Solving the Heat Equation with Neural Networks

Alexander Havrilla & Alden Pritchard
Carnegie Mellon University
Pittsburgh, PA 15213
alumhavr@andrew.cmu.edu; atpritch@andrew.cmu.edu

December 9, 2020

1 Introduction

The Deep Galerkin Method was first proposed in 2018 by Justin Sirignano and Konstantinos Spiliopoulos, who used the method to numerically solve the Black-Scholes equation for options pricing. The main advantage of the DGM is that it is mesh-free, and as a result does not suffer from the curse of dimensionality, allowing us to compute values in feasible time for higher-dimensional setups. Sirignano and Spiliopoulos proved rigorously that using a deep neural network, as the number of layers goes to infinity, the network approximation converges point wise to the true solution. This is done by choosing a loss function which consists of distance from some initial condition, distance from some boundary condition, and distance from the size of the differential operator applied to the function. By minimizing the sum of these three terms, the method aims to generate a function which closely approximates the initial condition, boundary condition, and a solution on the interior of the domain, thereby giving a numerical solution to the PDE. We aim to apply this method to solve the heat equation.

2 Background

Here we review the mathematics of PDEs.

2.1 PDEs and Heat Equation

Partial differential equations, or PDEs, define a subset of differential equations where derivatives with respect to different variables exist in the same equation. PDE's are commonly found in nature as they can describe many different kinds of diffusion processes. For example, the heat equation describes the diffusion of heat in a specified number of dimensions. In the 1-D case, there does exist a closed-form analytical solution to the heat equation. However, in the 2-D

and more general multivariate case, no analytical solution is known to exist for sufficiently complex boundary conditions. As a result, if we wish to know how heat will diffuse in a given area under some specified initial conditions, then we need to find an approximation to the heat equation.

2.2 Traditional Approximation Methods

Traditional PDE approximation methods involve creating a mesh over the domain of the problem. In the 1-D case, this amounts to breaking the domain interval into some set of n evenly-spaced points. At each point the PDE is approximated as an ODE with initial conditions satisfying the original PDE and then solved numerically at each point in time. When each of these discrete solutions is taken as a whole, we get an approximate solution to the PDE.

This process of meshing the domain from a continuous problem into a discrete set of problems and solving the individual ODEs requires a significant amount of both work and approximations, both of which contribute to making solving PDEs very numerically challenging. Since the number of mesh points increases exponentially in the dimension of the problem, it is necessary to use either a smaller domain or a larger distance between mesh points if we want to solve the PDE in a reasonable amount of time. However, as the distance between mesh points increases, the stability of the approximation breaks down. Likewise, restricting the size of the domain yields less information about the PDE than we may want. This trade-off demonstrates the difficulty in numerically solving PDEs, as we are forced to choose between runtime and accuracy.

3 Related Work

Solving PDEs using the DGM architecture was originally proposed by Sirignano and Spiliopoulos in 2018. The weights for the network are determined by training the neural network via stochastic gradient descent with backpropogation, adam optimizer, and momentum. For the loss function, the authors propose using the sum of distances from the boundary, initial, and differential conditions to approximate the error in the solution. The theoretical inspiration for the DGM lies in its similarity to the Galerkin method. However, there is also a theoretical argument for why using a neural network might be effective in approximating PDEs. Specifically, Sirignano and Spiliopoulos proved that as the neural network depth goes to infinity, the neural network approximation converges point-wise to the PDE solution.

Al-Aradi et al applied the DGM to solve various PDEs, including the Black-Scholes PDE for options pricing in finance, the Fokker-Planck PDE in statistical mechanics, Mean Field Equations for backward stochastic differential equations, and optimal control problems. Through fine-tuning model hyper-parameters, the

authors were able to achieve similar performance to popular traditional methods on each PDE. This suggests that the DGM can be used for solving PDEs, and that the main obstacle is finding the right combination of parameters that best suits the model to the PDE.

3.1 DGM Architecture

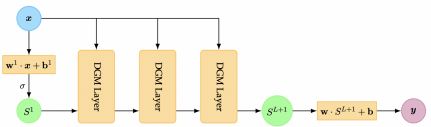


Figure 5.2: Bird's-eye perspective of overall DGM architecture.

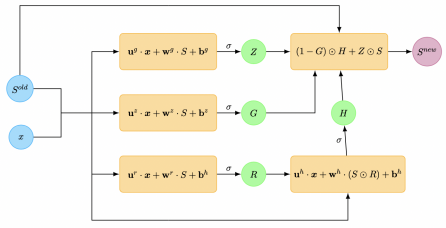


Figure 5.3: Operations within a single DGM layer.

