A semi-discretization method for solving parabolic Partial Differential Equations with Deep Learning

Ruining (Ray) Wu
Department of Computer Science
University of Toronto
Toronto, ON, Canada
rwu@cs.toronto.edu

Abstract

We introduce the Deep Galerkin Method with Timestepping (DGMT), which improves on the Deep Galerkin Method (DGM) [22] by using the dependence of current solution values in time on past solution values and applying a semi-discretization in the time domain. We show improved performance as measured by pointwise error compared to the original method on certain test problems. We also show some other desireable properties of the DGMT.

1 Introduction

Partial Differential Equation (PDE) models are ubiquitous tools for modelling mathematical problems, with two well-known problems being Schrödinger's Equation [21] in quantum mechanics and the Black-Scholes Equation [4] in financial option pricing. These equations belong to the class of PDEs known as parabolic PDEs. A main property of parabolic PDEs is that they are time-dependent problems, that is, the solution at any point in time depends on the solution at all past and current points in time across the entire domain.

The numerical solution of these PDEs typically use a time and space discretization of the points of interest and solve systems of linear equations that arise from it. For example, one approach to solving the Black-Scholes equation would be approximating spatial derivatives with finite difference formulas and applying Crank-Nicolson-Rannacher timestepping [5, 19] to iteratively compute solutions at each timestep, starting at the initial condition until the final time is reached.

However, the dimensionality of the PDE is a prohibitive barrier to solving them efficiently using standard numerical PDE methods. In applications, Schrödinger's Equation has $3N$ dimensions where N is the number of electrons in the physical system, and the Black-Scholes Equation can have as many dimensions as the number of underlying assets of the financial option. Since the total number of gridpoints is the product of the discretization size of all dimensions, the number of gridpoints scales exponentially in the number of dimensions. This is known as the curse of dimensionality.

High-dimensional PDEs are interesting because they can arise from many applications. For example, when we are interested in atoms that have more than a couple of electrons or financial options written on several underlying options, we end up with PDEs that are difficult if not impossible to solve with traditional numerical methods. In practice, financial options can have hundreds of underlying options.

We focus on the one-dimensional Black-Scholes PDE, which is given by

$$V_\tau = \mathcal{L}V = \frac{\sigma^2 S^2}{2} V_{SS} + rSV_S - rV, \tag{1}$$

where τ denotes the reversed time, S denotes the value of the underlying stock, σ denotes the volatility of the stock, r denotes the interest rate, and subscripts denote partial derivatives (i.e.

$V_{SS} = \partial^2 V/\partial S^2$). Typical options are call and put options, which are given by the initial conditions $V_{\text{call}}(S) = \max(S - K, 0)$ and $V_{\text{put}}(S) = \max(K - S, 0)$ for some positive constant K, known as the strike price. We use a one-dimensional Put option for our test problem.

2 Related Work

One approach to overcome the curse of dimensionality is to use the idea that neural networks are universal function approximators [12] and hence can approximate functions to arbitrary accuracy, even in high dimensions. FermiNet [18] is a deep learning architecture that solves the Schrödinger Equation in high dimensions, while two recent approaches for general PDE problems, including finance problems are the Deep Galerkin Method (DGM) [22] and the Deep Backward Stochastic Differential Equation Method (Deep BSDE Method) [6, 10]. In related work, there have been proofs [9, 13] that have shown the capability of neural networks to approximate certain PDEs to arbitrary accuracy, with complexity in polynomial time of both the dimension and the inverse of a prespecified accuracy. Some other works build upon the idea of the Deep BSDE method and use other numerical analysis techniques such as operator splitting [3] to improve the efficiency of the Deep BSDE method. Another example of deep learning for numerical analysis problems is the Deep Ritz method [25] for variational problems.

In general, the idea of using deep learning and neural networks for option pricing is not new; The review paper [20] provides a history of the developments in this field. However, the deep learning methods described in [20] traditionally rely on historical or generated data. What makes the DGM and Deep BSDE approach different is that the neural networks directly approximate the PDE. Hence, they are applicable to any PDE and not just finance problems.

Compared with the Deep BSDE method, there are two important advantages of the DGM that we believe are important:

1. **The DGM is more general**. Whereas the Deep BSDE method solves semilinear heat equations that only allows nonlinearity in the first derivative term, the DGM doesn't enforce any restrictions on the differential operator which allows nonlinearity in the second-derivative (diffusion) term. The relevance to financial problems occurs when we relax assumptions of the Black-Scholes model or consider related problems: many nonlinear problems such as Transaction Cost models [14, 16, 17], Uncertain Volatility models [2], and Passport options [1] give rise to PDE problems with nonlinearity in the diffusion term.

2. **The DGM solves the PDE along the entire domain**. The neural network in the DGM is intended to directly approximate the solution; once a function is learned it can be evaluated anywhere. In contrast, the Deep BSDE method computes values pointwise. This leads to two points in favour of the DGM:

   (a) It is not necessary to re-train the entire neural network to compute prices at different points. The same neural network can be evaluated on a different point in the domain.

