MATHEMATICAL STUDY OF L-BFGS TRAINED NEURAL NETWORKS FOR NON-LINEAR PDE

VEENA VISHWANATH SANGVIKAR

https://orcid.org/0009-0001-2490-6945
Department of Mathematics and Statistics,
Dr. Vishwanath Karad MIT World Peace University, Pune, MS, India. Email: *veena.kshirsagar@mitwpu.edu.in*

Abstract. The architecture of neural networks and their extensive ability to numerically solve the partial differential equations accurately with limited input data points or even no labeled data, has motivated for this research work. The use of neural networks for solving differential equations, partial differential equations, fractional partial differential equations, integral differential equations, stochastic partial differential equations etc, has become a novel methodology of obtaining optimal solutions. The objective of this paper is to use the physics-informed neural network (PINN) for solving the non-linear Burgers' equation and also the mathematical interpretation of the gradient based second order unconstrained convex optimization (L-BFGS) used for training the network is elucidated in detail. PINNs takes the physical information contained in the partial differential equation as a regularization term to improve the networks performance. The well documented PyTorch library of the Python programming language is used for training the network using its automatic differentiation module. The test problem shows the effective implementation of the approach.

Keywords: non-linear partial differential equation, artificial neural network, physics informed neural network, limited memory-BFGS, Burgers' equation, pytorch.

1. Introduction

In the field of scientific machine learning, the use of deep learning techniques to solve partial differential equations has emerged as a promising approach. Partial differential equations (PDEs) play a crucial role in understanding natural phenomena and most of the physical phenomena can be mathematically modelled into a partial differential equation, linear or non-linear, satisfying certain conditions prescribed on boundary of a given domain. However, obtaining analytical solutions for partial differential equations is often challenging, which lead to the development of various numerical methods like finite volume, finite element and finite difference methods. These computational techniques have significantly advanced the study of partial differential equations in fields like aerospace, science and technology, economics, commerce, healthcare, weather prediction and the applications list is endless. Physics-informed neural networks (PINNs) have recently gained significant attention in this domain, as they leverage the power of deep learning while incorporating the underlying physical constraints of the problem [1, 2]. By integrating physical laws into the neural network architecture, PINNs provide a mesh-free alternative to traditional numerical methods, offering advantages in accuracy, efficiency, and flexibility. The first attempt to construct data-efficient and physics-informed learning architectures can be seen in the work by Karniadakis et. al.[3]. The extensions to non-linear problems were proposed by Raissi et. al. [4, 5, 6]. Due gratitute to the authors for their incredible work. By leveraging the impressive function-fitting capabilities of deep neural networks, particularly with Physics informed neural networks (PINNs), this paper successfully delves into the application of deep learning methods augmented with physical constraints to solve partial differential equations such as the convection-diffusion equation.

The convection-diffusion equation is a fundamental partial differential equation which describes the transport of a conserved quantity, such as energy or mass, in a fluid flow. This equation is widely applicable in various fields, including heat transfer, fluid dynamics and mass transport which was the motivation of replacing it with artficial neural network for finding solution with limited data points. A short review of literature shows that recent research has demonstrated the effectiveness of physics-informed neural networks in solving the convection-diffusion equation [1, 7]. These methods involve embedding the governing equation into the loss function of the neural network, ensuring that the predicted solution satisfies the physical constraints of the problem. He et. al. [8] specifically addresses the forward and inverse problems of nonlinear diffusion convection-reaction equations, achieving approximation errors as low as 10^{-5} for forward problems and 10^{-4} for inverse problems. The study highlights the effectiveness of the adaptive activation function in enhancing the

performance of PINNs, validating their practical efficacy in complex nonlinear scenarios. One such approach, presented in [9], employs physics-informed neural networks to learn the generic solutions for multiphase transport in porous media. The authors highlight the importance of combining the physics, conservation laws, and constitutive rules to guide the training of neural networks, which often leads to superior generalization and accuracy. Beguinet et al. in [10], discusses the emergence of deep learning-based numerical schemes, particularly Physically Informed Neural Networks (PINNs), as alternatives to classical numerical methods for solving Partial Differential Equations. It highlights the ease of implementing vanilla versions of PINNs based on strong residual forms and their high approximation capabilities. Shu et al. in [11], highlight the advantages of finite difference-based PINNs (FD-PINNs) over traditional PINNs that utilize automatic differentiation (AD) for discretization. Another study, described in [12], introduces a deep learning library called DeepXDE that enables the application of physics-informed neural networks to solve different types of partial differential equations, including the convection-diffusion equation. The key advantage of physics-informed neural networks is their ability to solve partial differential equations with limited or even no labeled data. These methods rely on the universal approximation theorem of neural networks and their high expressivity to capture the underlying physics of the problem [9, 13]. Moreover, the incorporation of physical constraints can alleviate overfitting issues and reduce the need for large training datasets.

The evolution of PINNs since recent years has been marked by significant advancements in methodology, architecture, and application. These networks have proven to be a versatile and powerful tool for solving non-linear PDEs, offering advantages in accuracy, efficiency, and flexibility over traditional numerical methods. As research continues, further innovations in training strategies, architecture design, and handling of complex PDEs will likely expand the scope and applicability of PINNs in scientific computing.

The plan of the paper further is as follows: In section 2, we study the detailed fundamentals of the generic Artificial Neural Network architecture and the PINNs approach to solve the partial differential equation. The Mathematical formulation of the limited-memory BFGS second order optimizer, from the family of quasi-Newton methods, is discussed in section 3. A test problem, applying the neural network technique and the second order optimizer and the solution approach with code snippets, particularly, solution of the Burgers' equation is discussed in section 4 where prediction and comparison graphs are graphically represented using Python and PyTorch.

