Deep Galerkin Method for solving European-option pricing model

A Project Report Submitted
for the Course

MA699 Project I

by

Akash Debnath
(Roll No. 212123004)

to the

DEPARTMENT OF MATHEMATICS
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, INDIA

APRIL 2023

CERTIFICATE

This is to certify that the work contained in this project report entitled "**Deep Galerkin Method for solving European-option pricing model**" submitted by **Akash Debnath (Roll No.: 212123004)** to the Department of Mathematics, Indian Institute of Technology Guwahati towards the requirement of the course **MA699 Project** has been carried out by him under my supervision.

Guwahati - 781 039
April 2023

(Prof. Natesan Srinivasan)
Project Supervisor

ABSTRACT

The Deep Galerkin Method (DGM) is a novel approach for solving partial differential equations (PDEs) that utilizes deep neural networks to approximate the solution. In this work, we apply the DGM to the European-option pricing model, a popular model in finance that calculates the price of a European option using the Black-Scholes equation.

We demonstrate that the DGM can accurately and efficiently approximate the solution to the Black-Scholes equation, even in cases where traditional numerical methods struggle to converge. We compare the performance of the DGM with that of other numerical methods, including finite difference and Monte Carlo methods, and show that the DGM offers superior accuracy and speed in many cases.

We also analyze the effect of various hyperparameters, such as the number of hidden layers and nodes in the neural network, on the performance of the DGM. Finally, we discuss the potential implications of our results for the broader field of numerical methods in finance and beyond. Overall, our results suggest that the DGM has the potential to be a powerful tool for solving a wide range of PDEs in finance and other fields.

Contents

List of Figures

Chapter 1

Introduction

In this work, we implemented a class of neural networks known as the deep neural network for solving a large class of partial differential equations (PDEs) and PDE systems.The PDEs we take into account have a variety of applications in diverse areas such as option pricing, mean-field game analysis, systemic risk analysis to name a few. The **Deep Galerkin Method** (DGM) is the foundation for the numerical approach which we are going to implement.

The core idea of DGM is to use a deep neural network for approximating the unknown function corresponding to our problem, by suitably choosing the corresponding loss function. The network is trained by minimizing losses associated to the differential operator, the initial/terminal conditions, and the boundary conditions given in the initial value and/or boundary issue, keeping in mind that the function must fulfil a known PDE. The neural network's training data is compiled from a variety of potential inputs to the function and is drawn at random from the area where the PDE is specified. The fact that this method lacks a mesh, in contrast to other widely used numerical methods like finite difference techniques, is one of its important characteristics.

The Main goal of this Project is

1. Give a succinct description of PDEs occurring in the quantitative finance sector, and numerical techniques for addressing them.

2. Give a succinct introduction to deep learning, emphasising the concept of neural networks and describing how to create and use them.

3. Talk about the theoretical underpinnings of DGM with an emphasis on the reasons why it is anticipated that this approach would work well.

4. And also we compare the analytical solution of the Black-Scholes PDE with our implemented Deep Galerkin Method.

We present the findings in a way that emphasizes our own learning process, highlighting our mistakes and the actions we took to address any problems we encountered. The following are some of the key findings:

1. The selection of the sampled random points used for training is the single most crucial component in determining the accuracy of the method because DGM is based on random sampling.

2. Having a priori knowledge about the solution can speed up the implementation and outcomes substantially.

3. Training time is important because neural networks may require more time for training which can be a constraint and performs better just by keeping the algorithm run for a greater time period.

Chapter 2

An Introduction to Partial Differential Equations

In this chapter we discuss about the briefly partial differential equations and it's type and some method to solve.

2.1 Overview

Partial Differential Equations (PDEs):

Partial Differential Equations we can use in many areas in Science , Engineering , Economics and Finance. So now in the context of Finance finding the solution of PDEs is a crucial problems of derivative pricing and optimal investment and optimal execution and many fields. In this part we can tell some introductory topics of Partial Differential Equations and uses their Importance in Quantitative finances.

What is Partial Differential Equations (PDEs):

The main property of a Partial Differential Equation (PDE) is that there is more than one independent variable $x = (x_1, x_2, ...x_n)$ and there is a dependent variable that is an unknown function of these variables $u(x)$. We will often denote its derivatives by subscripts; thus

$$\frac{\partial u}{\partial x} = u_x \tag{2.1}$$

and so on. A PDE is an identity that relates the independent variables, the dependent variable u, and the partial derivatives of u. It can be written as of the k-th order PDEs is

$$F(\mathcal{D}^n u(x), \mathcal{D}^{k-1}u(x), \ldots, \mathcal{D}u(x), x) = 0; \quad x \in \Omega \subset \mathbb{R}^n$$

where $\mathcal{D}^n$ is the collection of the all partial derivative of order n and

$$u : \mathit{\Omega} \to \mathbb{R}$$

is the unknown function that we want to solve.

Now we write a some from of PDEs:

- **Linear PDE:** Derivative of coefficients and function term does not depend on any derivative.

  $$\sum_{|\alpha| \leq n} a_\alpha(x)(\mathcal{D}^\alpha u) = f(x)$$

  where linear in derivatives : $a_\alpha(x)(\mathcal{D}^\alpha u)$ and function term is : $f(x)$.

- **Semi liner PDE:** Coefficients of highest derivatives does not depend on the lower derivatives:

$$\sum_{|\alpha|\leq n} a_\alpha(x)(\mathcal{D}^\alpha u) + a_0\left(\mathcal{D}^{n-1}u,\ldots,\mathcal{D}u,u,x\right) = 0$$

  where linear in the highest order derivatives : $a_\alpha(x)(\mathcal{D}^\alpha u)$ and function term : $a_0\left(\mathcal{D}^{n-1}u,\ldots,\mathcal{D}u,u,x\right) = 0$.

- **Quasi liner PDE:** In a quasi-linear PDE, the coefficients are linear in the highest order derivative but depend on lower order derivatives:

$$\sum_{|\alpha|\leq n} a_\alpha\left(\mathcal{D}^{n-1}u,\mathcal{D}u,u,x\right)(\mathcal{D}^\alpha u) + a_0\left(\mathcal{D}^{n-1}u,\ldots,\mathcal{D}u,u,x\right) = 0$$

  where coefficient term of highest order derivatives : $a_0\left(\mathcal{D}^{n-1}u,\ldots,\mathcal{D}u,u,x\right)(\mathcal{D}^\alpha u)$ and function term does not depend on highest order derivative: $a_0\left(\mathcal{D}^{n-1}u,\ldots,\mathcal{D}u,u,x\right)$.

- **Fully Nonliner PDE:** Relies on highest order derivatives non-linearly.

A collection of various PDEs with numerous unknown functions makes up a system of partial differential equations

$$F\left(\mathcal{D}^n u(x),\mathcal{D}^{n-1}u(x),\ldots,\mathcal{D}u(x),u(x),x\right) = 0,\ x\in\Omega\subset\mathbb{R}^n$$

where $u:\Omega\to\mathbb{R}$. The PDE forms above are generally ranked in descending order of difficulty. Furthermore:

- PDE's with higher orders are more challenging to solve than PDE's with lower orders.
- Solving systems of PDE's is more challenging than solving a single PDE.
- PDE's become more challenging when state variables are added.

In some circumstances, we demand that the unknown solution u should be given in terms of known function on the domain boundaries. Such a circumstance is referred to as a border circumstance (or an initial/terminal circumstance when addressing a time dimension).

Lastly we will discuss some PDE's to take an example and demonstrate that how helpful PDE's are in Financial applications.

