PINNs

Scientific Computing Case Study

Candidate Number: 1092278

1 Introduction

Differential equations are one of the key foundations of mathematical modelling within science and engineering, describing a diverse range of phenomena, including fluid flow, heat transfer, electromagnetic fields, population dynamics, and financial derivatives. Classical numerical methods, such as finite differences, finite elements, and finite volumes, are incredibly powerful and well-trusted methods, having been used successfully for decades for even safety-critical tasks [19]. Each method has its own distinct advantages and drawbacks, and although the advantages cover a wide range of problems, there are still many areas that require development or are not naturally suited to these methods. For example, these mesh-based approaches can become prohibitively expensive in high dimensions, requiring exponentially more grid points as spatial dimensions grow.

Advances in computational hardware has enabled machine learning to undergo a revolution in past decade. Before long, machine learning researchers began to turn their attention to the problem of solving differential equations, armed with powerful new techniques. Neural networks have incredible potential in this field because of their ability as function approximators capable of learning complex nonlinear mappings from data [7, 4]. Early attempts to leverage this capability amounted to applying traditional machine learning techniques, providing value-solution pairs to a standard neural network [12, 5]. While this approach has some merits, it suffers from major drawbacks, including the need for other methods to generate data. Mainly though, by failing to consider the underlying physics of the problem within the framework, solutions have no incentive to satisfy conservation laws and often fail to respect this when extrapolating. Physics-informed approaches, such as Physics-Informed Neural Networks (PINNs) [17], bridge this gap by embedding the governing equations directly into the training loss, yielding solution models that satisfy both data fidelity and physical constraints.

In this report, we review the basics of neural networks, before introducing the PINN formulation and considering a range of differential equation examples. We chose simple ordinary and partial differential equations with known solutions to ensure that we could investigate the accuracy of the PINN method. Experiments are performed to understand the effect of varying network architecture and hyperparameters, which is still relatively poorly understood (in comparison to classical methods). Finally, we consider the application of PINNs to the convection-diffusion equation, a challenge for many numerical methods.

2 Neural Network Theory

2.1 Architecture

Neural networks draw inspiration from the brain. One assembles layers of nodes connected by artificial neurons which combine and manipulate signals before passing this signal further down the network. A single artificial neuron

1. Receives inputs $y_1, y_2, ...y_n$
2. Multiplies each input by a corresponding weight $w_1, w_2, ...w_n$ and sums.
3. Adds a bias b (simulating an actiation 'threshold' that the signal must overcome). Together with the weights, this gives us a pre-activation signal
$$z = \sum_{i=1}^{n} w_i x_i + b. \tag{1}$$
4. Passes the pre-activation signal through a nonlinear activation function $\phi(z)$, to produce the output signal
$$a = \phi(z). \tag{2}$$

The weights and biases are the parameters which are tuned to achieve the desired objective of the neural network. Once we stack many of these artificial neurons into layers, we obtain a 'feedforward' neural network[1]. It is useful to describe the overall flow of information and introduce the notation we shall use

1. Input layer ($l = 0$): The input is a vector of features $\mathbf{x}$. In our case, this will be the coordinate in the domain of the differential equation.
2. Hidden layers ($l = 1, 2, ..., L - 1$): Each of these layers consists of many of the artificial neurons, propagating signals from the previous layer to the next. Every node on one layer is connected to every node on the next. The overall effect of the many artifical neuron transformations is
$$\mathbf{a}^{(l)} = \phi\Big(W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\Big), \tag{3}$$
where $W^{(l)}$ is a matrix of weights w_{ij} where i is the input neuron number and j is the output neuron number.

[1]Named as such because information propagates forwards through the network, differentiating it from other types such as cyclic networks where information is cycled through the same stages repeatedly.

3. Output layer ($l = L$): This layer outputs the final prediction, with the transformation
$$\mathbf{a}^{(L)} = \phi_{out}\left(W^{(l)}\mathbf{a}^{(l-1)} + b^{(l)}\right). \tag{4}$$
We denote the set of parameters w and b as θ. The output of the neural network we denote as $u_\theta := a^{(L)}$ (always a scalar - the final output layer has a single node because we only consider scalar-valued equations) to be the solution of the differential equation.

We use the hyperbolic tan function for our activation functions throughout this report. The overall effect of the network is a non-linear mapping $u_\theta = \Psi_\theta(\mathbf{x})$.