2. Fundamentals of PINNs Approach

By and large the building blocks of PINNs approach are composed of broadly three components namely, a neural network, a physics informed network and feedback mechanism (the backpropagation learning algorithm) [14, 15]. Essentially for solving a partial differential equation this can be viewed as steps involving the generation of training data, defining a neural network architecture, defining the model loss function, specifying training options by selecting a reliable and relevant optimization algorithm, then train the neural network, proceed to evaluation and results post processing. The break down discussions of each component follows ahead.

2.1. Artificial Neural Network (ANN) Architecture.

The ANN replicates the human brain neurons which intend to stimulate and process the information from neuron cells in brain and model different networking as per various connections. They are of great importance in the field of Artificial Intelligence for computationally solving partial differential equations and most preferred architecture amongst the Deep Learning research fraternity because of their universal approximation property in the mathematical theory of ANN, which states that any continuous function can actually be arbitrarily closely approximated by a multi-layer perceptron with only one hidden layer with finite number of neurons [16]. Weights are assigned to connections between different neurons representing the amount of impact a neuron of one layer has on the neuron of succeeding layers.

Although neural networks easily demostrate any complex functions compactly, the exact number of layers, neurons and perhaps precise count of parameters (biases and weights) needed to solve any particular partial differential equation using an appropriate ANN architecture could be difficult and challenging. The prominent neural network architectures explored in the Deep Learning taxonomy include FFNN: fully connected feed-forward neural networks, CNN: convolutional neural networks, RNN: recurrent neural networks, AE: auto-encoder, DBN: deep belief network, GAN: generative adversarial network and BDL: bayesian deep learning. In this paper, the feed forward neural network [17] is used which is also referred as multi-layer perceptron(MLP) with several hidden layers moving only forward, while the learning of the neural network is done by the most popular learning algorithm backpropagation [18]. The output from neurons of one layer are used as input to the neurons of next layer, where adjacent layers being connected and neurons within a single layer are not internally linked.

We start by establishing some nomenclature. We label l to name a layer out of total L layers. Let m^l be the number of inputs to the l^{th} layer and n^l be the number of outputs from the l^{th} layer. Then the associated weight and bias matrices

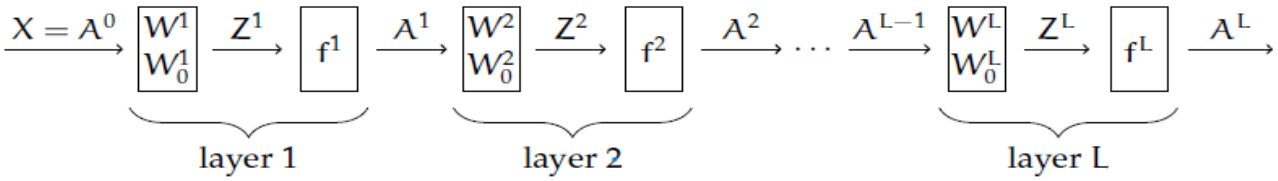


Figure 1. Structural decomposition of the network

W^l and W_0^l are of shape $m^l * n^l$ and $n^l * 1$ respectively. Let f^l be activation function of layer l throwing out the activated output as A^l where $A^l = (a_1^l, a_2^l, \ldots a_n^l)^T$. Then, the pre-activation outputs are the $n^l * 1$ vector:

$$Z^l = W^{l^T} A^{l-1} + W_0^l$$

and the activation outputs are simply the $n^l * 1$ vector

$$A^l = f^l(Z^l)$$

Figure 1 on page 3 shows the structural decomposition to organize our algorithmic thinking and implementation. Each layer has two blocks, one representing the linear part of the operation and one the non-linear activation function.

Mathematically, a comprehensive deep neural network having L layers can be regarded as composition of L functions $\mathbf{f}_i(x_i, W_i)$ where x_i denotes the known input variables and W_i the set of weights and biases for the i^{th} layer. So, any function say $u(x)$ can be approximated as,

$$u_W(x) = \mathbf{f}_L \circ \mathbf{f}_{L-1} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1(x) \tag{1}$$

where $\mathbf{f}_i \in V_i \times T_i$ for V_i and T_i are two inner product spaces and the composition $\circ$ as $\mathbf{f}_2 \circ \mathbf{f}_1(x) = \mathbf{f}_2(\mathbf{f}_1(x))$. Further, every neuron of all layers (except the input) is a mathematical operation that uses an activation function and applies it to the weighted sum of its own inputs and added to a bias factor. Given an input $x \in \Omega$, a multi-layer perceptron passes it through layers of neurons where the non-linear activation functions are applied within neurons and each layer is composed of affine linear maps between neurons, thus forming composition of functions which then transforms it to an output. So, equation (1) can be specified as

$$\mathbf{f}_i^l(x_i, W_i^l, W_0^l) = \sigma_i^l(W_i^l x_i + W_0^l)$$

equipping each V_i and T_i with standard Euclidean inner product ie., $V = T = \Re$ and σ_i stands for a scalar (non-linear) activation function which could be picked up from the several activation functions prescribed particularly for neural networks in the Machine Learning literature. The Tanh function, sigmoid function, rectified linear unit function (ReLU), softmax function, softplus function, are a few to mention. Thus, a neural network with L layers consists of an input layer, an output layer and $(L-2)$ hidden layers.

2.2. Physics Informed Neural Networks (PINNs). Usually, a large quantity of data is required to train a traditional neural network to gain a desirable accuracy of the model performance as they are entirely based on data-driven approach and do not take into account the physical laws contained in data. On the contrary, PINNs addresses problems having fewer data or noisy experiment observations, adhering to any given physical information during training process, specified by a general nonlinear partial differential equation. Specifically, adding a regularization about the partial diferential equation to the loss function, speeds up the training process and requires less data. In this study, physics informed neural network approach is introduced for the forward problem. This approach can even be used to solve the inverse problems where the parameters of the partial differential equation are to be obtained from the whatsoever limited training data available.