2.2 Black Scholes Partial Differential Equation

Two of the most well-known findings in quantitative finance are the Black Scholes Equation and the associated Black Scholes PDEs that are covered in the pioneering work of Black and Scholes (1973). So now, utilising the Black Scholes Method, let's discuss European Style Derivatives. [1]

2.2.1 European Style Derivatives

In mathematical finance, the Black–Scholes equation is a partial differential equation (PDE) governing the price evolution of a European call or European put under the Black–Scholes model. Broadly speaking, the term may refer to a similar PDE that can be derived for a variety of options, or more generally, derivatives. So now we assume a simple market model that is Black Scholes Model with a risky asset follows a Geometric Brownian Motion (GBM) with a constant drift and volatility parameters and rate of interest is constant. so for the dynamics of the price process of a risky asset $X = (X_t)_{t>0}$ and the riskless bank account $B = (B_t)_{t\geq 0}$ and under the real world probability measure P are given by

$$\frac{\mathrm{dX_t}}{X_t} = \mu dt + \sigma dW_t\,,$$

$$\frac{\mathrm{dB_t}}{B_t} = rdt\,,$$

where P-Brownian motion $W = (W_t)_{t\geq 0}$ is represented.

We are interested in determining the cost of a claim that is made against the asset X and has the payout function $F(x)$ and an expiration date T. Dynamic hedging and no-arbitrage arguments can then be used to demonstrate that the claim's price function, $s(t,x)$, which determines the value of the claim at time t while the underlying asset is at level $X_t = x$ is sufficiently smooth:

So the Black Scholes partial differential equation is

$$\frac{\partial s(t,x)}{\partial t} + rs.\frac{\partial s(t,x)}{\partial x} + 1/2\sigma^2 x^2.\frac{\partial^2 s(t,x)}{dx^2} = r.s(t,x),$$

$$s(T,x) = F(x).$$

There are other methods to extend this straightforward model and the associated PDE, for instance

- incorporating more uncertainty sources;
- taking non-traded processes into account as fundamental sources of uncertainty;
- enabling more complex asset price dynamics, such as stochastic volatility and leaps;
- Pricing more intricate reward functions, including path-dependent payoffs, for example.

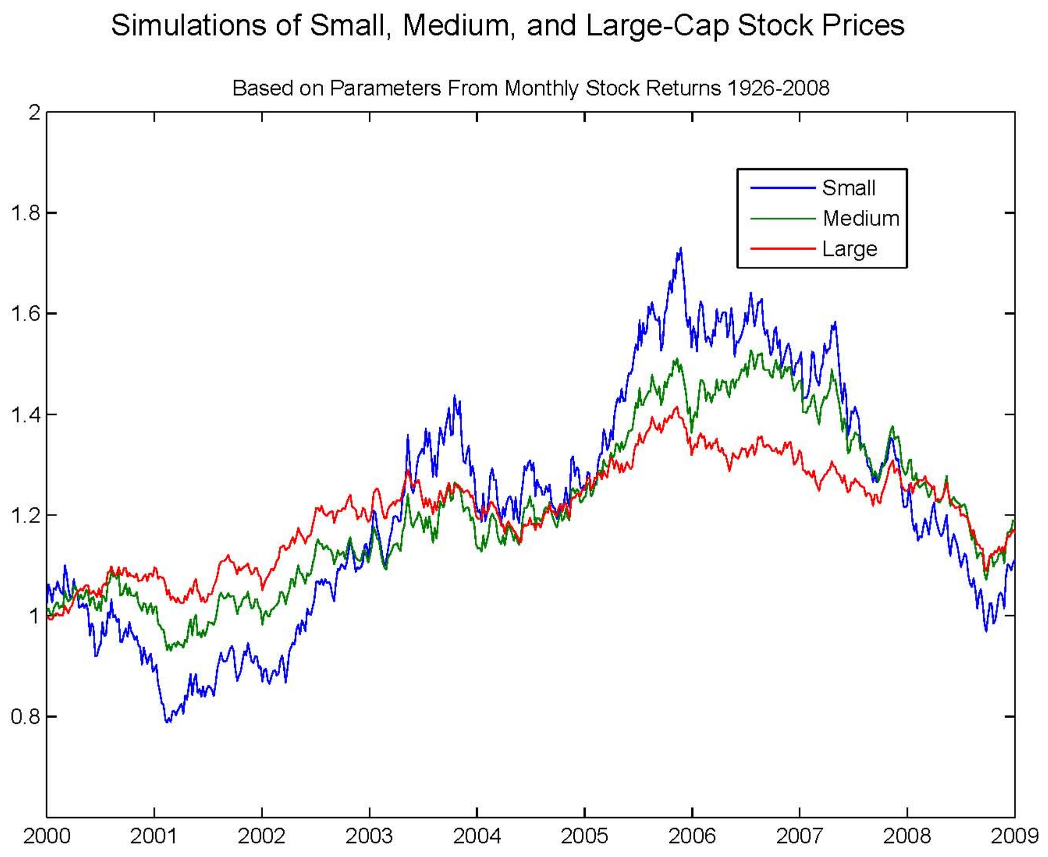


Figure 2.1: Simulated geometric Brownian motions with parameters from market data

Chapter 3

Numerical Methods for Partial Differential Equations

Numerical Methods for Partial Differential Equations is the branch of Numerical Analysis that studies the numerical solution of Partial Differential Equations.So in this chapter, we try to discuss some of the techniques that are taken to solve some of the PDEs numerically. And we also try to see some difficulties in this approach that arise like stability and computational cost in the higher dimension.

3.1 Finite Difference Methods

Finite Difference Methods is a numerical technique that used to solve Differential Equation and the numerical techniques that used to solve approximate solutions but they're much easier to execute than a lot of the techniques for finding an exact solution or formulas for a solution and numerical techniques end up being a really main way that differential equation gets to solve in practices. The idea behind this method is that we replace the given differential operator with its discrete counterpart using the corresponding difference operators and solve the corresponding discretized form to get the unknown function.

3.1.1 Euler's Method

As we know that the easiest finite difference method is Euler's method for Ordinary Differential Equations (ODEs). Suppose we have given an initial value problem

$$\frac{dy(t)}{dt} = g(t),$$

$$y(0) = y_0.$$

so for this type of problem, we try to solve the function $y(t)$ using the Taylor series expansion which we can write it in the given below

$$y(t+h) = y(t) + \frac{dy(t)}{1!dt}h + \frac{d^2y(t)}{2!dt^2}h^2 + \dots$$

for every real-valued function y that is endlessly differentiable. Terms of order h^2 and higher are un-important if h is small enough and the derivatives of y meet certain regularity requirements, thus we can approximate.

$$y(t+h) \approx y(t) + \frac{dy(t)}{dt}h.$$

Note that we may rewrite this equation as follows:

$$\frac{dy(t)}{dt} \approx \frac{y(t+h)-y(t)}{h}.$$

which nearly matches what a derivative is described as

$$\frac{dy(t)}{dt} = \lim_{h\to 0} \frac{y(t+h)-y(t)}{h}.$$

If we go back to the initial issue, we can write $g(t)$, where g is the precise value of $\frac{dy(t)}{dt}$ as

$$y(t+h) \approx y(t) + hg(t).$$

The notation for the discretization scheme generally applied to finite difference methods can now be introduced. Let $\{t_i\}$ be the sequence of approximations of $y(t)$ such that $y_i = y(t_i)$, and let y_i be the sequence of values assumed by the time variable, such that $t_0 = 0$ and $t_{i+1} = t_i + h$. The previous sentence can be rewritten as

$$y_{i+1} \approx y_i + hg(t_i).$$

This enables us, given the value of $y(t_{i+1}) \approx y_{i+1}$, to approximate the value of $y_i \approx y(t_i)$. For any value of $t > 0$, we can find numerical approximations for $y(t)$ using Euler's approach.