2.2 Training

Once we have decided upon our network architecture, we must to find a combination of weights and biases that endow our network with the ability to predict what we would like. To do this, we define $\theta := \{w, b\}$, our set of all weights and biases, and define a loss function $L(\theta)$ that measures the error between the network's predictions and the true targets for a set of training data. We then adjust θ to minimise this loss. Training the network is therefore an optimisation problem, given by
$$\textit{Find } \theta^* \textit{ such that } \theta^* = \text{argmin}_\theta L(\theta) \tag{5}$$

For regression tasks, which includes function approximation, a common loss function is the Mean Squared Error (MSE)
$$L = \frac{1}{N}\sum_{i=1}^{N} \ell := \frac{1}{N}\sum_{i=1}^{N}(u_{exact,i} - u_{\theta,i})^2, \tag{6}$$
over the training dataset of size N. ℓ is the squared error of a single data pair $\{u_{exact}, u_\theta\}$.

2.3 Loss Optimisation

We typically use first-order optimisation methods, variations on gradient descent. This is for a number of reasons, firstly each update scales linearly with the number of parameters, unlike Newton or Quasi-Newton methods [15], even quasi-Newton with low-rank updates begins to become costly for a large quantity of parameters. Secondly, the loss function graph, known as the 'loss landscape' is typically highly non-convex, rendering higher order methods unstable [8].

In gradient descent, we descend according to the steepest gradient, θ is updated by

$$\theta \leftarrow \theta - \eta \nabla L, \tag{7}$$

where η is the descent step size, and can be thought of as the learning rate. Computing the gradient of L with respect to each parameter is non-trivial, because L generally depends on these parameters indirectly through multiple layers.

To compute these gradients, we use back-propagation, an efficient algorithm based on the chain rule. Our aim is to determine the change in L due to every weight and bias in the network. L is the sum of the MSE of each individual data point ℓ. In a network of size D, with layers $\{1, ..., d\}$. We compute the effect of weights and biases in arbitrary layer k. By the chain rule, we have

$$\frac{\partial \ell}{\partial w_{ij}^{(k)}} = \frac{\partial \ell}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial w_{ij}^{(k)}}, \quad \frac{\partial \ell}{\partial b_i^{(k)}} = \frac{\partial \ell}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial b_i^{(k)}} \tag{8}$$

To compute $\frac{\partial y_i^{(k)}}{\partial w_{ij}}^{(k)}$, recall that in arbitrary layer k, neuron i computes

$$y_i^{(k)} = \phi\Bigg(\sum_j w_{ij}^{(k)} y_j^{(k-1)} + b_i^{(k)}\Bigg). \tag{9}$$

The chain rule then gives

$$\begin{aligned} \frac{\partial y_i^{(k)}}{\partial w_{ij}^{(k)}} &= \frac{d\phi(z_i^{(k)})}{dz_i^{(k)}} \frac{\partial z_i^{(k)}}{\partial w_{ij}^{(k)}} = \phi'(z_i^{(k)})\, y_j^{(k-1)} \\ \frac{\partial y_i^{(k)}}{\partial b_i^{(k)}} &= \frac{d\phi(z_i^{(k)})}{dz_i^{(k)}} \frac{\partial z_i^{(k)}}{\partial b_i^{(k)}} = \phi'(z_i^{(k)}). \end{aligned} \tag{10}$$

so if we define $\delta_i^{(k)} := \frac{\partial \ell}{\partial y_i^{(k)}} \phi'(z_i^{(k)})$, then (8) becomes

$$\frac{\partial \ell}{\partial w_{ij}^{(k)}} = \delta_i^{(k)} y_j^{(k-1)}, \quad \frac{\partial \ell}{\partial b_i^{(k)}} = \delta_i^{(k)}, \tag{11}$$

which is then applied recursively to obtain $\nabla_\theta L$ in θ [7].

2.3.1 Optimisers used in practice

Stochastic Gradient Descent (SGD) In practice, computing the exact gradient over the entire training dataset at each step is slow for large datasets. Stochastic gradient descent is a method of approximating the gradient using a subset - 'batch' - of the training data set at each iteration.

At iteration t, we select a mini-batch $\mathcal{B}_t \subset \{1, ..., N\}$, for which we compute the gradient of the loss:

$$g_t = \nabla_\theta \left(\frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \ell(\Psi_\theta(x_i), u_{exact,i}) \right) \tag{12}$$

Adam: Adam is an adaptive first-order optimizer that computes individual learning rates for each parameter by maintaining and bias-correcting running estimates of the gradients' first and second moments [11]. It is credited with fast, robust convergence across diverse deep learning tasks [7].