Consider the most general form of a partial differential equation defined on domain Ω with boundary $\partial\Omega$ as

$$\mathbf{F}(u(X), \mu) = \hat{g_1}(X) \ for X \in \Omega \subset \Re^d \tag{2}$$

$$B(u(X)) = \hat{g_2}(X) \ for X \in \partial\Omega \tag{3}$$

where $\mathbf{F}$ represents the non-linear partial differential operator, u represents unknown solution, $X = [x_1, x_2, \ldots, x_{d-1}, t]^T$ is the space time coordinate vector, μ the parameter vector related to physics and $\hat{g_1}$ function relates data of problem. Operator B indicates the initial or boundary conditions where the initial condition can be taken as Dirichlet boundary condition on the spatio-temporal domain and $\hat{g_2}$ the boundary function. Here, the boundary conditions could be any of Dirichlet, Neumann, Robin or periodic boundary conditions.

We first construct neural network for computationally approximating the solution $u(X)$ of a partial differential equation.

This network is denoted by $\hat{u}_\Theta(X)$, whose input is the X vector, Θ represents the neural network parameters and this network outputs a vector of same dimension as actual solution $u(X)$. The parameter Θ is actually a collection of the weight matrix: $W^l \in \Re^{n^l * n^{l-1}}$ and bias vector: $b^l \in \Re^{n^l}$ for each layer l with n^l neurons, where Θ is continuously optimized during the training phase. The network $\hat{u}_\Theta$ should be capable of reproducing the observations when X is used as input and also should conform the underlying physics of the partial differential equation. We thus, define a residual network F to meet the requirements of the second part as:

$$F(X,\Theta) := N[\hat{u}_\Theta(X)] \tag{4}$$

To construct this extension of the neural network, many researchers commonly use the automatic differentiation (AD) techniques [19], which have been largely integrated into many Deep Learning frameworks such as PyTorch [20] and Tensorflow. Here, for the surrogate network F, we derive the network by AD according to the chain rule. Further, as the networks F and $\hat{u}_\Theta$ have the same parameters, both networks are trained by minimizing a loss function.

Training refers to finding the best neural network parameters by minimizing the defined loss function. So, if we assume Θ^* to be the best parameter vector, then we are searching Θ^* that minimizes our loss function $J_{pinn}(\Theta)$, ie.,

$$\Theta^* = arg_\Theta min\,[J_{pinn}(\Theta)]$$

For a partial differential equation, the total loss function $J_{pinn}(\Theta)$ is the sum of following loses in general.

$$J_{pinn}(\Theta) = w_B L_B(\Theta) + w_F L_F(\Theta) + w_{data} L_{data}(\Theta) \tag{5}$$

where, $L_F(\Theta)$ is the loss due to the pde, $L_B(\Theta)$ is loss due to the boundary conditions, $L_{data}(\Theta)$ is loss due to some more known additional data given, if any, and w_B, w_F and w_{data} are adequate weights associated with each type of loss respectively. Further, each loss function can be estimated by different metrics as per the type of learning. Here, we use the mean squarred error (MSE) metric [21]. Furthermore, the optimization for equation (5) is to seek the parameters: W^* : vector of weights and b^* : vector of biases.

$$W^* = arg_{W\in\Theta} min[J(W)] \tag{6}$$

$$b^* = arg_{b\in\Theta} min[J(b)] \tag{7}$$

In this study, we use a gradient based optimizer to minimize the loss function. Particularly, a second order derivative based algorithm named limited-memory BFGS (L-BFGS), that allows us to know both which direction to move and also how far to move in that direction. Other first order optimization algorithms include gradient descent, stochastic gradient descent, mini-batch gradient descent, momentum gradient descent, Adagrad, Adadelta, RMSprop and Adam [22].

3. Mathematical Interpretation of L-BFGS Optimizer

A good optimization setting requires not only the gradient $\bigtriangledown f$ of the function f to be optimized, but, we also need to have some inferences about the second derivative ("Hessian") by observing how $\bigtriangledown f$ changes for many different points of x, as incorporating this info in our optimization problem can speed up the predicted solution's convergence. This leads to "quasi-Newton" methods and BFGS update is the most widely used method to approximate the Hessian when the second order derivatives cannot be calculated for the optimization problems. The BFGS algorithm is a local search algorithm which is intended for convex optimization problems with a single optima [23, 24]. So, here's presenting a detailed mathematical interpretation of the basic ideas underpinning the foundations of L-BFGS optimizer.

3.1. Newton Steps: Our problem is to find $min_{x\in\Re^d} f(x)$, (analogous to optimizing loss function $f(x)$) where we have efficiently computed both $f(x)$ and $\bigtriangledown f(x)$ for different data points of x. (Note that in our previous section discussion J is the loss function to be optimised.)

On the k^{th} step of optimization, the iterate is x^k and let $\bigtriangledown f|_{x^k} = g^k$. Now, the second derivative is the "Hessian matrix say H^k," $\left(H^k_{ij} = \frac{\partial f}{\partial x_i \partial x_j}|_{x^k}\right)$ which can be derived from the second-order (quadratic) Taylor expansion of $f(x^k+\alpha)$ given by, $f(x^k+\alpha) \approx f(x^k) + \alpha^T g^k + \frac{1}{2}\alpha^T \alpha H^k = q(\alpha)$ *say*. H^k is positive-definite near a local minima. While, the minimum of $q(\alpha)$ is

$$\alpha = -(H^k)^{-1} g^k.$$

Infact, when the gradient near x^k is approximated by the first-order Taylor expansion, $\bigtriangledown f|_{x^k+\alpha} \approx g^k + H^k\alpha$, the Newton step obtained above will match exactly with the step when computing a root of $\bigtriangledown f = 0$. But now the issue is that the Newton step α, thus obtained, may tend to be immense, leading to inaccuracy in our Taylor expansion and hence further badly impact $f(x^k+\alpha)$. Here's few practised approaches to fix this issue.

