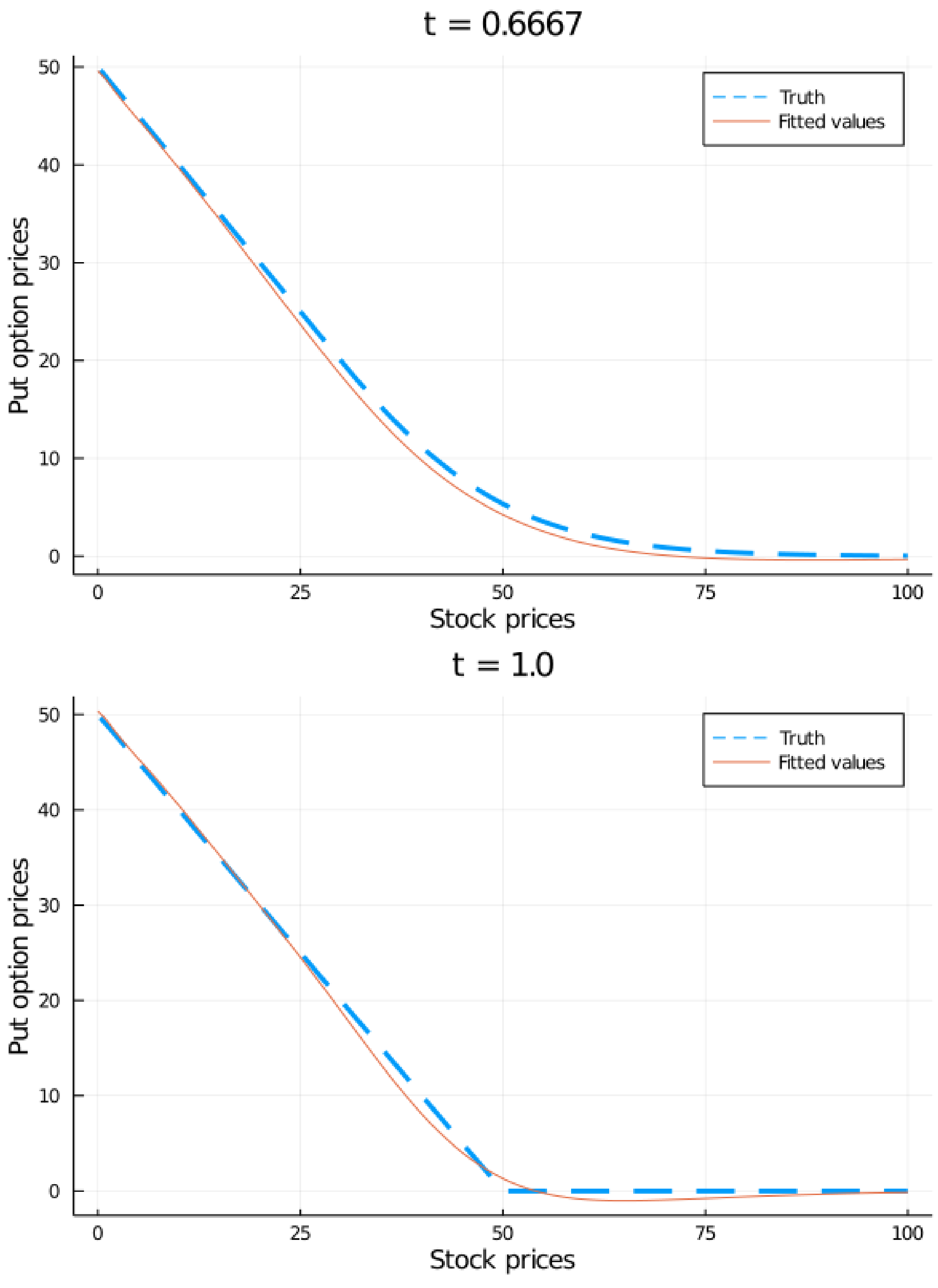


Figure 10: DGM vs. Black Scholes for European Call Option Pricing problem for t = $\frac{2T}{3}, T$

Due to lack of an analytical solution, a finite-element method was implemented to benchmark our result. The plots below compare our output with those of the FEM. As can be observed, our DGM implementation is able to replicate the FEM output to fairly high degree of accuracy, reinforcing that DGM is correctly implemented.

## 4 Conclusion and Future Work

In this project, we discussed two mesh-free deep learning based partial differential equation solvers in detail, the Deep Ritz method and the Deep Galerkin method. A summary of the mathematical foundations underpinning these models was provided along with detailed explanations about the workings of each of these was provided, supplemented by schematic diagrams and algorithms. These models were then implemented in Julia and the efficacy of these implementations was exhibited through numerical experiments, the results of which were benchmarked either against well-known analytical solutions or against their finite-element counterparts, while providing an overview of the optimization techniques employed to improve the performance of our code.

While this work covers two of the most mesh-free techniques in the field, this project is far from a comprehensive review of these techniques in general. Though we optimized our code to the best of our knowledge and resources at hand, there is still a lot of room for improvement in this direction. This work can be extended in many novel directions going forward. One particularly interesting avenue would be to explore parallelization opportunities during loss computation. Since we compute inner and boundary losses separately almost all the time in the examples discussed, it should be possible to multi-thread these computations to improve model training performance.

Another avenue to consider would be to consider introducing an auto-encoder model that learns latent space representations of the different forms of boundary conditions and geometries encountered in this work thus resulting laying the foundations of domain-invariant model. Building a CNN based solver on top of this encoder-decoder structure could then improve generality and robustness to boundary conditions and geometries.

## Acknowledgement

The author expresses his deepest gratitude to Dr. Rackauckas for going out of his way to guide him throughout this project. The author want to thank Ranjan and Sung for their efforts in evaluating homeworks and clearing any doubts. Finally, the author is very thankful for all the guidance provided by his advisor Prof. Lermusiaux during the course of the semester.

## References

[1] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

[2] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018.

[3] Weinan E and Bing Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems, 2017.

[4] L.C. Evans and American Mathematical Society. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 1998.

[5] Reza Khodayi-Mehr and Michael M. Zavlanos. Varnet: Variational neural networks for the solution of partial differential equations, 2019.

[6] Lu Lu, Xuhui Meng, Zhiping Mao, and George E. Karniadakis. Deepxde: A deep learning library for solving differential equations, 2020.

[7] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686 – 707, 2019.

[8] Maziar Raissi. Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations, 2018.

[9] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

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

[11] Ben Stevens and Tim Colonius. Finitenet: A fully convolutional lstm network architecture for time-dependent partial differential equations, 2020.

[12] Nick Winovich, Karthik Ramani, and Guang Lin. Convpde-uq: Convolutional neural networks with quantified uncertainty for heterogeneous elliptic partial differential equations on varied domains. *Journal of Computational Physics*, 394, 05 2019.

[13] Ali Girayhan Özbay, Sylvain Laizet, Panagiotis Tzirakis, Georgios Rizos, and Björn Schuller. Poisson cnn: Convolutional neural networks for the solution of the poisson equation with varying meshes and dirichlet boundary conditions, 2019.