   (b) Computation of the derivatives of the value function that are used as hedging parameters is much simpler; either with automatic (symbolic) differentiation or finite differences.

Although the Deep BSDE method has other advantages, namely its efficiency and its use of the problem structure, we consider the advantages of DGM to be more important. By problem structure, we mean the dependence of solution values on past values: $v(\tau^*, x)$ depends on all τ such that $v(\tau \leq \tau^*, \forall x)$. The focus of the project is to attempt to address the "simplicity" of the Deep Galerkin Method by modifying it to exploit the problem structure of parabolic PDEs.

3 Formal Description

First, we will describe the DGM. Then, we will describe our proposed improvement.

3.1 Deep Galerkin Method

The DGM solves PDEs of the form

$$\begin{aligned}
&V_\tau(\tau, S) = \mathcal{L}V(\tau, S) \text{ where } (\tau, S) \in [0, T] \times \Omega \\
&V(\tau = 0, S) = v_0(S) \\
&V(\tau, S) = g(\tau, S) \text{ for } S \in \partial\Omega
\end{aligned} \tag{2}$$

where V is the unknown solution to the PDE, $\mathcal{L}$ is the differential operator that computes the partial derivatives of V with respect to the spatial variable S, τ is the time variable that goes from 0 to T, Ω is the spatial domain, $\partial\Omega$ is the boundary, $v_0(x)$ defines the initial condition, and g defines the Dirichlet boundary conditions.

The authors define a neural network $f(\tau, x; \theta)$ to directly approximate the solution V. The authors minimize sum of squared errors of the solution approximation at internal points, solution approximation at boundary points, and initial condition approximation. They correspond to lines 1, 2, and 3 of the objective function 3 for the DGM.

We use Adaptive Moment Estimation (Adam) [15] as the training algorithm to find the optimal parameters for $f(\tau, x; \theta)$. Since our input is a continuous domain, we are required to resample points from the domain each time. Since we are typically interested in the solution value at or around the strike price, we sample from a normal distribution centered at the strike price K and with variance $K^2/4$.

We use Adam to solve the optimization problem

$$\begin{aligned}
\theta^* = \arg\min_\theta \{G(\tau, x; \theta) \equiv &\|f_\tau(\tau, x; \theta) - \mathcal{L}f(\tau, x; \theta)\|^2_{[0,T]\times\Omega,\nu_1} \\
&+ \|f(\tau, x; \theta) - g(\tau, x)\|^2_{[0,T]\times\partial\Omega,\nu_2} \\
&+ \|f(0, x; \theta) - u_0(x)\|^2_{\Omega,\nu_3}\}
\end{aligned} \tag{3}$$

where for a generic function η, $\|\eta(y)\|^2_{\mathcal{Y},\nu} = \int_{\mathcal{Y}} |\eta(y)|^2 \nu(y) dy$ and ν is a probability density on $y \in \mathcal{Y}$.

After learning the parameters θ^*, $f(\tau, x; \theta)$ approximates $V(\tau, S)$.

3.2 Computational Architecture

The authors of [22] tested DGM mainly on financial derivative pricing problems, and such problems often involve initial conditions that have discontinuous derivatives, while the solution functions are smoother away from the initial point. Therefore, the authors of [22] designed their neural network in a way that captures the "sharp turns" in the initial condition characterized by the cusps at the strike price K by using a neural network architecture that is similar to LSTM networks [11] and Highway networks [23]. Although the neural networks in our proposed improvement do not need to capture the "sharp turns" in the initial condition, as they are used away from the starting point, we nevertheless decide to keep their neural network design.

The DGM is composed of DGM layers, which are defined by

$$\begin{aligned}
f(\tau, x; \theta) &= WS^{L+1} + b \\
S^1 &= \sigma(W^1 x + b^1), \\
Z^\ell &= \sigma(U^{z,\ell} x + W^{z,\ell} S^\ell + b^{z,l}) & \ell &= 1, \ldots, L \\
G^\ell &= \sigma(U^{g,\ell} x + W^{g,\ell} S^\ell + b^{g,l}) & \ell &= 1, \ldots, L \\
R^\ell &= \sigma(U^{r,\ell} x + W^{r,\ell} S^\ell + b^{r,l}) & \ell &= 1, \ldots, L \\
H^\ell &= \sigma(U^{h,\ell} x + W^{h,\ell} (S^\ell \odot R^\ell) + b^{h,l}) & \ell &= 1, \ldots, L \\
S^{\ell+1} &= (1 - G^\ell) \odot H^\ell + Z^\ell \odot S^\ell & \ell &= 1, \ldots, L
\end{aligned} \tag{4}$$

and the activation function σ is $\tanh$. Although other activation functions can be used, compared to an activation function like ReLU the $\tanh$ function is smooth and infinitely differentiable which ensures that the neural network is also smooth and infinitely differentiable. The authors use three DGM layers, that is, $L = 3$.