3.1.2 Comparing Explicit and Implicit systems

For a straightforward initial value problem, we constructed Euler's approach in the preceding section. Imagine having a problem that is a little different, where the source term g is now a function of both t and y.

$$\frac{dy(t)}{dt} = g(t),$$
$$y(0) = y_0.$$

And the same argument as before we will take the expression for y_{i+1} is written

$$y_{i+1} \approx y_i + hg(t_i, y_i)$$

where y_{i+1} is specifically stated as a term sum that only depends on time t_i. These kinds of schemes are said to as explicit. If we had applied the approximate

$$y_{i+1} \approx y_i + hg(t_{i+1}, y_{i+1}).$$

So now we arrive at the slighty different expression for y_{i+1}

$$y_{i+1} \approx y_i + hg(t_{i+1}, y_{i+1}).$$

Clearly, we can see that the term y_{i+1} is in both sides of the equation and for that, we can't find any explicit formula for $y_(i+1)$ in general and these schemes are called *implicit schemes.* In the general case for each step in the implicit method, we requires to solve for the above case y_{i+1} by a root finding technique like a Newton's Method or other fixed point iteration methods.

Implicit methods are more complicated to use but usually much more stable and so larger time steps can be implemented. Explicit methods are generally known to be numerically unstable for a large range of equations (specially stiff problems) making them unstable for most in a practical situations.

A-stability which assesses the method's stability for the (linear) test equation $\frac{dy(t,y)}{dt} = \lambda y(t)$, with $\lambda < 0$, is a crucial metric of numerical stability for finite difference methods.

While the explicit euler technique is only stable if $|1 + h\lambda| < 1$, which may necessitate using a small number for h if the absolute value of λ is big, the implicit Euler method is stable for all values of $h > 0$ and $\lambda < 0$. A lower number for h is obviously undesirable given that it necessitates a finer grid and increases the computational cost of the numerical approach, all other factors being equal.

3.1.3 Finite Difference Techniques for solving PDEs

As we discussed in the preceding paragraph that the method for solving ODEs in numerical. But the Finite difference method is also used to solve PDEs as well as. Suppose the boundary problem heat equation in one dimension which describes, suppose a rod of length l and the dynamics of heat transfer equation is

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2},$$
$$u(0,x) = u_0(x),$$
$$u(t,0) = u(t,l) = 0.$$

In the differential operator, we approximation using the Forward Difference operator for the Partial Derivative in time and a second order also the second ordered central difference operator for the Partial Derivative in space. In that case, the notation are $u_{i,j} \approx u(t_i, x_j)$, where $t_{i+1} = t_i + k$ and $x_{j+1} = x_j + h$, so we can write the given system of linear eqations

$$\frac{u_{i+1} - u_{i,j}}{k} = \alpha^2 \left(\frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \right)$$

where $i = 1, 2, 3, \ldots, N$ and $j = 1, 2, 3, \ldots, N$.

Assuming that both dimensions have an equal number of discrete points. This illustration is two dimensional, we see the point t_i, x_j is a size $O(N^2)$ two-dimensional grid.

So now for an n-dimensional problem, the grid of size $O(N^n)$ is required. In practice, if the grid size is growing rapidly then the problem is growth exponential.

3.2 Galerkin methods

Galerkin methods are a class of numerical methods for solving partial differential equations (PDEs) by projecting the solution onto a finite-dimensional subspace of functions.

These methods are widely used in engineering and science, particularly in the fields of fluid mechanics, solid mechanics, and electromagnetics.[2] The basic idea of the Galerkin method is to find an approximate solution to a PDE by minimizing the residual error over a finite-dimensional subspace of functions. The subspace is usually chosen to be a space of polynomials or piecewise polynomials, and the approximate solution is represented by a linear combination of basis functions in this subspace.

The Galerkin method is particularly useful for problems that have a variational formulation, such as the Poisson equation and the wave equation. In these cases, the Galerkin method is equivalent to minimizing the energy functional of the PDE, which makes it particularly effective for solving problems in solid mechanics and electromagnetics. The Galerkin method can be extended to include time-dependent problems by using a time-stepping algorithm, such as the Crank-Nicolson method or the backward difference method, to discretize the time variable. Suppose we try to solve an equation $F(x) = y$ for x , where x and y are the members of spaces of function A and B respectively and the function $F : A \to B$ is a function. Suppose also $(\phi)_{i=1}^{\infty}$ and $(\phi)_{j=1}^{\infty}$ form linearly independent bases for A and B . So now by Galerkin Method, an approximation for x will be

$$x_n = \sum_{i=1}^{n} \alpha_i \phi_i$$

where α_i coefficient satisfy the equations

$$\left\langle F\left(\sum_{i=1}^{n} \alpha_i \phi_i\right),\ \phi_j \right\rangle = \langle y,\ \phi_j \rangle$$

where $i = 1, 2, 3, \ldots, N$ and $j = 1, 2, 3, \ldots, N$.

Chapter 4

An Introduction to Deep Learning

In Artificial intelligence (AI) one of the subfield is Deep Learning and also Machine learning. Machine learning focuses on enabling machines to learn from data and improve their performance on a specific task over time. In other words, it is a method of teaching computers to learn and make decisions without being explicitly programmed. Machine learning algorithms use statistical techniques to identify patterns and relationships within datasets and use that knowledge to make predictions or decisions. The field of machine learning has grown rapidly in recent years, thanks to advances in computing power, data storage, and the availability of vast amounts of data. It has applications in a wide range of fields, from natural language processing and computer vision to robotics and self-driving cars. Machine learning has the potential to revolutionize many industries and transform the way we live and work.

Deep learning is a subset of machine learning that involves the use of artificial neural networks to model and solve complex problems. The term "deep" refers to the multiple layers of neural networks that are used in deep learning algorithms to learn and represent data at different levels of abstraction. Deep learning algorithms are designed to recognize patterns in data by processing large amounts of information through multiple layers of interconnected nodes or neurons.

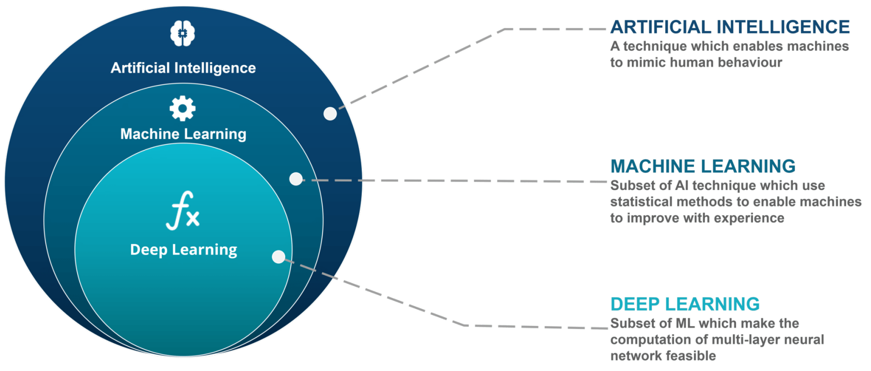

Artificial Intelligence
Machine Learning
f_x
Deep Learning
ARTIFICIAL INTELLIGENCE
A technique which enables machines to mimic human behaviour
MACHINE LEARNING
Subset of AI technique which use statistical methods to enable machines to improve with experience
DEEP LEARNING
Subset of ML which make the computation of multi-layer neural network feasible

Figure 4.1: AI vs ML vs Deep-Learning

Deep learning has been made possible due to advances in computing power, data storage, and the availability of vast amounts of data. Deep learning algorithms have shown remarkable success in a variety of applications, including image and speech recognition, natural language processing, and autonomous vehicles.