L-BFGS. L-BFGS is a low-rank (each BFGS update is a rank two correction) quasi-Newton method with limited memory requirements. An implicit approximation of the inverse Hessian from a window of past gradient and parameter updates to achieve superlinear convergence without ever forming the full Hessian matrix [15].

2.4 Function approximation

Neural networks owe much of their power to the Universal Approximation Theorem: even a single hidden layer network with a smooth, nonpolynomial activation can approximate any continuous function on a compact domain to arbitrary accuracy, given sufficient width [4]. Deep architectures enhance this by composing multiple nonlinear transformations which enable efficient representation of highly complex functions with exponentially fewer parameters than shallow models.

3 Physics-Informed Neural Networks (PINNs)

Turning our attention to solving differential boundary value problems, we have the differential equation posed on domain $\Omega \in \mathbb{R}^n$ with boundary $\partial\Omega$

$$\mathcal{N}[u](x) = f(x), \quad x \in \Omega, \tag{13}$$

where $\mathcal{N}$ is a (possibly nonlinear) differential operator, f is given problem data, and u is the solution we seek. Equation (13) is supplemented by the boundary condition

$$\mathcal{B}[u(x)] = g(x), \quad x \in \partial\Omega, \tag{14}$$

where $\mathcal{B}[u(x)]$ is a differential operator. For example, for a second order $\mathcal{N}$, one could split the boudnary $\partial\Omega$ into Dirichlet and Neumann parts, and $\mathcal{B}$ could consist of a fixed $u = g_D(x)$ on the Dirichlet section $\partial\Omega_D$ and $\frac{\partial u}{\partial x} = g_N(x)$ on the Neumann section $\partial\Omega_N$.

The standard supervised learning approach has one apply this neural network architecture with MSE loss to a training data set of precomputed input-solution pairs $\{(x_i, u(x_i))\}$ to achieve a continuous predicted solution across the entire domain. Unfortunately, this approach suffers from several major drawbacks. Firstly, one must have access to a dense grid of solutions $u(x_i)$. For a complex differential equation, in which the solution is not known, this means one must use traditional numerical techniques such as finite differences or finite elements. This followed by neural network training is likely to be more expensive than just simply using the traditional numerical method and interpolating, which defeats the entire point in attempting to develop a new method. Secondly, the method is simply learning to interpolate between data points. There is nothing inherent to the differential equation that is being satisfied. In regions with sparse sampling, there is no guarantee the network will provide a good prediction that respects the actual physics of the problem. For example, conservation laws may be heavily violated.

Physics-Informed Neural Networks were developed to tackle these issues and offer an alternative to traditional numerical methods.

3.1 PINN Formulation

Physics-Informed Neural Networks, first introduced by Raissi et al. [17], are a class of neural networks that integrate physical laws directly into the training process. The key idea is to construct the loss function so that it penalises violation of the differential equation. Specifically, we define the loss function to minimise the residual of the PDE, which is

$$r_\theta(x) \coloneqq \mathcal{N}[u_\theta](x) - f(x). \tag{15}$$

3.1.1 Soft Boundary Condition Method

We aim to minimise this residual across a set of collocation points. Within Ω, we select a set of N_f points $\{x_f^i\}_{i=1}^N$ and on the boundary $\partial\Omega$ we select a set of N_b points $\{x_b^j\}_{j=1}^{N_b}$. We then define the following MSE loss components:

1. Interior loss:

$$L_f \coloneqq \frac{1}{N_f} \sum_{i=1}^{N} \Big(\mathcal{N}[u_\theta](x_u^i) - f(x_u^i) \Big)^2$$

2. Boundary loss:

$$L_b := \frac{1}{N_b} \sum_{i=j}^{N} \Big(\mathcal{B}[u](x_b^j) - g(x_b^j) \Big)^2$$

Our overall loss function is then a weighted sum of these two components

$$L(\theta) = L_f(\theta) + \beta L_b(\theta), \tag{16}$$

where β is a hyper-parameter (a model parameter) which governs the relative importance given to the interior and boundary residuals. Note that the loss function is a measure of how well u_θ satisfies the entire BVP.

3.1.2 Hard Boundary Condition Method

Another way to enforce the boundary conditions is to design the neural network architecture, in which one builds the boundary term directly into the solution.