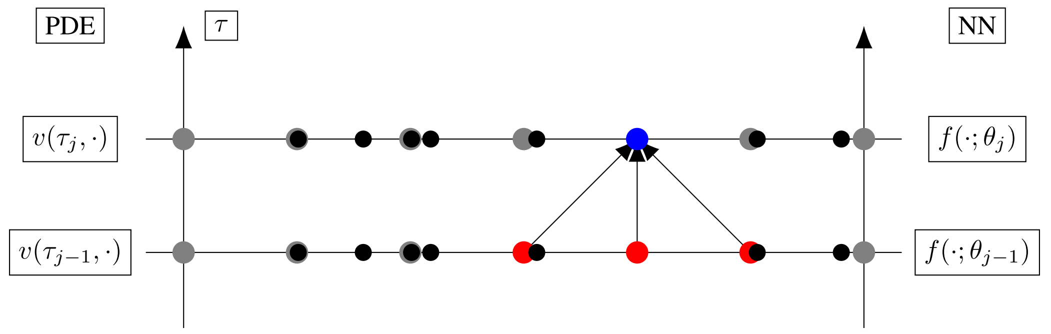

Figure 1: Illustration of the dependence of solution values on previous solution values in time. In traditional PDE methods such as finite difference methods, the partial derivatives at a specific node (colored in blue) are computed using the nodes at the previous point in time and its neighbours (colored in red), and hence each node depends on three previous nodes. However, in higher dimensions there are 3^d nodes, leading to intractable computations. On the left is the PDE and on the right are the NNs that approximate the PDE at various points in time. The NNs are related to each other by applying a time-discretization scheme and sampling gridpoints (colored in black) instead of defining fixed gridpoints (colored in gray). A different sample (colored in black) is used each epoch.

3.3 Proposed Improvements

The original neural network is designed to directly approximate the solution V along the whole domain with one neural network, which means that it does not take into consideration the dependence of future solution values on previous solution values. At the core of our idea is the application of a semi-discretization in time which turns the continuous time domain into discrete time by partitioning the interval $[0, T]$ into N_τ subintervals and relates solution values at a previous point in time to a solution value at the current point in time. We use a combination of Backwards Euler (Fully Implicit) or Crank-Nicolson and leave the spatial dimensions to the neural network to solve. This does not create dimensionality issues since only one dimension (time) is discretized; the treatment of spatial dimensions is unchanged. We call our method the Deep Galerkin Method with Timestepping (DGMT), since the time-discretization is commonly referred to as timestepping in traditional PDE methods. Our idea is similar to semi-discretization methods used in traditional PDE methods such as [24], with the difference being we are applying the discretization in time and [24] applies the discretization in space. Figure 1 illustrates our method, and how we have similar ideas with traditional PDE methods such as finite differences/finite elements.

From words to equations, this means applying a time-discretization and relating the solution values between two consecutive timesteps:

$$(\mathcal{I} - \vartheta \Delta\tau \mathcal{L}) v_j = (\mathcal{I} + (1 - \vartheta)\Delta\tau \mathcal{L}) v_{j-1}, \tag{5}$$

where I is the identity operator, $\Delta\tau$ is the timestep, and v_j denotes $V(\tau_j, \cdot)$.

Instead of a neural network $f(\tau, x; \theta)$ approximating the solution for the entire range of $[0, T]$, now, f will only be a function of x and $f(x; \theta)$ will only approximate the solution at a specific point in time. The objective function will be modified accordingly; in particular, the contribution of the internal region to the squared error becomes

$$\left(\left[f_j(x;\theta) - \vartheta \Delta\tau \mathcal{L} f_j(x;\theta) \right] - \left[f_{j-1}(x;\theta) + (1-\vartheta)\Delta\tau \mathcal{L} f_{j-1}(x;\theta) \right] \right)^2, \tag{6}$$

where f_{j-1} denotes the neural network that approximates v_{j-1} and f_j denotes the neural network that approximates v_j, and there is no longer any contribution from the initial condition, because it is directly incorporated into the computation procedure. Note that there is also no boundary condition, since the problem we are solving for technically does not have a boundary, although one is frequently imposed when truncating the semi-infinite domain of S. The parameter ϑ is chosen to be either 1 or $1/2$ corresponding to a Fully Implicit or Crank-Nicolson time discretization.

3.4 Algorithm Description

We formally describe our algorithm in Algorithm 1. We decide our timesteps based on a equal partition of the time interval $[0, T]$, however, our algorithm easily extends to variable timesteps, which are popular for finite difference and finite element methods [7]. We wish to avoid taking derivatives of the nondifferentiable initial condition, hence, for the first timestep, we set $\vartheta = 1$, which avoids taking derivatives of the initial condition but leads to a Backwards Euler (Fully Implicit) timestepping scheme which has a lower order of accuracy compared to the Crank-Nicolson timestepping scheme which sets $\vartheta = 1/2$. To compensate, we halve the timestep and take two half-sized timesteps with Backwards Euler, and the remaining timesteps with Crank-Nicolson. We note that this is similar to Rannacher smoothing [19], however, we do not make similar claims about improved smoothness since our neural network has a continuously differentiable $\tanh$ activation function, and hence is continuously differentiable.