The use of deep learning has revolutionized many industries, from healthcare and finance to manufacturing and transportation. It has the potential to improve efficiency, increase accuracy, and enable new innovations that were previously impossible. As more and more data becomes available, and computing power continues to increase, deep learning is expected to become even more powerful and widespread in its applications.

4.1 Applications

Machine learning and deep learning have a wide range of applications across various industries. Some of the most common applications of machine learning include:

1. Image and Speech Recognition: Machine learning algorithms can be used to identify and recognize objects in images and videos, as well as transcribe and translate speech.
2. Natural Language Processing: Machine learning algorithms can be used to analyze and understand natural language, allowing for the development of chatbots and virtual assistants.
3. Fraud Detection: Machine learning algorithms can be used to identify fraudulent activities in real-time, such as credit card fraud or identity theft.
4. Predictive Analytics: Machine learning algorithms can be used to analyze historical data and predict future outcomes, such as sales forecasting, customer churn, or demand forecasting.
5. Recommendation Systems: Machine learning algorithms can be used to personalize recommendations for products and services, based on users' past behaviors and preferences.

Some of the most common applications of deep learning include:

1. Computer Vision: Deep learning algorithms can be used for object detection and classification in images and videos, as well as for autonomous driving.
2. Speech Recognition: Deep learning algorithms can be used to transcribe and translate speech, as well as to recognize different accents and languages.
3. Natural Language Processing: Deep learning algorithms can be used to understand and generate natural language, as well as to analyze sentiment and emotions.
4. Healthcare: Deep learning algorithms can be used for medical imaging analysis, diagnosis, and personalized treatment planning.
5. Robotics: Deep learning algorithms can be used for object detection and localization, as well as for robot navigation and control.
6. Overall, the applications of machine learning and deep learning are virtually limitless, and their potential impact on various industries is significant.

4.2 Neural Networks and Deep Learning

Neural networks are a type of machine learning algorithm that are modeled after the structure and function of the human brain. A neural network is made up of a large number of interconnected processing nodes or artificial neurons that work together to recognize patterns and relationships within data.

Deep learning is a subfield of neural networks that uses multi-layer neural networks to model and solve complex problems. The term "deep" refers to the multiple layers of artificial neurons that are used in deep learning algorithms to learn and represent data at different levels of abstraction.

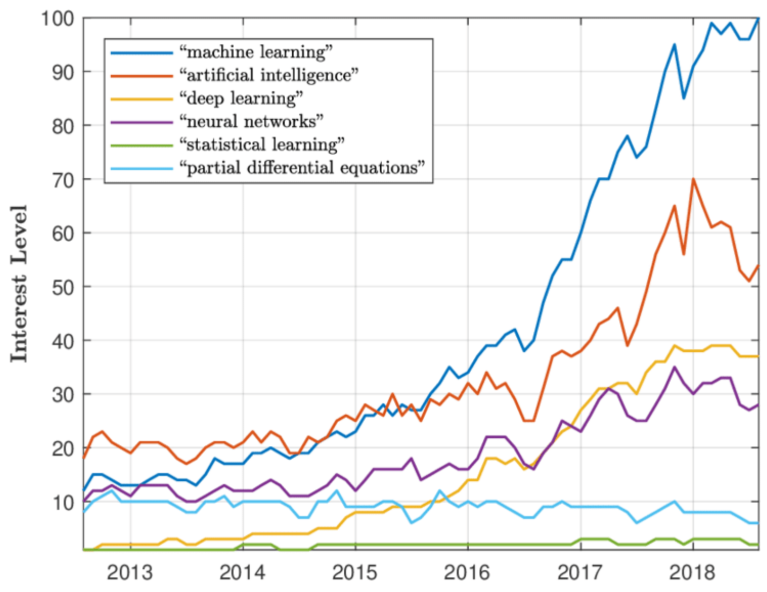


Figure 4.2: Google search frequency for various terms. A value of 100 is the peak popularity for the term; a value of 50 means that the term is half as popular.

The key difference between traditional neural networks and deep learning algorithms is the number of layers involved. Traditional neural networks typically consist of only a few layers, while deep learning algorithms can have dozens or even hundreds of layers.

The ability to use multiple layers allows deep learning algorithms to learn complex representations of data, making them more accurate and effective at tasks such as image and speech recognition, natural language processing, and autonomous vehicles.

Overall, neural networks and deep learning are powerful tools that have the potential to transform many industries, from healthcare and finance to manufacturing and transportation.

A neural network can be mathematically defined as a collection of interconnected processing nodes or artificial neurons that are organized into layers. Each neuron takes one or more inputs, performs a mathematical computation on those inputs, and produces an output.

The inputs to each neuron are typically weighted, meaning that they are multiplied by a learnable parameter known as a weight. The weighted inputs are then summed together and passed through an activation function, which determines the output of the neuron.

The output of each neuron is then fed into the next layer of neurons, creating a hierarchy of interconnected neurons that can recognize patterns and relationships within data.

Mathematically, the output of a single neuron can be defined as:

$$y_j = \phi_j \left(b_j + \sum_{i=1}^{d} w_{i,j} . x_i \right)$$

Where y_j is the output of the neuron, ϕ_j is the activation function, w_i are the weights of the inputs $x_{i,j}$ and b is the baise term. The weights and baise terms are learned through a process called backpropagation, which involves adjusting the parameters to minimize the error between the predicted output and the actual output.

The mathematical equations for neural networks become more complex as the number of layers and neurons increase, For example, with one hidden layer the output would become:

$$y_k = \underbrace{\phi\left[b_k^{(2)} + \sum_{i=1}^{m_2} w_{j,k}^{(2)}.\psi\underbrace{\left(b_i^{(1)} + \sum_{i=1}^{m_1} w_{i,j}^{(1)} x_j\right)}_{\text{input layer to hidden layer}}\right]}_{\text{hidden layer to output layer}}$$

where $\phi, \psi : \mathbf{R} \to \mathbf{R}$ are non-linear activation function for each layer. So now we can extension of this process by chain rule then we visualize that

$$f(x) = \phi_d(....\phi_2(\phi_1(x)))$$

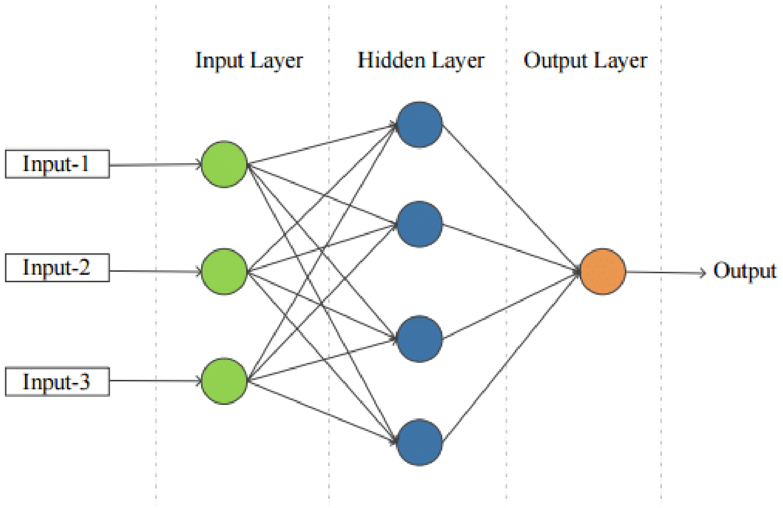


Figure 4.3: Feedforward Neural Network with one Hidden Layer

Here each layer of this network is represented by a function ψ_i incorporating the weighted sums of previous inputs and activations to connected outputs. The number of layers in the graph is referred to as the depth of the neural network and the number of neurons in a layer represents the width of that particular layer; see Figure 4.3.