1. **Trust Region**: The minimization of $f(x^k+\alpha)$ (or $q(\alpha)$) is done in a "trust region," ie., region for α^k sufficiently small. (Lets say we consider the trust region to be $||\alpha||_2 < r^k$ which is a spherical region for some radius r^k. Here we can optimize a convex dual problem as a strong duality holds in this case. The radius is repeatedly altered until a new step value is acceptable.)
2. **Line Search**: The minimization of $f(x^k+\varepsilon\alpha)$ can be done along the Newton step: α direction, over $\varepsilon \in \Re$. Ingeneral, this exact minimization is complex and one can resort to an inexact line search, that is to keep trying a different ε as long as the result is acceptable.
3. **Backtracking**: We can also try minimization of $f(x^k+\varepsilon\alpha)$ for $\varepsilon = 1, \tau, \tau^2, \tau^3, \ldots$ instead of the exact line search, until the result is acceptable, where $0 < \tau < 1$ is some parameter.

In above approaches to find a suitable Newton step α, both a trust region and backtracking line search requires a decision about acceptance or non-acceptance of given step α. Simply, we can check if $f(x^k+\alpha) < f(x^k)$, although more stronger conditions are to be imposed. That is, we want to choose only those values of step α which lead to reasonable accuracy for quadratic approximation $q(\alpha)$. Hence, the **"Wolfe Conditions,"** one or both of them, are imposed on step α, which are given by:

1. $f(x^k+\alpha) \leq f(x^k)+c_1\alpha^T g^k$ where $0 < c_1 < 1$, usually $c_1 = 10^{-4}$: f must decrease at least proportional to the prediction of the gradient g^k. (Theoretically this means the value of $f|x^k+\alpha$ must be less than its first order Taylor expansion at x^k.)
2. $|\alpha^T g^{k+1}| \leq c_2|\alpha^T g^k|$, where $g^{k+1} = \bigtriangledown f|_{x^k+\alpha}$ and $0 < c_2 < 1$, the derivative $\alpha^T \bigtriangledown f$ along the search direction must decrease sufficiently. (Note that for an exact line search $g^{k+1} = 0$). This condition prevents the inexact line search or the trust region methods from taking too small α values. It also directs to BFGS updates and its acceptable properties, discussed below.

3.2. Newton Steps Update: Quasi-Newton Method: The Newton step method gave us the step size $\alpha = -(H^k)^{-1}g^k$ and hence the new iterated value of x is $x^{k+1} = x^k+\alpha$. The constraint is that, for large n, the exact Hessian is difficult to compute (costs atleast n times the cost of evaluating f once which corresponds to computing the gradient n times: one more gradient for each component of $\bigtriangledown f$). Infact, storing even just the H matrix (n^2 numbers) may be critical and almost impossible for large n. So, the "quasi" Newton methods use a low-rank "approximate Hessian H^k," while applying same Newton steps. Infact, we needed $(H^k)^{-1}$ in the Newton step and so one can store a "low-rank approximate inverse Hessian." For generating this matrix, we construct H^{k+1} [or $(H^{k+1})^{-1}$] iteratively, given only the gradient of f.

In order to obtain the new approximate to Hessian H^k, the original H^k must possess few desired properties which are:

1. As $k \to \infty$, H^k should tend to the exact Hessian for a convex quadratic function $q(\alpha)$.
2. **Secant Condition**:
$$g^{k+1} - g^k = H^{k+1}[x^{k+1} - x^k]$$
3. Real-symmetric positive-definite: This makes the $q(\alpha)$ function convex and $\alpha^k = -(H^k)^{-1}g^k$ is in the down hill direction from x^k.
4. If $q(\alpha)$ is not exactly quadratic then the secant conditions cannot be imposed on all steps at a time as it would become nearly impossible. And hence, a better H^k must "remember" as much information as possible from the previous gradients and steps.

The uncertainity of the last desired property of H^k has a wide scope of developing more quasi-Newton algorithms. To deal with the information remembering property that H^k must possess, we resort to much easier and powerful technique which is to simply minimize $||H^{k+1} - H^k||$ in some norm. One can also minimize $||(H^{k+1})^{-1} - (H^k)^{-1}||$ alternately. This leads to an update in the quasi-Newton approach called as "BFGS" update discussed in the next section.

3.3. BFGS Updates and Low-storage quasi-Newton: This update named for Broyden, Fletcher, Goldfarb and Shanno uses the gradient of the cost function to approximate the Hessian matrix and is obtained by solving the optimization problem:

$$min_{H\in\Re^{n*n}}||H^{-1} - (H^k)^{-1}||_W \tag{8}$$

$$subject\ to\ H^{-1}[g^{k+1} - g^k] = \alpha\ and\ H^T = H \tag{9}$$

ie., we minimize the change in H^{-1} subject to the condition that it be real-symmetric. Here, $||\ldots||_W$ is a weighted Frobenius norm that provides a way to measure the size or magnitude of a matrix and is of vital importance in the Deep Learning literature to deal with matrices having large numbers. Applying the Frobenius norm to the gradient matrix helps prevent exploding or vanishing gradients by keeping them within a reasonable range and hence preventing instability

during training.