Algorithm 1 DGMT with ϑ-timestepping

```
1: Pick time points $\tau_j$, $j = 0, ..., M$, with $\tau_0 = 0$ and $\tau_M = T$. Let $\Delta\tau_j = \tau_j - \tau_{j-1}$.
2: Initialize a neural network $f(x;\theta)$
3: Solve the optimization problem
       $$\theta_1 = \arg\min_{\theta} \Big( f(x;\theta) - \Delta\tau_1 \mathcal{L} f(x;\theta) - v_0(x) \Big)^2 \quad (7)$$
   where $v_0$ is the initial condition of the PDE problem.
4: for $j = 2, \ldots, M$ do
5:     Solve the optimization problem
       $$\theta_j = \arg\min_{\theta} \Big( \Big[ f(x;\theta) - \vartheta\Delta\tau_j \mathcal{L} f(x;\theta) \Big] - \Big[ f(x;\theta_{j-1}) + (1-\vartheta)\Delta\tau_j \mathcal{L} f(x;\theta_{j-1}) \Big] \Big)^2 \quad (8)$$
       with $\theta_{j-1}$ as the initial guess for $\theta$.
6: end for
7: $f(x;\theta_M)$ now approximates $V(\tau = T, S)$
```

Since the initial optimization problem (7) is starting from a "cold start", (we follow the authors of the original paper and use Xavier initialization [8]) we need many more training epochs to achieve a reasonable approximation to the optimium, compared to the subsequent optimization problems (8) which have a warm start (the learned parameters for the previous timestep result in a good approximation, since the PDE is limited in how much it can change per timestep), so less epochs in total are used. We use 4000 epochs for the optimization problem (7), but only 500 for each of the subsequent problems (8). In our implementations we create a separate neural network to approximate $f(x;\theta_{j-1})$ and copy the parameters from the original neural networks to the new neural network, and freeze θ_{j-1} so they cannot be modified.

4 Results and Comparison

We compare our results to the original DGM. Although we introduced other improvements such as the non-uniform sampling of the spatial domain, we kept the comparison fair applying it to the DGM also. We use the same predetermined, piecewise constant learning rate as the one given in [22] for our simulation of the DGM method. For the DGMT method we use a similar learning rate for the initial optimization problem (7) with a reduced learning rate schedule and we use much reduced values for the subsequent optimization problems (8). In both cases we greatly reduce the number of training epochs and scale the learning rate accordingly. Specifically, we scale down every part of their training procedure by 10 times: where [22] starts with 5,000 epochs at $\alpha = 10^{-4}$, we only have 500 before reducing the learning rate. It should be pointed out that even with the extended learning rate the DGM method does not have a better performance than the DGMT method, so there is no reason to test the DGM method on the learning rates given for DGMT because it is very likely that convergence will not be reached. For both DGM and DGMT, we use networks with 3 "DGM" layers and 50 internal nodes. However, since the original DGM has both time and space as input,

Name	Computed Value	Relative Error	time(s)
reported performance in [22]	n/a	5×10^{-4}	n/a
original DGM	10.8721900169	6.48×10^{-3}	7.02×10^{2}
	10.8070961609	4.52×10^{-4}	1.01×10^{3}
	10.7799907306	2.06×10^{-3}	1.67×10^{3}
	10.7772666829	2.31×10^{-3}	2.92×10^{3}
	10.8039460486	1.61×10^{-4}	1.22×10^{4}
DGMT with 4 timesteps	10.636006	1.54×10^{-2}	7.34×10^{2}
DGMT with 8 timesteps	10.742462	5.53×10^{-3}	1.07×10^{3}
DGMT with 16 timesteps	10.794114	7.50×10^{-4}	1.74×10^{3}
DGMT with 32 timesteps	10.803133	8.53×10^{-5}	3.02×10^{3}

Table 1: Comparisons of different methods: the original DGM method is examined at different points in time equivalent to the amount of time used in the DGMT method with 4, 8, 16, and 32 timesteps. The exact value of the PDE at the final time and strike price is computed with the Black-Scholes formula. The original DGM method was allowed to run for 100,000 epochs, similar to [22].

and the DGMT method only has space as input, the comparsion cannot be held completely fairly. However, we will point out that we are actually training N_τ neural networks, were N_τ is the number of timesteps compared to the DGM which is just training one neural network. Neverthless, as we shall show, the performance of the DGMT is superior to DGM even with limited time.

4.1 Direct comparison of DGM with DGMT

We summarize our results in Table 1. We take into consideration is the reported performance of the algorithm in the original paper [22] with the caveats that

- The report was for a three-dimensional problem rather than the one-dimensional problem that we tested.
- It was a more difficult American option rather than an European option.