The overall DGM network architecture consists of a fully connected layer, followed by some number of stacked DGM layers that take as input the original input x and the output of the previous DGM layer, with another fully connected layer at the end. We experimented with different depth networks in anticipation of a trade-off between training time and network accuracy as implied by the results from Sirignano and Spiliopoulos.

3.2 Individual DGM layer

Each individual DGM layer consists of four sums of linear maps and an output function. In total, each layer contains eight weight matrices, four bias vectors, and four activation functions, which are eventually combined in the layer's output function.

4 Methods

Here we discuss our model architecture, sampling strategy, and our initial/boundary conditions. Our strategy is similar to [4], leveraging the DGM architecture. We solve the (fairly) general heat PDE:

$$\begin{aligned} u_t(t,x) &= k\Delta u(t,x) && (t,x) \in [0,1] \times [-1,1]^d \\ u(0,x) &= f(x) && f \in C_c(\mathbb{R}^d) \\ u(t,\partial[-1,1]^d) &= c && c \geq 0 \end{aligned}$$

where the dimension is arbitrary. In practice we take $k = 1$.

4.1 Sampling Methodology

In order to train the network we randomly sample a number of points from the domain at each training step. In particular we sample points from spatial domain when $t = 0$ to approximate the initial condition, points from the boundary of the spatial domain for arbitrary $t > 0$, and points from the interior when $t > 0$ and x is not on the spatial boundary.

We found a uniform sampling over both space and time works best, when compared with gaussian sampling over space with uniform sampling over time and gaussian sampling over space with poisson sampling over time. This intuitively makes sense, as the resulting uniform points are more likely give a better approximation over all times and space. Using nonuniform distributions improves the approximation in areas with high density, but underperforms in lower density and thus undersampled areas.

4.2 Initial/Boundary Conditions

Our implementation is quite robust to initial condition, allowing any distribution to be satisfied. In practice we use mostly gaussian heat spikes, since these allow us to approximate simple dirac delta initial conditions which can be solved analytically. This is specified using tensorflow functions. Our boundary conditions are less robust, at this time only supporting fixed constant heats for each boundary, although this could be easily improved. In practice it's often the case the boundary has 0 heat anyway(think of it as adjacent to an open system/heat sink). This is the case we test on.

4.3 Architecture

Our architecture is similar to the setup in [4]. We use a varying number of DGM layers sandwiched by two linear layers and a maxpool. After tuning we present most of our results using a model with 3 DGM layers. This is constructed using Keras and run using Tensorflow.

Our loss function is the sum of three terms. The first penalizes the L_2 space distance of the network from the PDE, ie. we compute higher order gradients using tensorflow and use these to compute the PDE for a point evaluated by the network. The second term penalizes distance in L_2 from the initial condition, and the last term distance in L_2 from the boundary conditions. On each of these we introduce a scaling factor that we tune, controlling the weight of each penalty.

Training is done using SGD with momentum exactly as done in [4].

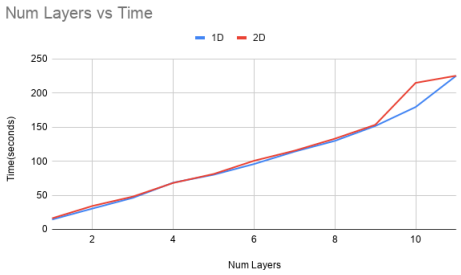
5 Results

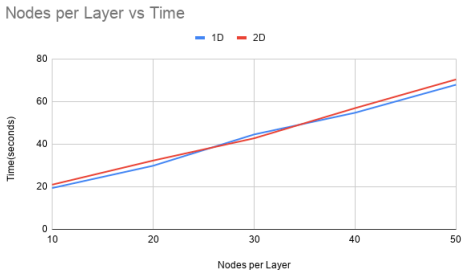
We studied training time and accuracy of the approximation against the depth of the network, number of samples, dimension of the PDE, and complexity of the

initial and boundary conditions. Here we summarize the results and compare accuracy against a traditional finite element method in 1D and 2D cases.