Consider $E = H^{-1} - (H^k)^{-1}$. The requirement that the previous iterate H^k should be symmetric, equivalently, leads to following optimization problem.

$$min_{E \in \Re^{n*n}} ||E||_W^2 \tag{10}$$

$$subject\ to\ E\gamma = r\ and\ E^T = E \tag{11}$$

where $\gamma = g^{k+1} - g^k$ and $r = \alpha - (H^k)^{-1}\gamma^2$. This is infact a Quadratic Programming in optimization where the objective function is minimization of a convex quadratic function and the constraints are affine type. We can solve its dual which is given by the Lagrange dual problem as a strong duality holds here, which in turn is equivalent to solving the Karush-Kuhn-Tucker (KKT) conditions. And, by choosing a right weight matrix W, we get a good E update. In the non-linear programming, for a solution to be optimal, the KKT conditions can be mathematically interpreted as first order derivative necessary tests with certain conditions imposed. Only equality constraints are allowed in the Lagrange multipliers method whereas KKT is more generalised approach to it allowing inequality constraints. Infact, various methods for solving the KKT system of equations and inequalities numerically, give rise to different optimization algorithms.

Consider applying the Lagrange Dual problem. We define Lagrange multipliers $\lambda \in \Re^n$ for the $E\gamma - r = 0$ constraint and $\Gamma^T \in \Re^{n*n}$ for the $E - E^T = 0$ constraint to obtain the Lagrangian as:

$$L(E, \lambda, \Gamma) = tr\left[\frac{1}{2}WEWE^T + (E\gamma - r)\lambda^T + \Gamma(E - E^T)\right]$$

Here, note that

$$\begin{aligned} tr[(E\gamma - r)\lambda^T] &= tr[\lambda^T(E\gamma - r)] \\ &= tr\Big[[\lambda_1, \lambda_2, \ldots \lambda_n]_{1*n}\ [n\,constraints\,of \\ &\qquad\qquad (E\gamma - r = 0)]_{n*1}\Big] \\ &= [output]_{1*1} \\ &= [\lambda^T(E\gamma - r)]_{1*1} \end{aligned}$$

which is ordinary sum of n constraints times the n Lagrange multipliers λ_i. We are able to combine it with the $||E||_W^2$ trace by re-ordering it into a rank 1 matrix. Also, note that

$$\begin{aligned} tr[\Gamma(E - E^T)] &= \Sigma_{i,j}\Gamma_{ji}(E - E^T)_{ji} \\ &= \Sigma_{i,j}(\Gamma^T)_{ij}(E - E^T)_{ij} \end{aligned}$$

which is a Frobenius inner product of the n^2 Lagrange multipliers $(\Gamma^T)_{ij}$ with n^2 constraints from $E^T = E$. Further, recall that

$$\bigtriangledown_B[tr(BC)] = \bigtriangledown_B[\sum_{i,j} B_{ij}Cji] = C_{ji} = C^T,$$

where $\bigtriangledown_B = \frac{\partial}{\partial B_{ij}}$ is the matrix of partial derivatives, and similarly, $\bigtriangledown_B[tr(B^TC)] = (C^T)^T = C = C_{ij}$. We can now solve the KKT conditions

$$\begin{aligned} \bigtriangledown_E L = 0 \ \Rightarrow\ & WEW + \lambda\gamma^T + \Gamma^T - \Gamma = 0 \\ & subject\,to\,constraints\ E\gamma - r = 0 \\ & E^T - E = 0 \end{aligned} \tag{12}$$

Equation (12) gives

$$E = -W^{-1}[\lambda\gamma^T + \Gamma^T - \Gamma]W^{-1} \tag{13}$$

Now the requirement that $E = E^T$ would then mean that

$$\begin{aligned} [\lambda\gamma^T + \Gamma^T - \Gamma] &= [\lambda\gamma^T + \Gamma^T - \Gamma]^T \\ &= \gamma\lambda^T + \Gamma - \Gamma^T \\ \Rightarrow\ \Gamma^T - \Gamma - \Gamma + \Gamma^T &= \gamma\lambda^T - \lambda\gamma^T \end{aligned}$$

$$\Rightarrow \Gamma^T - \Gamma = \frac{1}{2}\gamma\lambda^T - \lambda\gamma^T$$

Substituting in (13) gives

$$E = -\frac{1}{2}W^{-1}[\gamma\lambda^T + \lambda\gamma^T]W^{-1} \tag{14}$$

Also, the condition $E\gamma = r$ now implies

$$-\frac{1}{2}\gamma\lambda^T W^{-1}\gamma - \frac{1}{2}\lambda\gamma^T W^{-1}\gamma = Wr$$
$$\gamma\lambda^T W^{-1}\gamma + \lambda(\gamma^T W^{-1}\gamma) = -2Wr$$

We solve this for λ to get,

$$\lambda = -\frac{[2Wr + \gamma\lambda^T W^{-1}\gamma]}{\gamma^T W^{-1}\gamma} \tag{15}$$

which maynot be immediately helpful as we still have a λ^T on rightside. We multiply both sides by $\gamma^T W^{-1}$ and then take transpose, to get rid of the unknown scalar $\lambda^T W^{-1}\gamma$ on the right, to obtain,

$$\lambda^T W^{-1}\gamma = -\frac{[2r^T\gamma + \gamma^T W^{-1}\gamma(\lambda^T W^{-1}\gamma)]}{\gamma^T W^{-1}\gamma}$$
$$\lambda^T W^{-1}\gamma = -\frac{(r^T\gamma)}{\gamma^T W^{-1}\gamma}$$