Although the hardware that was available to the authors (Blue Waters Supercomputer with multiple GPUs) is vastly superior to what I had access to (a single GeForce RTX 2060), for measuring the time I trained both systems on the same hardware. Additionally, it didn't seem that their reported error scaled very much with the dimension. We use their reported relative error as a guideline on the performance of our own algorithm.

We notice two major observations in Table 1:

- The DGMT, like traditional PDE methods, shows clear improvement of the error as the number of timesteps increase. With 32 timesteps, the DGMT method shows superior performance compared to
  - The reported performance in [22],
  - The DGM with approximately the same amount of training time, and
  - The DGM running for 100,000 iterations and taking much longer time.
- The original DGM is not very stable. It is unclear if extra training iterations result in a more accurate solution.

Although comparing the loss from the original DGM directly with the DGMT is not fully meaningful due to the different inputs, we nevertheless find that the value of the loss function is a reasonable proxy to the error. We show the direct comparison in Figures 2 and 3 for the loss and error, respectively.

4.2 Improved Stability

An additional benefit of DGMT is that the loss function does not change much per epoch. We show our experimental results of the change in the loss function per epoch in Figure 4 and the change in the error per epoch in Figure 5. A benefit of this is that it is easier to devise a stopping criterion, since

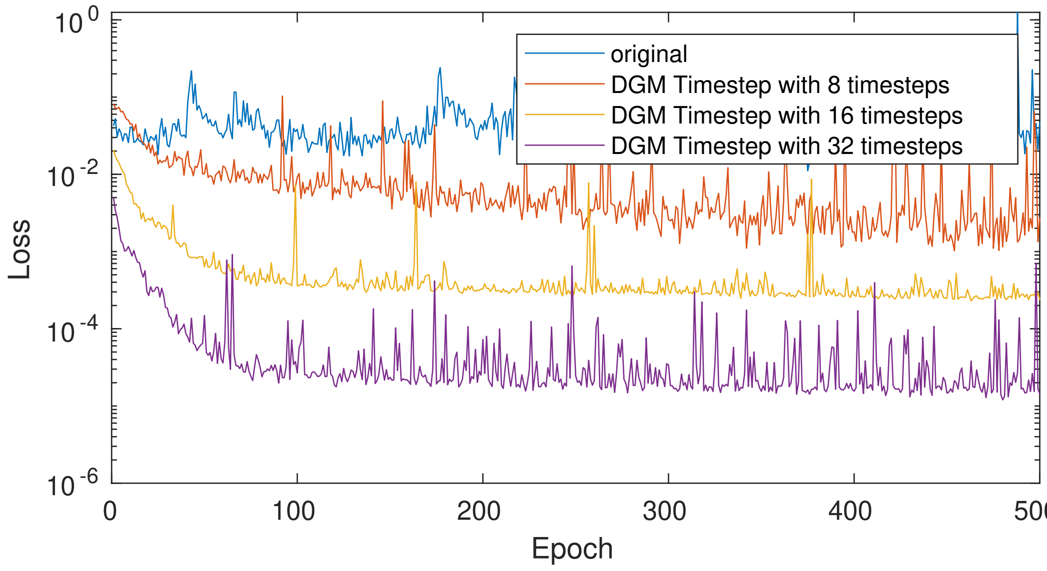

Figure 2: Progression of the loss function per epoch. As the number of timesteps increase, the loss decreases for the DGMT. This is natural because in PDE methods the truncation error is reduced when a method reduces the number of timesteps.

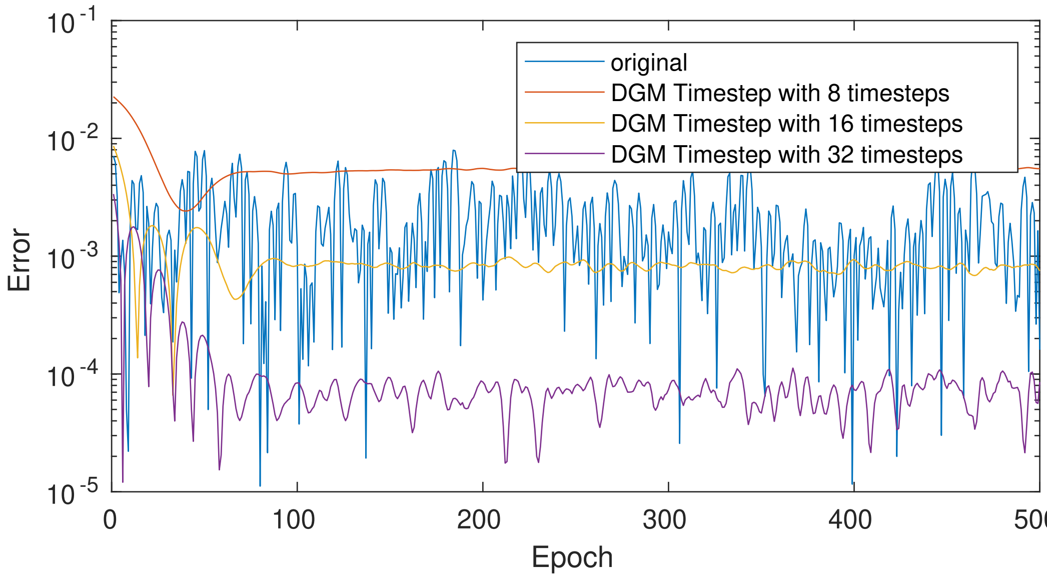

Figure 3: Progression of the pointwise error per epoch. DGMT with larger number of timesteps (and hence smaller timesteps) can achieve higher accuracy in the estimation of the solution.