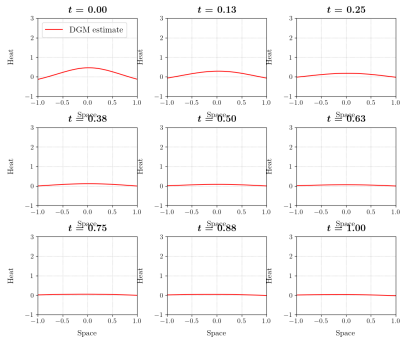
5.1 Depth of Network

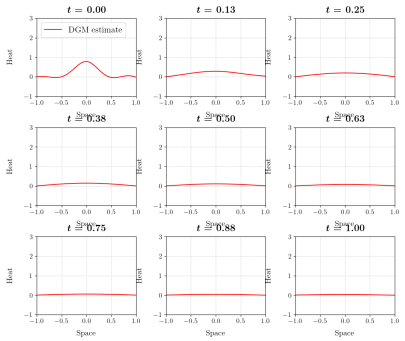
Here we study the training and classification time of 100 points vs network complexity.





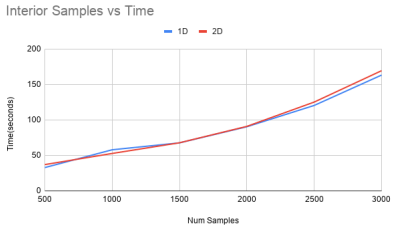
We plot the resulting approximations for 1 layer and 3 layers.



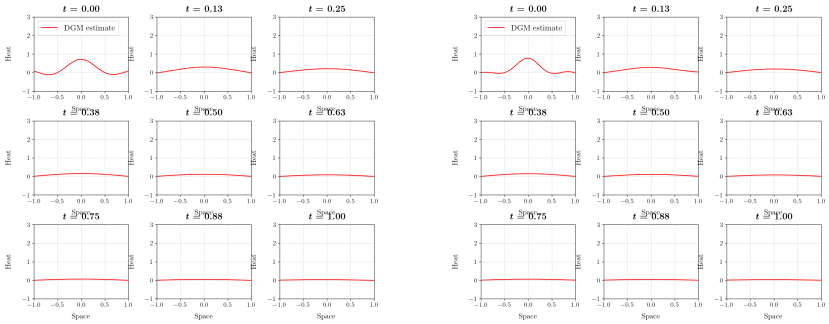


5.2 Number of Samples

Here we study the training and classification time of 100 points vs number of samples.

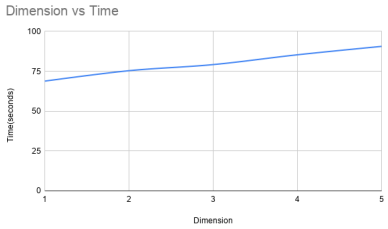


We plot the resulting approximations for 1 layer and 3 layers.

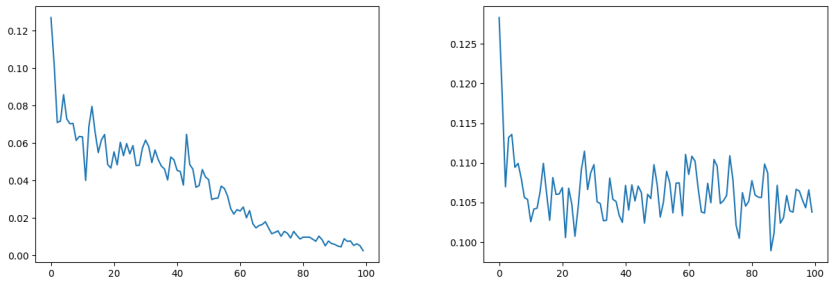


5.3 Dependence on Dimension

Here we study the training and classification time of 100 points vs dimension.

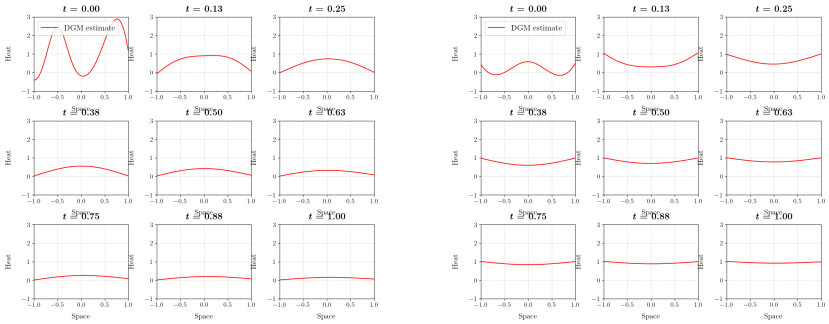


Since above 2 dimensions is difficult to visualize, we present loss curves for 1 dimension and 5 dimensions.