In deep learning, a loss function is a mathematical function that measures the difference between the predicted output of a neural network and the actual output or target value. The goal of a deep learning model is to minimize this difference, or loss, in order to make accurate predictions.

There are many different types of loss functions that can be used in deep learning, depending on the specific problem being solved. Some common loss functions include:

1. **Mean Squared Error (MSE)**: This loss function calculates the average squared difference between the predicted output and the target value.

2. **Binary Cross-Entropy**: This loss function is used for binary classification problems, and measures the difference between the predicted probability of a positive outcome and the actual outcome.

3. **Categorical Cross-Entropy**: This loss function is used for multi-class classification problems, and measures the difference between the predicted probability distribution and the actual distribution of target values.

4. **Kullback-Leibler Divergence**: This loss function is used for measuring the difference between two probability distributions, and is often used in generative models such as autoencoders and variational autoencoders.

The choice of loss function will depend on the specific problem being solved, and may be influenced by factors such as the distribution of the target values, the type of data being used, and the performance metrics being used to evaluate the model.So now here we first define a loss function $L(\theta; x; y)$ which will determine the performance of a given parameter set θ for r the neural network consisting of the weights and bias terms in each layer. The goal is to find the parameter set that minimizes our loss function.

4.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used for training machine learning models, especially deep neural networks. It is a variant of the Gradient Descent algorithm that updates the model's parameters based on the gradients computed on a small subset of the training data, called a mini-batch, rather than the entire training set.

The basic idea behind SGD is to randomly sample a mini-batch of training data, compute the gradient of the loss function with respect to the model parameters using this mini-batch, and then update the model parameters in the direction of the negative gradient. This process is repeated for many iterations until the model parameters converge to a local minimum of the loss function.

The advantage of using SGD over Gradient Descent is that it can converge much faster because it makes more frequent updates to the model parameters. However, it can also be more noisy because the mini-batch samples are stochastic and may not be representative of the entire training set.

Suppose we have a dataset of N training examples $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$ where xi is the input data and y_i is the corresponding label. Let $f(x; w)$ be a function that maps the input x to a predicted label y_hat using the model parameters w. The goal of training is to find the optimal values of w that minimize the loss function L(w). The Gradient Descent algorithm updates the model parameters w by taking a step in the direction of the negative gradient of the loss function:

$$w_{new} = x_{old} - \eta . \nabla_{w_{old}} L(w)$$

where η is the learning rate, which controls the step size, and $\nabla_{w_{old}} L(w)$ is the gradient of the loss function with respect to w_{old}.

So now the loss function we are minimizing is additive, it can be written as :

$$\nabla L(\theta; x, y) = (1/m) \sum_{i=1}^{m} \nabla_\theta L_i \left(\theta; x^{(i)}, y^{(i)}\right)$$

. In Stochastic Gradient Descent, we randomly sample a mini-batch B of size $m^{'} << m$ from the training set and compute the gradient of the loss function with respect to w using only the examples in the mini-batch:

$$\nabla_\theta L(\theta; x, y) = (1/m^{'}) \nabla_\theta \sum_{i=1}^{m'} L_i \left(\theta; x^{(i)}, y^{(i)}\right)$$

This process is repeated for many iterations until the model parameters converge to a local minimum of the loss function. The main advantage of Stochastic Gradient Descent over Gradient Descent is that it can converge much faster because it updates the model parameters more frequently

4.4 Backpropagation

Backpropagation, also known as backward propagation of errors, is an algorithm used for training neural networks. It is a gradient-based optimization algorithm that calculates the gradient of the loss function with respect to the weights of the network, which is then used to update the weights through the optimization process.

The main idea behind backpropagation is to propagate the error, or the difference between the network's output and the true output, backwards through the network, layer by layer, and compute the gradient of the loss function with respect to the weights in each layer.

Here are the steps involved in backpropagation:

1. **Forward pass**: The input is propagated forward through the network to obtain the output of the network.
2. **Compute the error**: The difference between the network output and the true output is computed, and the error is backpropagated through the network to compute the gradients of the weights.
3. **Backward pass**: The gradients are propagated backward through the network, layer by layer, using the chain rule of calculus.
4. **Weight update**: The weights of the network are updated using the gradients computed in step 3 and a learning rate.
5. Repeat steps 1-4 for multiple iterations until the network converges to a solution.

Backpropagation is a computationally efficient algorithm for training neural networks, and it has been instrumental in the success of deep learning. There have been many extensions and improvements to the basic backpropagation algorithm, including the use of momentum, adaptive learning rates, and regularization, to name a few.

4.4.1 Summary

In summary, the training neural networks are broadly composed of three ingredients:

1. Defining the architecture of the neural network and a loss function;
2. Finding the loss minimizer using stochastic gradient descent;
3. Using backpropagation to compute the derivatives of the loss function.

It is presented in more mathematical detail in below:

---

1. Define the architecture of the neural network by setting its depth (number of layers), width (number of neurons in each layer) and activation functions
2. Define a loss functional $L(\theta; x, y)$ mini-batch size m0 and learning rate η
3. Minimize the loss function to determine the optimal θ:
   (a) Initialize the parameter set, θ_0
   (b) Randomly sample a mini batch of $m^{'}$ training examples $\big(x^{(i)}, y^{(i)}\big)$
   (c) Compute the loss functional for the sampled mini-batch $L(\big(\theta; (x^{(i)}, y^{(i)})\big)$
   (d) Compute the gradient $\nabla_\theta L(\big(\theta; (x^{(i)}, y^{(i)})\big)$ using backpropagation

(e) Use the estimated gradient to update θ_i based on SGD:

$$\theta_{i+1} = \theta_i - \eta.\nabla_\theta L\left(\theta; (x^{(i)}, y^{(i)})\right)$$

(f) Repeat steps (b)-(e) untill $\|\theta_{i+1}\theta_i\|$ is small.

4.5 The Vanishing Gradient Problem

The vanishing gradient problem is a common issue that occurs during the training of deep neural networks using gradient-based optimization algorithms, such as backpropagation. It occurs when the gradients of the loss function with respect to the weights in the lower layers of the network become very small, making it difficult to update these weights and slowing down or even preventing the convergence of the network.

The vanishing gradient problem arises because of the way gradients are computed during backpropagation. The gradients are propagated backward through the network using the chain rule of calculus, and each layer multiplies the gradient it receives by its own weight matrix. If the weights in the matrix are small, then the gradients will also be small, and this effect can accumulate as the gradients are propagated backward through the network.

This problem is particularly acute in deep neural networks with many layers because the gradient can become exponentially smaller as it is propagated backward through the network. As a result, the weights in the lower layers may not be updated effectively, and the network may fail to learn important features or patterns in the data.

There are several techniques to mitigate the vanishing gradient problem, such as using activation functions that have non-zero derivatives in a wide range, such as ReLU (Rectified Linear Units) or Leaky ReLU. Another approach is to use normalization techniques like batch normalization or layer normalization. Additionally, using alternate optimization algorithms such as Adam or RMSProp can help mitigate the problem by adaptively adjusting the learning rates based on the gradients' magnitude. Finally, careful weight initialization and regularization can help prevent the problem from occurring in the first place.