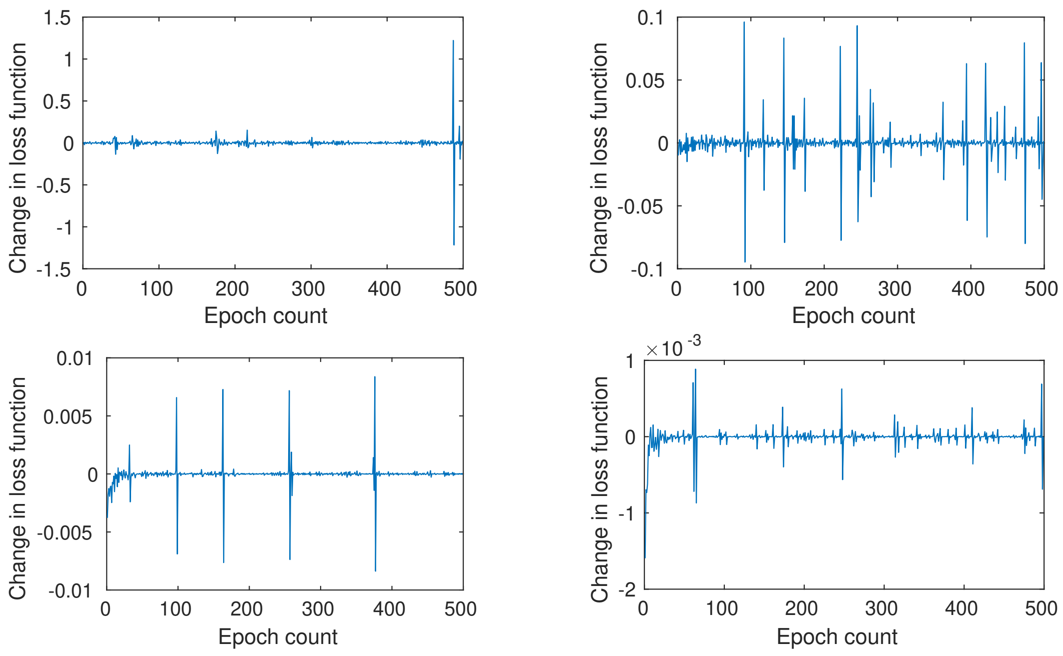

Figure 4: Change of the loss function per iteration. The DGM run is truncated to the last 500 epochs, while the DGMT runs are for the last optimization problem. Top Left: Original DGM method, trimmed to last 500 epochs. Top Right: DGMT, 8 timesteps. Bottom Left: DGMT, 16 timesteps. Bottom Right: DGMT, 32 timesteps. The DGMT in general changes less per epoch, and hence we expect this to cause less of a change in the error. Note in particular the changing scale on the y-axis.

the error clearly plateaus when a certain point is reached compared to the original DGM. We simply point out the possibility and leave this idea for future work.

5 Limitations

There are several limitations to the study done above:

- The first major limitation of the study is that I did not conduct experiments of problems in higher dimensions.
- Another limitation of the study is that I conducted my studies on European options whereas the original authors [22] attempted American options.

Another issue is the fact that as the timesteps decrease (64, 128, etc), the error in the computed solution stops decreasing correspondingly. This may be a limitation in the neural network's ability to approximate the solutions accurately, since the error is from two sources: the first is the semi-discretization of time leads to a solution that is second-order accurate, meaning that there is some truncation error involved which is $\mathcal{O}(\Delta\tau^2)$ in theory. The second source of error is the neural network's inability to exactly match the solution, so there is a fixed approximation error combined with truncation error. As the truncation error decreases we have not increased the size of the neural network, so it may be a case of the approximation error dominating similar to the breakdown of finite difference methods when a very small timestep is taken, and the roundoff error dominates the truncation error.