In the hard-constraint PINN approach, we construct our network ansatz in the form

$$u_\theta(\mathbf{x}) \;=\; A(\mathbf{x}) \;+\; M(\mathbf{x})\, \Psi_\theta(\mathbf{x}), \tag{17}$$

where:

- $a(\mathbf{x})$ is a *lifting function* chosen so that $\mathcal{B}\big[A\big](\mathbf{x}) = g(\mathbf{x}) \quad \forall\, \mathbf{x} \in \partial\Omega$.
- $m(\mathbf{x})$ is a *mask function* satisfying $B[m](\mathbf{x}) = 0 \quad \forall\, \mathbf{x} \in \partial\Omega$, so that the product $M(\mathbf{x})\, \Psi_\theta(\mathbf{x})$ vanishes on the boundary.
- $\Psi_\theta(\mathbf{x})$ is a standard, unconstrained neural network with parameters θ.

By construction, for any choice of θ,

$$B\big[u_\theta\big](\mathbf{x}) = B\big[A\big](\mathbf{x}) + B\big[M\,\Psi_\theta\big](\mathbf{x}) = g(\mathbf{x}) \quad \text{on } \partial\Omega, \tag{18}$$

so the boundary condition is enforced exactly, and the training objective reduces to minimizing only the PDE residual in the interior:

$$\min_\theta \frac{1}{N_f} \sum_{i=1}^{N_f} \ell^2. \tag{19}$$

3.2 Automatic differentiaton

In the construction of the PDE residual, one must obtain the derivatives of u_θ with respect to the inputs $x \in \Omega$. We do this using Automatic Differentiation (AD), a computational technique that applies the chain rule at the level of elementary operations and elementary functions in computer code, yielding derivatives that are exact up to machine precision.

Other methods that could achieve this are ruled out for several reasons. Numerical differentiation is simple to implement but can be highly inaccurate due to round-off and truncation errors [1, 10]. Symbolic differentiation suffers from expanding complex expressions, known as 'expression swell' [3]. AD avoids these issues by traversing the computation graph of $u_\theta(\mathbf{x})$ either forward or backward to accumulate derivatives efficiently [1].

To implement this in our code we use Pytorch's `autograd` engine [16]. It works by building a dynamic tape of tensor operations then applying reverse-mode AD (which is equivalent to backpropagation). Derivatives with respect to the input, coordinate x or $\mathbf{x}$ are then obtained. A simplified pseudocode outline is:

```
# Forward pass: compute network output and PDE residual
x = collocation_points.requires_grad_(True)
u = model(x)
# compute first and second derivatives
du_dx = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u),
                            create_graph=True)[0]
d2u_dx2 = torch.autograd.grad(du_dx, x,
                              grad_outputs=torch.ones_like(du_dx))[0]
# Residual and loss
residual = differential_operator(u, du_dx, d2u_dx2) - f(x)
loss = (residual**2).mean() + boundary_loss
```

3.3 Implementation

We implement our PINN in PyTorch, defining a `nn.Module` that represents Ψ_θ with fully connected layers and tanh activations. During each training epoch we:

1. Sample interior collocation points and boundary points.
2. Compute u_θ, its derivatives via `autograd`, and form the PDE and boundary residuals.

3. Backpropagate the combined loss and update parameters using the chosen optimizer.

Figure 1 illustrates the network architecture for our simple boundary-value problem.

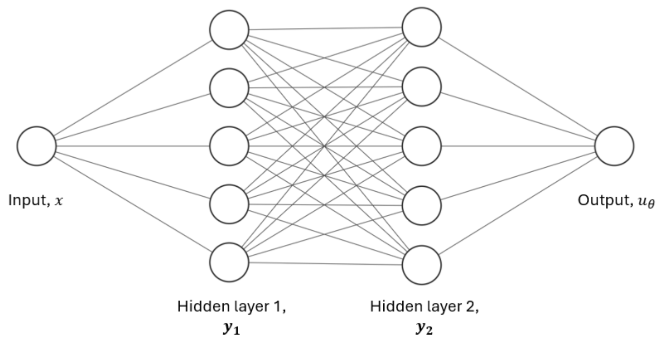
Input, x
Hidden layer 1, $\mathbf{y_1}$
Hidden layer 2, $\mathbf{y_2}$
Output, u_θ

Figure 1: Network architecture for the boundary value problem.

4 Differential Equation Applications

4.1 Ordinary Differential Equation

We first consider the following simple boundary value problem (BVP)

$$\begin{cases} y'' = 2, & x \in [0,1], \\ y(0) = 0, \quad y(1) = 0 \end{cases} \tag{20}$$

We solve this problem using a PINN with two hidden layers consisting of 50 nodes and the soft constrained method, with $\beta = 1$. Unless otherwise specified, we used five equally spaced internal collocation points and soft constraints. We trained the model for a total of 5000 epochs, where each epoch is a single pass through the entire dataset [13, 7].