Substituting this in equation (15) we get,

$$\lambda = -\frac{\left[2Wr + \gamma(\frac{-r^T\gamma}{\gamma^T W^{-1}\gamma})\right]}{\gamma^T W^{-1}\gamma} = \left[\frac{\gamma\gamma^T r}{(\gamma^T W^{-1}\gamma)^2} - \frac{2Wr}{\gamma^T W^{-1}\gamma}\right]$$

Now finally, substituting this value of λ in equation (14) we obtain E as,

$$E = -\frac{1}{2}W^{-1}\left[\gamma\left(\frac{\gamma\gamma^T r}{(\gamma^T W^{-1}\gamma)^2} - \frac{2Wr}{\gamma^T W^{-1}\gamma}\right)^T + \left(\frac{\gamma\gamma^T r}{(\gamma^T W^{-1}\gamma)^2} - \frac{2Wr}{\gamma^T W^{-1}\gamma}\right)\gamma^T\right]W^{-1}$$

and after a few algebraic simplifications, we get

$$E = frac1\gamma^T W^{-1}\gamma\left[r\gamma^T W^{-1} + W^{-1}\gamma r^T - \frac{\gamma^T r}{\gamma^T W^{-1}\gamma}(W^{-1}\gamma\gamma^T W^{-1})\right] \tag{16}$$

this is a sum of rank-1 updates to the inverse Hessian.

Note that the weight matrix W is yet to be chosen. Infact, different quasi-Newton methods are developed with different choices of W. Also, note that the involvement of W in E is only via the combination $W^{-1}\gamma$. To preserve the property of positive-definiteness of E is to choose some W so that $W^{-1}\gamma = \alpha$. To be more concrete, if we choose $W^{-1} = (H^{k+1})^{-1} = E + (H^k)^{-1}$, we obtain, after some more algebraic simplifications, the famous BFGS update given by

$$(H^{k+1})^{-1} = (H^k)^{-1} - \frac{1}{\gamma^T\alpha}\left[(H^k)^{-1}\gamma\alpha^T + \alpha\gamma^T(H^k)^{-1} - \left(1 + \frac{\gamma^T(H^k)^{-1}\gamma}{\gamma^T\alpha}\right)\alpha\alpha^T\right]$$

And, after a few more simplifications, the corresponding update of H^k is

$$H^{k+1} = H^k + \frac{\gamma_k\gamma_k^T}{\gamma_k^T\alpha^k} - \frac{H^k\alpha^k(\alpha^k)^T H^k}{(\alpha^k)^T H^k\alpha^k}$$

which is easier to analyze (although in practice we compute and store H^{-1}).

Further, to make it low storage and easy to handle the set of $rank-1$ updates is stored and not the matrix. In other words, represent

$$(H^k)^{-1} \approx H^0 + \sum_{i=1}^{m} u^i (v^i)^T,$$

where uv^T is the $rank-1$ matrices and H^0 being a sparse matrix, the m most recent $rank-1$ updates are kept, for $10 \leq m \leq 100$ usually. This is known as the Low-storage or Limited-memory BFGS method, introduced in 1980 by Nocedal [23]. L-BFGS accelerates the convex quadratic loss function optimization, for achieving high accuracy.

In the next section we apply the above discussed optimization technique to test an important partial differential equation: the Burgers' equation. Considering the advantages of both second order as well as first order gradient based optimizers, in this study we use the combination of L-BFGS and Adam optimizer to minimize the loss function particularly for the test problem.

4. Test Problem and Result

In this section, the physics informed neural network model is built for the Burgers' equation based on the given initial and boundary conditions. There is a good accuracy and negligible error for the approximation results. Following is the brief experimental design and result of mentioned partial differential equation.

The Burgers' equation is a fundamental second order non-linear partial differential equation and is derived from the Navier-Stokes equation for the velocity field by eliminating the pressure gradient term [21, 25]. The general mathematical form in one space dimension is

$$u_t + \alpha\, u\, u_x + \beta\, u_{xx} = 0$$

It is used to simulate the propagation and reflection of shock waves. This equation arises in many applied mathematical areas such as gas dynamics, fluid mechanics, traffic flow and nonlinear acoustics. The numerical example of the non-linear Burgers' equation with Dirichlet boundary conditions, considering $\alpha = 1$ and $\beta = -0.005$ is

$$\begin{aligned} u_t + u\,u_x - (0.005)\,u_{xx} &= 0;\ x \in [-1,1],\ t \in [0,1] \\ u(0,x) &= -sin(\pi x) \\ u(t,-1) &= u(t,1) = 0 \end{aligned} \tag{17}$$

The first task is to construct a high resolution training dataset based on the partial differential equation and then we construct the neural network to approximate the solution of equation (17) to denote it as $\hat{u}(t,x)$. The network architecture that is considered here has five hidden layers containing 20, 30, 30, 30, 20 neurons respectively and the hyperbolic tangent function *tanh* is chosen to be the activation function. We define the physics informed neural network $f(t,x)$ as

$$f(t,x) = u_t + uu_x - 0.005u_{xx}$$

which is constructed using the automatic differentiation of PyTorch, one of the well documented open source libraries for Deep Learning implementation and control information of the equations. A short snippet of code can be seen in figure 2 on page 9 where we define the neural network class with 7 layers using PyTorch library. Input layer, starting with 2 inputs and expanding to 20 nodes in the first layer followed by activation function *Tanh* and other subsequent layers with finally reduce the 20 nodes of the last layer to 1 output node. The next task is training and note that the neural networks $\hat{u}(t,x)$ and $f(t,x)$ share same set of parameters which can be continuously optimized by minimising the mean square error loss given as