We conducted an experiment to determine this. By solving the optimization problem

$$\theta^* = \arg\min_{\theta} \left(f(x;\theta) - V(\tau = T, S) \right)^2 \tag{9}$$

to approximate the exact solution $V(\tau = T, \cdot)$ of the PDE, we are able to remove all time-discretization error and study only the approximation error of the neural network to the solution

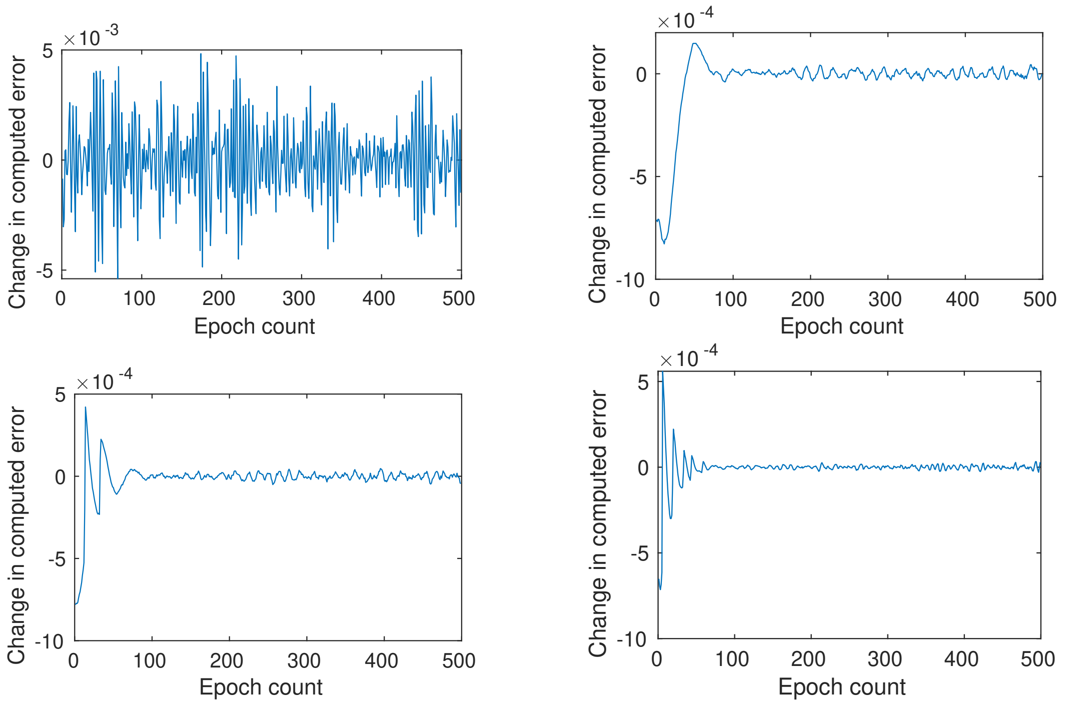

Figure 5: Change of the error per iteration. The DGM run is truncated to the last 500 epochs, while the DGMT runs are for the last optimization problem. Top Left: Original DGM method, truncated to last 500 epochs. Top Right: DGMT, 8 timesteps. Bottom Left: DGMT, 16 timesteps. Bottom Right: DGMT, 32 timesteps. As expected, the DGMT changes less per epoch. Note the changing scale on the y-axis.

function. We found that the error is around 2.48×10^{-4}, which is around the same level that the DGMT method stops at.

Figure 6 shows that the performance reaches a magnitude of around 10^{-4} and does not improve past it independent of any kind of time-discretization, hence it confirms our ideas that the experiments have been limited by the ability of the neural network to approximate the function more accurately when the number of timesteps have been increased.

6 Conclusions

In conclusion, we find that the DGMT attains a more accurate solution in less training epochs and less time compared to the DGM. Additionally, the error arising from the computed solution varies less per epoch. We believe that this property would help us to devise a stopping criterion for the training of the neural network, although due to time limitations this was unable to be completed. Subsequent work in addition to the development of a stopping criterion involve exploring different neural network architectures, testing the method on higher dimensional problems, and testing the method on nonlinear problems, particularly those with nonlinearity in the second-derivative (diffusion) term. It would also be interesting to test the problems with neural networks of larger size and observe if the accuracy of the solution can improve.

References

[1] L. Andersen, J. Andreasen, and R. Brotherton-Ratcliffe, *The passport option*, Journal of Computational Finance, 1 (1998), pp. 15–36.

[2] M. Avellaneda, A. Levy, and A. Paras, *Pricing and hedging derivative securities in markets with uncertain volatilities*, Applied Mathematical Finance, 2 (1995), pp. 73–88.