4.6 Long-Short Term Memory and Recurrent Neural Networks

Long-Short Term Memory (LSTM) is a type of recurrent neural network (RNN) that is designed to address the vanishing gradient problem that can occur in traditional RNNs. Like traditional RNNs, LSTMs are designed to process sequences of inputs, such as time-series data or natural language sentences, where the order of the inputs is important.

In an RNN, the hidden state at each time step is a function of the current input and the previous hidden state. This allows the network to maintain a memory of previous inputs and use it to make predictions. However, in traditional RNNs, the gradients can become very small as they are propagated backwards through time, leading to the vanishing gradient problem.

LSTMs address this problem by introducing a set of memory cells that can store information for a long time and selectively forget or update that information based on the current input. This is accomplished through a set of gates that control the flow of information into and out of the memory cells.

The LSTM architecture consists of three gates:

1. **Forget gate**: Determines which information from the previous time step should be forgotten.

2. **Input gate**: Determines which new information should be stored in the memory cells.

3. **Output gate**: Determines which information from the memory cells should be used to produce the output.

Each gate is implemented as a sigmoid function followed by an element-wise multiplication with the current input or previous hidden state. The sigmoid function outputs a value between 0 and 1, representing the amount of information that should be passed through the gate.

LSTMs have been successful in a wide range of applications, including natural language processing, speech recognition, and image captioning, among others. They have been shown to be particularly effective in tasks where long-term dependencies are important, such as speech recognition and language translation

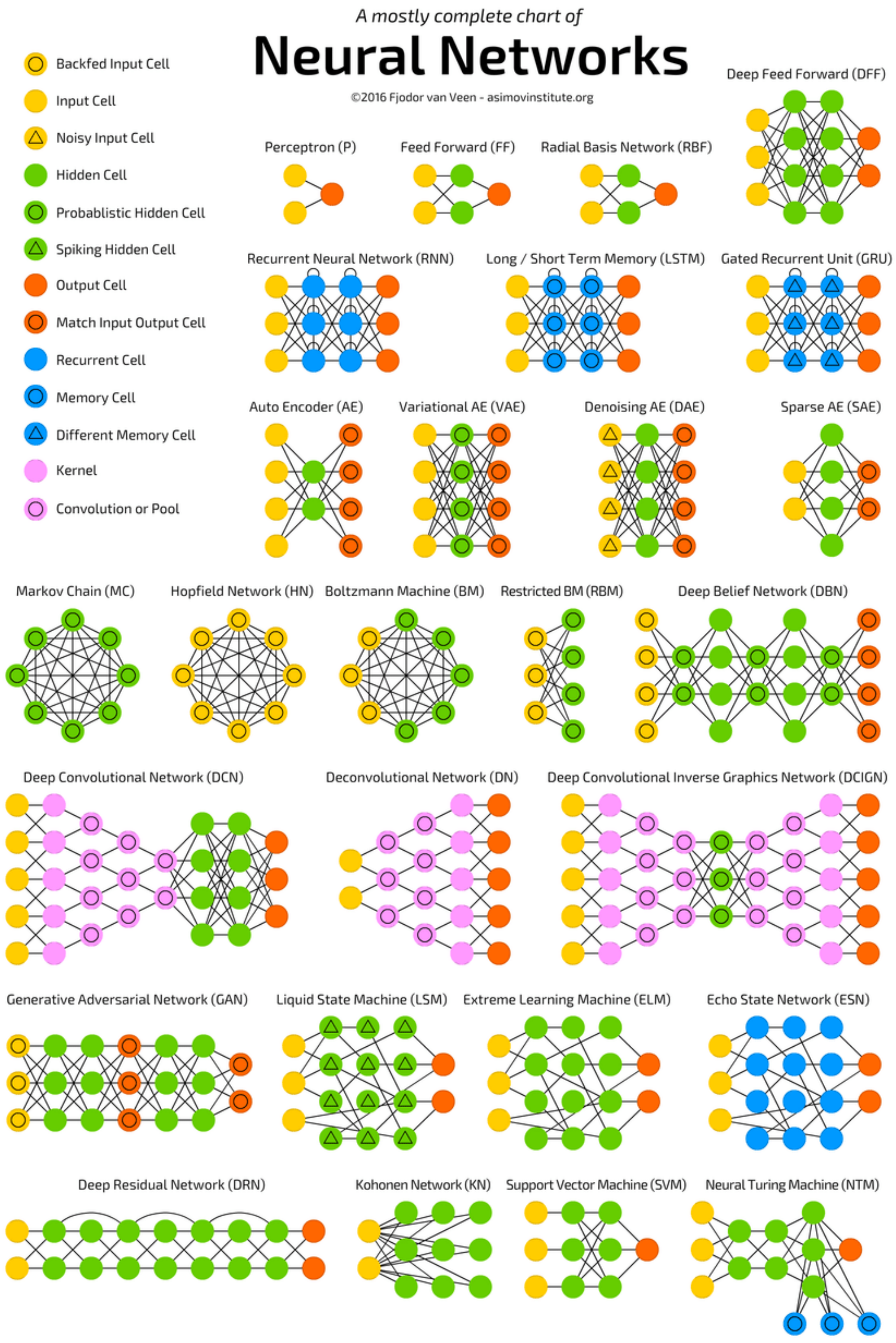

A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

Figure 4.4: Neural network architectures

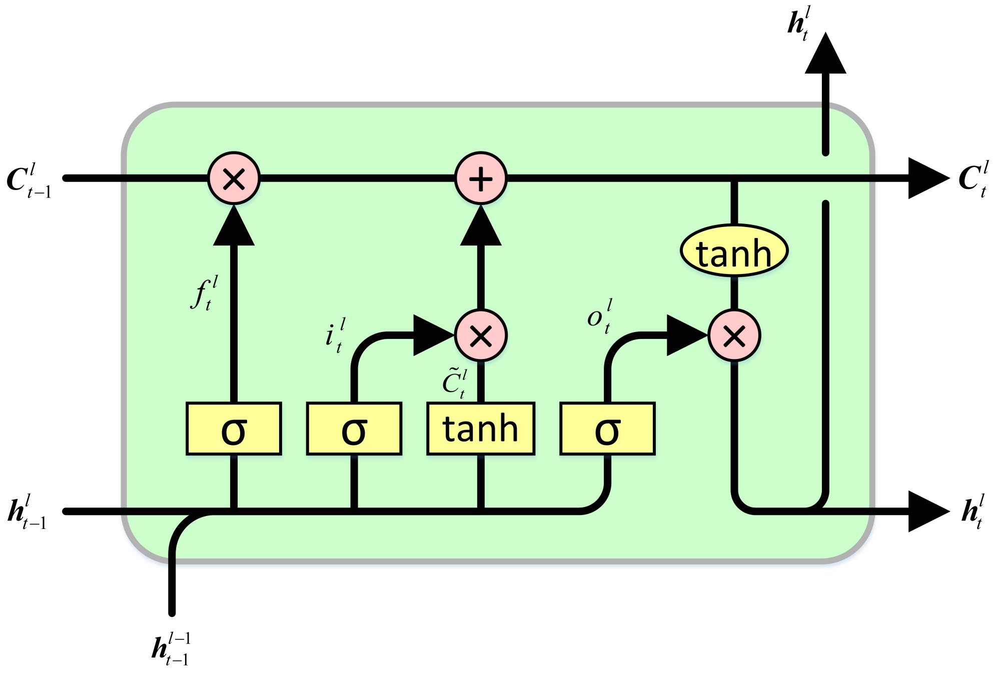


Figure 4.5: Architecture of LSTM units

Chapter 5

Deep Galerkin Method

5.1 Introduction

The Deep Galerkin Method (DGM) is a machine learning approach for solving differential equations. It was proposed in 2018 by Sirignano and Spiliopoulos.