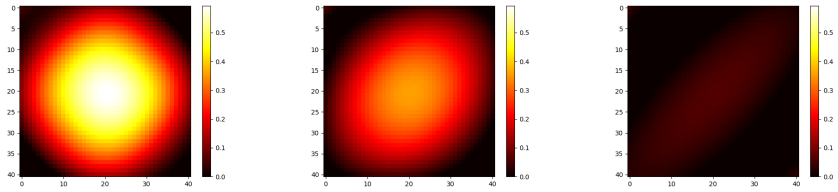


5.4 Dependence on Initial/Boundary Conditions

Here we approximate a PDE with initial condition $e^{-(x-.5)^2} + e^{-(x+.5)^2}$ with 0 boundary condition, and a PDE with initial condition e^{-x^2} with boundary fixed at heat 1 respectively.



We consider the similar 2d case of the second equation.



6 Analysis

Now we discuss the results presented in the above section.

6.1 Depth of Network

As was hoped the training and evaluation time of the network on a fixed parameter set ranging over the number of DGM layers scales nearly linearly, at least for small cases. It seems for the heat equation a sufficient depth is 3, which we found gets a reasonable result for most initial and boundary conditions. Note in contrast networks with depth 1 are not powerful enough, not capturing the initial condition very well but still achieving a smoothing effect as time runs.

We also scaled the number of nodes per DGM layer and saw a similar result, both in terms of training time and approximation fidelity.

6.2 Number of Samples

Here we see the number of randomly generated in the interior samples seriously impacts the amount of time training takes, as we would expect. Surprisingly, we achieve a pretty good approximation even with only 500 samples, when compared to the same parameters with 1500 samples.

6.3 Dependence on Dimension

The time spent training scales well with the dimension, at least for small dimensions. This is good to see as ease of computing higher dimensions is a major selling point for the DGM approach. However this is a bit misleading as we do not increase the number of samples and thus our approximation gets worse as dimension increases, as is suggsted by the loss curves plotted. This makes sense as the number samples to approximate in a higher dimension with fixed accuracy certainly must increase, by at least virute of the larger distance metric in higher dimensions.

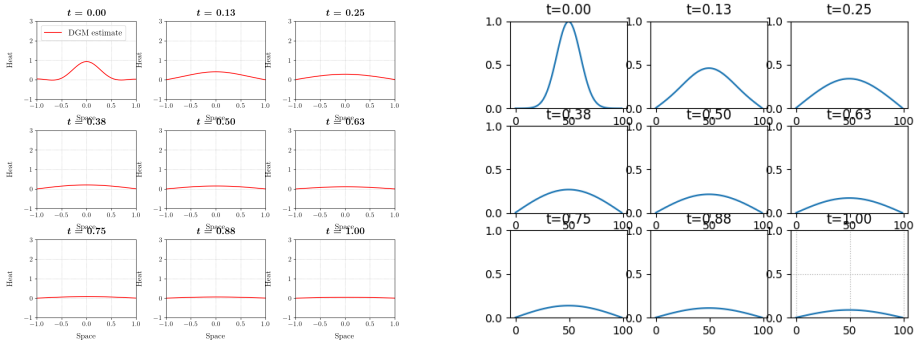
6.4 Dependence on Initial Conditions

We were pleasantly surprised to see how robust the DGM architecture is to different boundary and initial conditions. Qualitatively in the first case, notice how in the heat coalesces at $x = 0$ and then disperses. Similar complex behavior is correctly exhibited in the second plot, with different boundary conditions.

Our loss plots confirm the tightness of this approximation.

6.5 Comparison With Traditional Methods

From this tuning we found the optimal hyperparameters at dimension 1 were 3 dgm layers, 50 nodes per layer, with 1500 interior points, 500 boundary points, and 500 initial points sampled. We then compared this to a standard 1D heat equation solver. Here is a side by side comparison:



Further the losses (which we wanted to fit but did not have space) for the final model confirm a good approximation.s

Recall the network is a smooth function approximating the PDE solution as a opposed to a discretization, allowing us to query the result at any point: a huge advantage. Further we found on average the time to compute 100 points is significantly less for the DGM architecture than the 1D solver. Further we have an easy and natural generalization to higher dimensions.

References

[1] Al-Aradi A. Correia A. Naiff D. Jardim G. Solving Nonlinear High-Dimensional Partial Differential Equations via Deep Learning.

[2] Han. J. Solving High-dimensional PDEs Using Deep Learning. https://www.pnas.org/content/pnas/115/34/8505.fu

[3] Penko V. https://github.com/vitkarpenko/FEM-with-backward-Euler-for-the-heat-equation

[4] Sirignano. J, spiliopoulos K. DGM: A deep learning algorithm for solving partial differential equations. https://arxiv.org/abs/1708.07469