$$J(\Theta) = MSE_u + MSE_f$$

where

$$MSE_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2$$

and

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2$$

Here, MSE_u is loss function constructed using the initial and boundary training data denoted by $(t_u^i, x_u^i, u^i)_{i=1}^{N_u}$ and MSE_f is loss function constructed using data in the space-time domain denoted by $(t_f^i, x_f^i)_{i=1}^{N_f}$ that specify the collocation points for $f(t,x)$ introducing the physical information of partial differential equation. Though, generation of the training dataset could be done randomly, we have utilized a uniform mesh grid with training data N_u of the initial and boundary conditions being 400 points and the training data N_f in space-time domain being $20,000$ points. We learn the prediction by training all 3066 parameters (3060 weights and 6 biases, one common for each layer) of the 7 layers neural network. We start

```
# Define neural network class

class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.net = torch.nn.Sequential(
            nn.Linear(2,20),
            nn.Tanh(),
            nn.Linear(20,30),
            nn.Tanh(),
            nn.Linear(30,30),
            nn.Tanh(),
            nn.Linear(30,30),
            nn.Tanh(),
            nn.Linear(30,20),
            nn.Tanh(),
            nn.Linear(20,1)
        )

    def forward(self, x):
        out = self.net(x)
        return out
```

Figure 2. Code snippet defining neural network

with Adam optimizer to optimize the whole network and then for more accuracy apply L-BFGS full batch optimization to continue optimizing until convergence. Google's open source cloud platform Google Colab was used for python coding and particularly the PyTorch library to build and train the neural network model for which the GitHub platform was referred frequently.

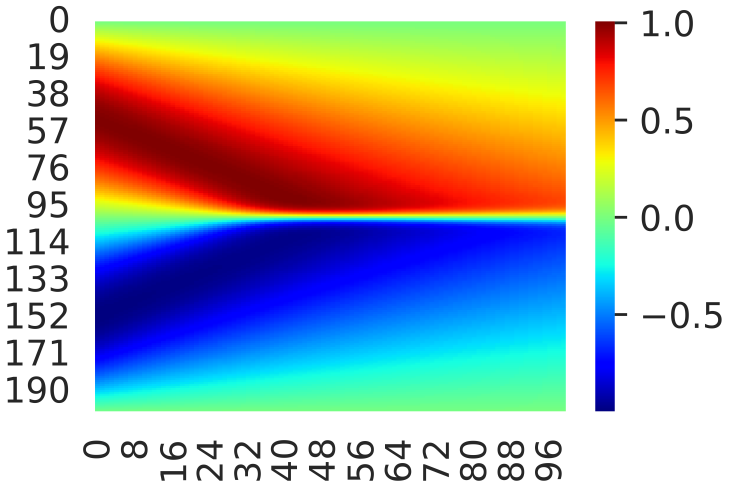
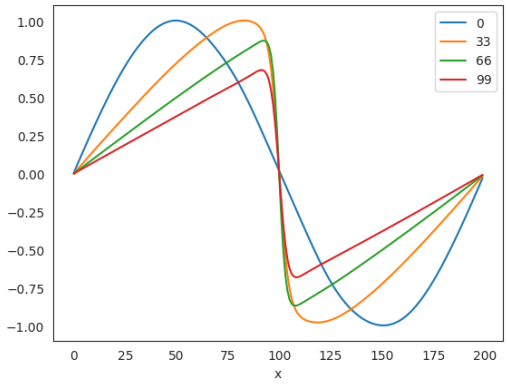
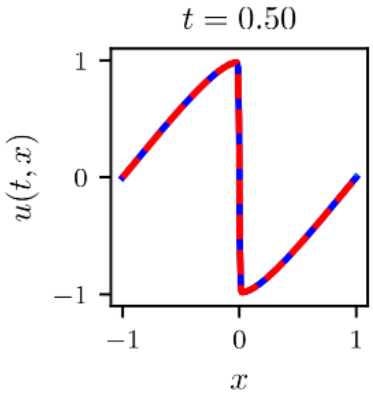


Figure 3. Heat map prediction of the Burgers' equation

Figure 3 on page 9 shows the heatmap of the spatio-temporal solution $u(t,x)$, while figure 4a on page 10 shows the predicted solutions at four different time instances, right from intial time $t = 0$ to $t = 0.99$ along with two other intermediate t values. With only a limited data points from the initial and boundary conditions, the physics informed neural network could accurately predict the intricate non-linear behaviour of the Burgers' equation that leads to the transition into a sharp internal layer around $t = 0.44$. The exact solution of Burgers' problem is analytically available in [14] and figure 4b on page 10 shows one instance where the predicted solution converges with exact solution at $t = 0.50$. The prediction error is negligible and is measured as $5.8 * 10^{-4}$ using L_2 norm which further validates the efficiency of the neural network approach.



(a) Predictions at different time steps



(b) Comparison of solution graph $u(t,x)$ of exact and predicted at $t = .50$. Blue: exact solution. Red: neural network predicted.

Figure 4. Comparison Graphs

5. Conclusion

Neural networks, an emerging class of universal function approximators are introduced in this paper which are capable of describing the physical laws governing the partial differential equation and given dataset. The Burgers' equation was tested with the physics informed neural network architecture and achieves good experimental results and accuracy. Computational optimization of the loss function plays a key role in any Deep Learning model and hence the detailed mathematical interpretation of the limited-memory BFGS was discussed and applied to the test problem (which otherwise is treated in the black box). The rapid growth in the Deep Learning technology and their wide range of applications in various scientific domains would mathematically give rise to modelling many real life phenomenas into complex non-linear equations involving partial differential equations too. Hence, designing data-driven algorithms, constructing computationally efficient physics-informed surrogate models and handling the non-convergence problem in loss function optimization becomes a crucial area of research. Although, PINNs is a powerful technique to solve the partial differential equations, there are a few challenges with strong nonlinearities and sharp gradients. The loss function can tend to become ill-conditioned which will further hinder the training process leading to vanishing gradient case. Hence, more improvised and specialised architectures are to be explored to tackle problem specific constraints. Infact, this would be the next area of focus for research. Additionally, future work will focus on comparing the performance of physics-informed neural network with other time tested classical numerical methods for solving partial differential equation.