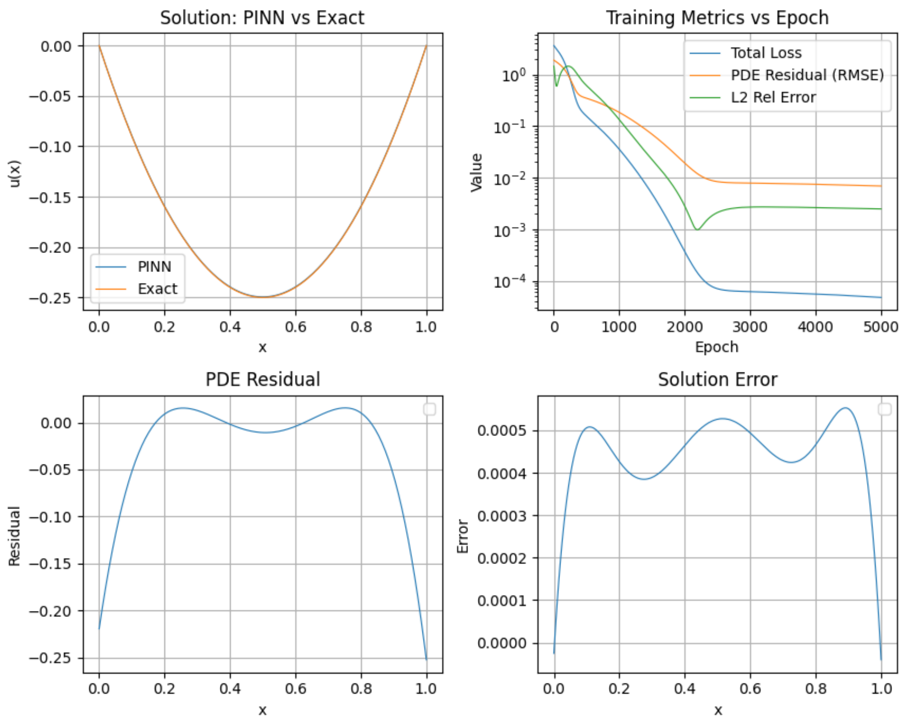

Figure 2: Top left: Neural network solution u_θ at epoch 5000. Top right: Training metrics: loss, residual, and L^2 relative error $\frac{\|u-u_\theta\|}{\|u\|}$. Bottom left: PDE Residual r_θ across the domain. Bottom right: Solution error $u - u_\theta$ across the domain.

Figure 2 shows the solution, residual, and error at epoch 5000, along with a training history of training metrics. The first thing we notice is that the result is very accurate, with a maximum solution error $\mathcal{O}(10^{-4})$. The PINN solution and exact solution are indistinguishable on the first plot. Secondly, we see that all three training metrics follow similar trends, although interestingly the L^2 error increases slightly around epoch 2000, despite continuing reductions in the overall loss and residual. This emphasises the difference between minimisation of the PDE residual and accuracy of the solution. The optimiser seeks to drive down the loss (based on the residual), which in general indirectly reduces the L^2 error, but it is certainly possible for the optimiser to find a parameter update which produces a u_θ that better satisfies the PDE on average and is less accurate in the L^2 norm. Investigating the residual and error across the domain at epoch 5000, we see that the residual is greatest at the boundaries whereas the error is greatest in the interior of the domain. This is an artefact of the soft constraint method, in which the interior loss term L_f uses the residual and the boundary loss term L_b uses the error. The different profiles are therefore a manifestation of the minimisation of the respective terms.

Next, we implement the hard constrained method using the ansatz

$$u(x) = x(x-1)\Psi_\theta(x), \tag{21}$$

with a lifting function $a(x) \equiv 0$ and a mask function $m(x) = x(x-1)$.