The DGM combines the strengths of deep learning and traditional numerical methods. It uses neural networks to approximate the solution to a differential equation, while still using the Galerkin method to enforce the differential equation's constraints.

In the DGM, a neural network is trained to approximate the solution to a differential equation on a set of discretized points. The neural network takes in the input variables (e.g., time and space) and returns an approximation of the solution. The Galerkin method is then used to enforce the differential equation's constraints on the neural network's output.

Compared to traditional numerical methods, the DGM can be faster and more accurate, especially for high-dimensional problems. However, it requires a significant amount of training data and careful tuning of the neural network architecture and hyperparameters.

The DGM has been applied to a variety of problems, including fluid dynamics, quantum mechanics, and option pricing in finance.

5.2 Mathematical Details

The form of the PDEs of interest are generally described as follows: let u be a unknown function of the time and space defined on the region $[0, T] \times \Omega$ where $\Omega \subset \mathbf{R}$ and u satisfies the PDEs:

$$\left(\frac{\partial}{\partial t} + \mathcal{L}\right) u(t, x) = 0 \quad (t, x) \in [0, T] \times \Omega$$

$$u(0, x) = u_0(x) \quad x \in \Omega \quad \text{(Intial condition)}$$

$$u(t, x) = g(t, x) \quad (t, x) \in 0, T] \times \partial\Omega \quad \text{(Boundary condition)}$$

So now the main goal is approximate u with an approximating function $f(t, x, \theta)$, given by deep neural network with parameter setθ. The loss function for the associate training problem consists of three parts;

1. Measure that how well the approximation satisfies the differential operator,

$$\| \left(\frac{\partial}{\partial t} + \mathcal{L}\right) f(t, x; \theta) \|^2_{[0,T]\times\Omega, v_1}$$

2. Measure of how well the approximation satisfies the boundary condition,

$$\|f(t,x;\theta)-g(t,x)\|^2_{[0,T]\times\partial\Omega,v_2}$$

3. measure of how well the approximation satisfies the initial condition,

$$\|f(t,x;\theta)-u_0(x)\|^2_{\Omega,v_3}$$

In all three terms above the error is measured in terms of L^2 -norm, And combining this three terms gives the total loss and we minimize this loss function.

$$L(\theta)=\|\left(\frac{\partial}{\partial t}+\mathcal{L}\right)f(t,x;\theta)\|^2_{[0,T]\times\Omega,v_1}+\|f(t,x;\theta)-g(t,x)\|^2_{[0,T]\times\partial\Omega,v_2}+\|f(t,x;\theta)-u_0(x)\|^2_{\Omega,v_3}$$

So now we minimize the loss function using Stochastic Gradient Descent(SGD) and more specifically we apply the algorithm that defined in the below.

1. Initialize the parameter set θ_0 and the learning rate α_n
2. Generate random samples from the domain interior and time, spatial boundaries
   - Generate (t_n,x_n) from $[0,T]\times\Omega$ according to v_1
   - Generate (τ_n,z_n) from $[0,T]\times\partial\Omega$ according to v_2
   - Generate w_n from Ω, according to v_3
3. Calculate the loss function for the current mini-batch (the randomly sampled points $s_n=((t_n,x_n),(\tau_n,z_n),w_n)$
   - Compute $L_1(\theta_n;,t_n,x_n)=\left(\left(\frac{\partial}{\partial t}+\mathcal{L}\right)f(\theta_n;t_n,x_n)\right)^2$
   - Compute $L_2(\theta_n;,\tau_n,z_n)=\left(f(\tau_n,z_n)-g(\tau_n,z_n)\right)^2$
   - Compute $L_3(\theta_n;w_n)=\left(f(0,w_n)-u_0(w_n)\right)^2$
4. Take a descent step at the random point s_n with Adam-based learning rates

$$\theta_{n+1}=\theta_n-\alpha_n.\nabla_\theta L(\theta_n;s_n)$$

5. Repeat the steps (2) and (4) until $\|(\theta_{n+1}-\theta_n)\|$ is small.

5.3 A Neural Network Approximation Theorem

Let $L(\theta)$ is the loss function measuring the neural network fit to the differential operator and boundary, initial or terminal conditions.

$\mathbb{C}^{\ltimes}$ the class of neural networks with n hidden units.

$f^n=atgminL(\theta)$, the best n layer neural network approximates to the true PDE solution :

$$f^n\to u \quad \textbf{as} \quad n\to\infty$$

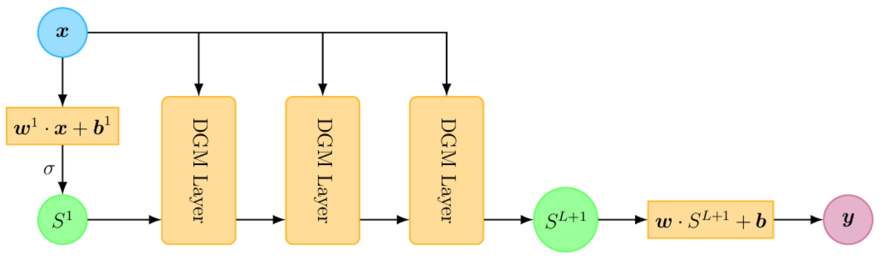


Figure 5.1: DGM Architecture

5.4 Implementation

We used Long-Short Term Memory(LSTM) model for the implementation of DGM Layer. In this layer has an input layer, hidden layer, and output layer.

So now we take the input k mini-batch point on the domain (i.e x in our case and also we chose random normal distribution to take the input on the domain) and also an output of the previous DGM layer. So in this process, we get a vector value output y which consists of the Neural Network and approximates the function of u.

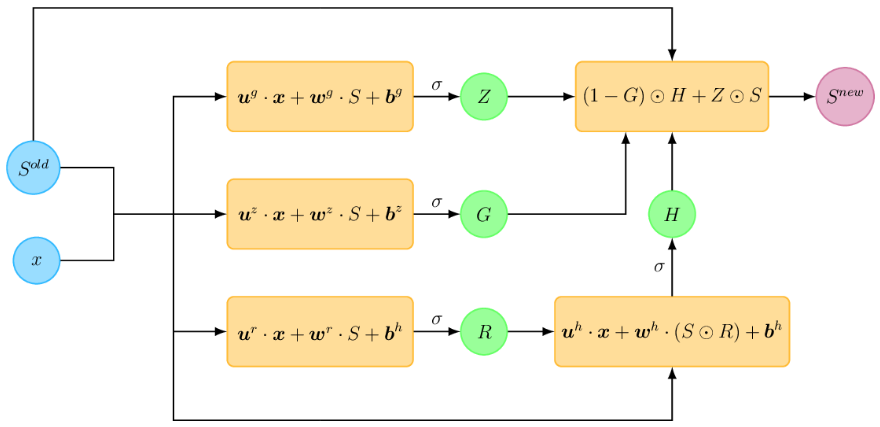
Now we give the equation that we used in Implementation and also give the visual representation.

$$S_1 = \sigma(w_1.x + b_1)$$
$$Z_l = \sigma(u_{z,l}.x + w_{z,l}.S_l + b_{z,l})$$
$$G_l = \sigma(u_{g,l}.x + w_{g,l}.S_l + b_{g,l})$$
$$R_l = \sigma(u_{r,l}.x + w_{r,l}.S_l + b_{r,l})$$
$$H_l = \sigma(u_{h,l}.x + w_{h,l}.(S_l \odot R_l) + b_{h,l})$$

$$S_{l+1} = (1 - G_l) \odot H_l + Z_l \odot S_l$$

$$f(t, x; \theta) = w.S_{L+1} + b$$

where $\odot$ denote element-wise multiplication and L denotes the total number of layer, σ denotes the activation function and x denotes the initial input, w denotes the weight and b is bias terms.[3]