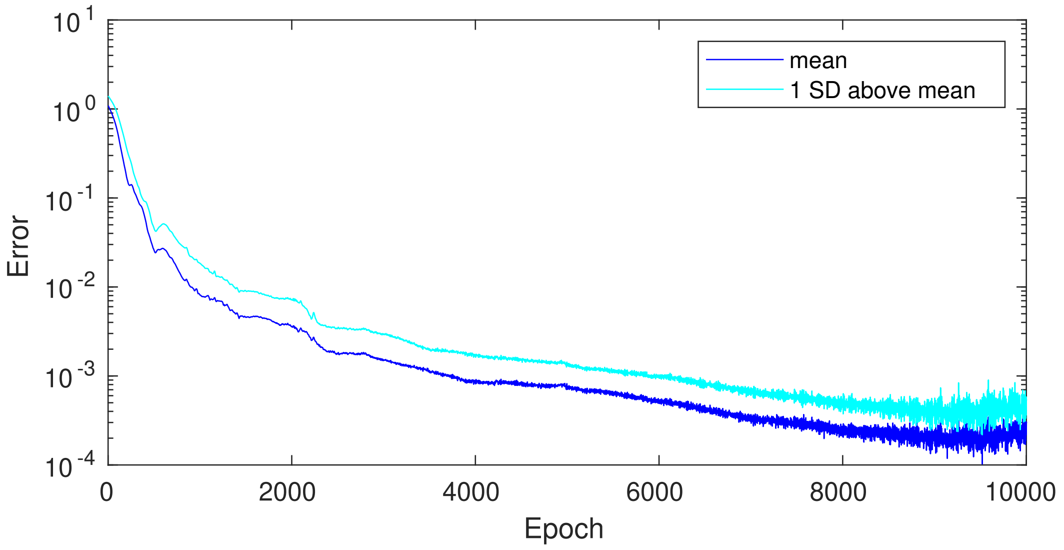

Figure 6: Training history of the DGMT method using the optimization problem (9). We can see that the performance does not improve past a magnitude of 10^{-4}, hence, it is unlikely that any choice of timestepping will lead to a significantly better performance. Average taken over 20 independent runs.

[3] C. Beck, S. Becker, P. Cheridito, A. Jentzen, and A. Neufeld, *Deep splitting method for parabolic pdes*, arXiv preprint arXiv:1907.03452, (2019).

[4] F. Black and M. Scholes, *The pricing of options and corporate liabilities*, Journal of political economy, 81 (1973), pp. 637–654.

[5] J. Crank and P. Nicolson, *A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type*, in Mathematical Proceedings of the Cambridge Philosophical Society, vol. 43, Cambridge University Press, 1947, pp. 50–67.

[6] W. E, J. Han, and A. Jentzen, *Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations*, Communications in Mathematics and Statistics, 5 (2017), pp. 349–380.

[7] P. Forsyth and K. Vetzal, *Quadratic convergence for valuing American options using a penalty method*, SIAM J. Sci. Comput., 23 (2002), pp. 2095–2122.

[8] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of the thirteenth international conference on artificial intelligence and statistics, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[9] P. Grohs, F. Hornung, A. Jentzen, and P. Von Wurstemberger, *A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of Black-Scholes partial differential equations*, arXiv preprint arXiv:1809.02362, (2018).

[10] J. Han, A. Jentzen, and W. E, *Solving high-dimensional partial differential equations using deep learning*, Proceedings of the National Academy of Sciences, 115 (2018), pp. 8505–8510.

[11] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural computation, 9 (1997), pp. 1735–1780.

[12] K. Hornik, M. Stinchcombe, and H. White, *Multilayer feedforward networks are universal approximators*, Neural networks, 2 (1989), pp. 359–366.

[13] M. Hutzenthaler, A. Jentzen, T. Kruse, and T. A. Nguyen, *A proof that rectified deep neural networks overcome the curse of dimensionality in the numerical approximation of semilinear heat equations*, SN Partial Differential Equations and Applications, 1 (2020), pp. 1–34.

[14] Y. M. Kabanov and M. M. Safarian, *On Leland's strategy of option pricing with transactions costs*, Finance and Stochastics, 1 (1997), pp. 239–250.

[15] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[16] H. E. Leland, *Option pricing and replication with transactions costs*, The Journal of Finance, 40 (1985), pp. 1283–1301.

[17] H. E. Leland et al., *Comments on "Hedging errors with Leland's option model in the presence of transactions costs"*, Finance Research Letters, 4 (2007), pp. 200–202.

[18] D. Pfau, J. S. Spencer, A. G. Matthews, and W. M. C. Foulkes, *Ab initio solution of the many-electron Schrödinger equation with deep neural networks*, Physical Review Research, 2 (2020), p. 033429.

[19] R. Rannacher, *Finite element solution of diffusion problems with irregular data*, Numerische Mathematik, 43 (1984), pp. 309–327.

[20] J. Ruf and W. Wang, *Neural networks for option pricing and hedging: a literature review*, Journal of Computational Finance, Forthcoming, (2020).

[21] E. Schrödinger, *An undulatory theory of the mechanics of atoms and molecules*, Physical review, 28 (1926), p. 1049.

[22] J. Sirignano and K. Spiliopoulos, *DGM: A deep learning algorithm for solving partial differential equations*, Journal of Computational Physics, 375 (2018), pp. 1339–1364.

[23] R. K. Srivastava, K. Greff, and J. Schmidhuber, *Highway networks*, arXiv preprint arXiv:1505.00387, (2015).

[24] D. A. Voss and A.-Q. M. Khaliq, *Time-stepping algorithms for semidiscretized linear parabolic PDEs based on rational approximants with distinct real poles*, Advances in Computational Mathematics, 6 (1996), pp. 353–363.

[25] B. Yu et al., *The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems*, Communications in Mathematics and Statistics, 6 (2018), pp. 1–12.