Conflicts of Interest: The author declares no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

PDEs	Partial Differential Equations
ANN	Artificial Neural Networks
FNN	Feedforward Neural Network
PINN	Physics Informed Neural Network
MLP	Multi-layer Perceptron
FD	Finite Difference
AD	Automatic Differentiation
Adam	Adaptive Moment Estimation
L-BFGS	Limited-memory BFGS
KKT	Karush-Kuhn-Tucker

References

[1] Guo, Yanan, Xiaoqun Cao, Bainian Liu, and Mei Gao. "Solving partial differential equations using deep learning and physical constraints." Applied Sciences 10, no. 17 (2020): 5917.

[2] Cai, Shengze, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. "Physics-informed neural networks (PINNs) for fluid mechanics: A review." Acta Mechanica Sinica 37, no. 12 (2021): 1727-1738.

[3] Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis. "Machine learning of linear differential equations using Gaussian processes." Journal of Computational Physics 348 (2017): 683-693.

[4] Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis. "Numerical Gaussian processes for time-dependent and nonlinear partial differential equations." SIAM Journal on Scientific Computing 40, no. 1 (2018): A172-A198.

[5] Raissi, Maziar, and George Em Karniadakis. "Hidden physics models: Machine learning of nonlinear partial differential equations." Journal of Computational Physics 357 (2018): 125-141.

[6] Raissi, Maziar. "Deep hidden physics models: Deep learning of nonlinear partial differential equations." Journal of Machine Learning Research 19, no. 25 (2018): 1-24.

[7] Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis. "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations." arXiv preprint arXiv:1711.10561 (2017).

[8] He, Ao, Jianping Shi, Jiajun Chen, and Hui Fang. "The data-driven solutions and inverse problems of some nonlinear diffusion convection-reaction equations based on Physics-Informed Neural Network." Physica Scripta 99, no. 11 (2024): 116001.

[9] Diab, Waleed, Omar Chaabi, Shayma Alkobaisi, Abeeb Awotunde, and Mohammed Al Kobaisi. "Learning generic solutions for multiphase transport in porous media via the flux functions operator." Advances in Water Resources 183 (2024): 104609.

[10] Beguinet, Adrien, Virginie Ehrlacher, Roberta Flenghi, Maria Fuente, Olga Mula, and Agustin Somacal. "Deep learning-based schemes for singularly perturbed convection-diffusion problems." ESAIM: Proceedings and Surveys 73 (2023): 48-67.

[11] Jiang, Qinghua, Chang Shu, Lailai Zhu, Liming Yang, Yangyang Liu, and Zhilang Zhang. "Applications of finite difference-based physics-informed neural networks to steady incompressible isothermal and thermal flows." International Journal for Numerical Methods in Fluids 95, no. 10 (2023): 1565-1597.

[12] Lu, Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. "DeepXDE: A deep learning library for solving differential equations." SIAM review 63, no. 1 (2021): 208-228.

[13] Huang, Shudong, Wentao Feng, Chenwei Tang, and Jiancheng Lv. "Partial differential equations meet deep neural networks: A survey." arXiv preprint arXiv:2211.05567 (2022).

[14] Lagaris, Isaac E., Aristidis Likas, and Dimitrios I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." IEEE transactions on neural networks 9, no. 5 (1998): 987-1000.

[15] Blechschmidt, Jan, and Oliver G. Ernst. "Three ways to solve partial differential equations with neural networks—A review." GAMM-Mitteilungen 44, no. 2 (2021): e202100006.

[16] Dissanayake, MWM Gamini, and Nhan Phan-Thien. "Neural-network-based approximations for solving partial differential equations." communications in Numerical Methods in Engineering 10, no. 3 (1994): 195-201.

[17] Svozil, Daniel, Vladimir Kvasnicka, and Jiri Pospichal. "Introduction to multi-layer feed-forward neural networks." Chemometrics and intelligent laboratory systems 39, no. 1 (1997): 43-62.

[18] Li, Jing, Ji-hang Cheng, Jing-yuan Shi, and Fei Huang. "Brief introduction of back propagation (BP) neural network algorithm and its improvement." In Advances in Computer Science and Information Engineering: Volume 2, pp. 553-558. Springer Berlin Heidelberg, 2012.

[19] Raissi, Maziar, Paris Perdikaris, and George E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." Journal of Computational physics 378 (2019): 686-707.

[20] Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in pytorch." (2017).

[21] Cuomo, Salvatore, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. "Scientific machine learning through physics–informed neural networks: Where we are and what's next." Journal of Scientific Computing 92, no. 3 (2022): 88.

[22] Bengio, Yoshua. "Gradient-based optimization of hyperparameters." Neural computation 12, no. 8 (2000): 1889-1900.

[23] Nocedal, Jorge, and Stephen J. Wright, eds. Numerical optimization. New York, NY: Springer New York, 1999.

[24] Ghosh, Sourangshu. "Mathematical Foundations of Deep Learning." (2025).

[25] Das, Indrajit, Debjit Das, Papiya Debnath, Subhrapratim Nath, and Manash Chanda. "Physics Informed Neural Networks (PINNs) for Burgers' equation." In 2024 4th International Conference on Artificial Intelligence and Signal Processing (AISP), pp. 1-6. IEEE, 2024.