S^{old}
x
$u^g \cdot x + w^g \cdot S + b^g$
$u^z \cdot x + w^z \cdot S + b^z$
$u^r \cdot x + w^r \cdot S + b^h$
σ
Z
G
R
H
$(1-G) \odot H + Z \odot S$
$u^h \cdot x + w^h \cdot (S \odot R) + b^h$
S^{new}

Figure 5.2: Notation each single DGM Layer

Chapter 6

Experimentation and Results

6.1 European Call Option

One-Dimension Black-Scholes PDE

$$\frac{\partial s(t,x)}{\partial t} + rs.\frac{\partial s(t,x)}{\partial x} + 1/2\sigma^2 x^2.\frac{\partial^2 s(t,x)}{dx^2} = r.s(t,x)$$
$$s(T,x) = F(x)$$

Solution

$$s(t,x) = x.N(v_+) - K.e^{-r(T-t)}N(v_-$$

where

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

Suppose you want to price a European call option on a non-dividend paying stock with a strike price of $K = 50$, a current stock price of $S = 50$, a time to maturity of $T = 1$, a continuously compounded risk-free interest rate of $r = 5\%$, and a volatility of $\sigma = 20\%$.

Now we take a sampled uniformly on the time domain and according to a lognormal distribution on the space domain and we also sampled uniformly at the terminal time.

So Finally the DGM network points that create a best-fit line that we see in the below line.

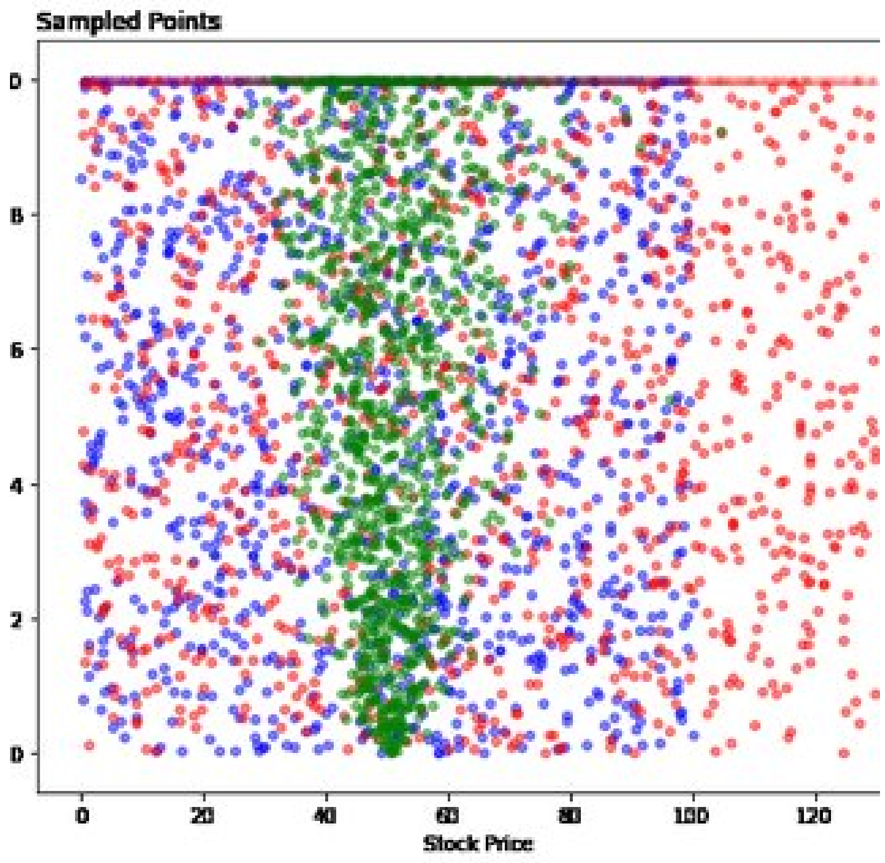


Figure 6.1: Different sampling schemes

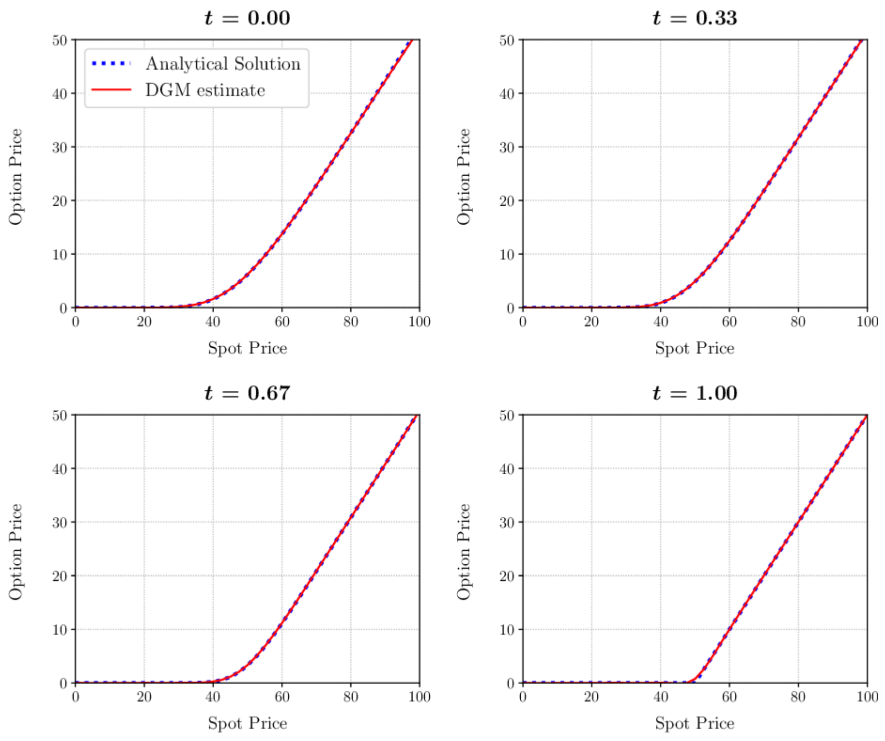


Figure 6.2: European call option prices at different times-to-maturity.

Chapter 7

Conclusions

We have shown that the Deep Galerkin Method(DGM) is a promising approach for solving the European-option pricing model.Our results demonstrate that the DGM can accurately and efficiently approximate the solution to the Black-Scholes equation, but in traditional numerical methods struggle to converge.

Furthermore, we have compared the performance of the DGM with that of other numerical methods, and shown that the DGM offers superior accuracy and speed in many cases. We have also analyzed the effect of various hyperparameters on the performance of the DGM, and provided insights into how to optimize these parameters for maximum performance.

Our findings have important implications for the field of numerical methods in finance and beyond. The DGM has the potential to be a powerful tool for solving a wide range of PDEs, not just in finance, but also in fields such as physics, engineering, and climate modeling.

Overall, the Deep Galerkin Method offers a promising approach for solving complex PDEs that traditional numerical methods may struggle. Its ability to accurately and efficiently approximate solutions makes it a powerful tool for a wide range of applications, and we expect to see continued development and adoption of this method in the future.

Bibliography

[1] UNIMAS Niche Areas and Corporate Memory. Browse by reference.

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

[3] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.