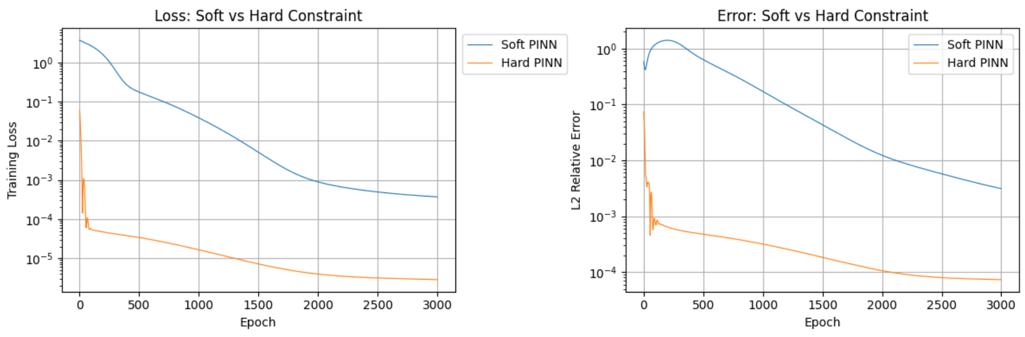

Figure 3: Performance metrics comparison for the hard and soft constraint methods for problem (20).

Figure 3 compares the soft and hard constraint loss and error. The hard constraint method achieves orders of magnitude better convergence with equivalent training, as we expect. By encoding the boundary conditions into the ansatz, we have dramatically reduced the degrees of freedom in the model. The remaining residual and error arise solely from approximation within the domain interior, ($L_b \equiv 0$). The drawback of the hard constraint method is that such an ansatz must be formulate in advance, which could be very difficult if not impossible for complex problems and domains.

4.2 Optimiser Selection

Noting that the training metrics in early training runs were highly oscillatory with increasing the epoch, we decided to investigate the impact of different optimisation methods. Principally, we compare the two most common optimisers used within deep learning, Adam with SGD. We also investigate variations of the Adam optimiser, by lowering the learning rate from 10^{-3} to 10^{-4} - 'AdamSLOW', introducing a decaying learning rate - 'AdamW', and a hybrid method of standard Adam with learning rate 10^{-3} followed by the second-order L-BFGS quasi-newton method - 'ADAM + LBFGS'.

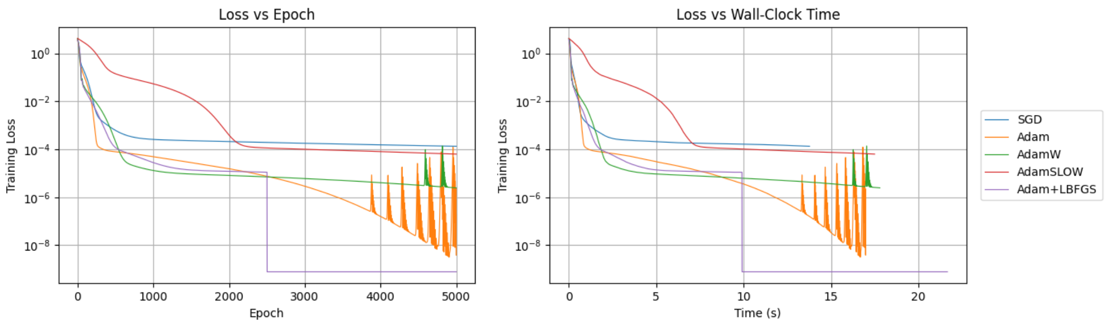

Figure 4: Left: Training loss against epoch. Right: Training time against epoch. Different optimisers.

Figure 4 compares these optimisers with the loss against epoch and wall clock time, which we included to enable comparison of efficiency. The similarity of the two plots demonstrates that the computational time per epoch for each of the methods are very close. We note that any comparisons in this section apply to this ODE and this selection of hyperparameters, it is difficult to say to what extent the conlusions can be extrapolated. Comparing SGD and Adam directly, we see that SGD convergence becomes extremely slow around epoch 1000. Adam does not suffer from this, the loss continues to decrease but becomes oscillatory. The trouble with the standard Adam is that these oscillations can bring the loss above the SGD loss. AdamW and AdamSLOW are simple learning rate variations, but they illustrate that often convergence issues can be solved by tuning this parameter.

Adam + LBFGS is the most promising. The LBFGS step kicks in at epoch 2500, but this could equally be set to kick in as soon as the loss reaches a set value, say 10^{-3}. The sharp drop we see is because the LBFGS is a batch optimiser which will run up to 500^2 inner quasi-Newton iterations in one epoch. On this problem, this is enough to converge to the gradient tolerance 10^{-9} within the epoch 2500.

In subsequent runs we use the AdamSLOW optimiser.

4.3 Sensitivity to Network Hyperparameters

We now investigate the impact of the network architecture on the sensitivity of the solution. We vary the width, depth, boundary loss hyperparameter β, and the number of grid points.

[2]Set by the user.

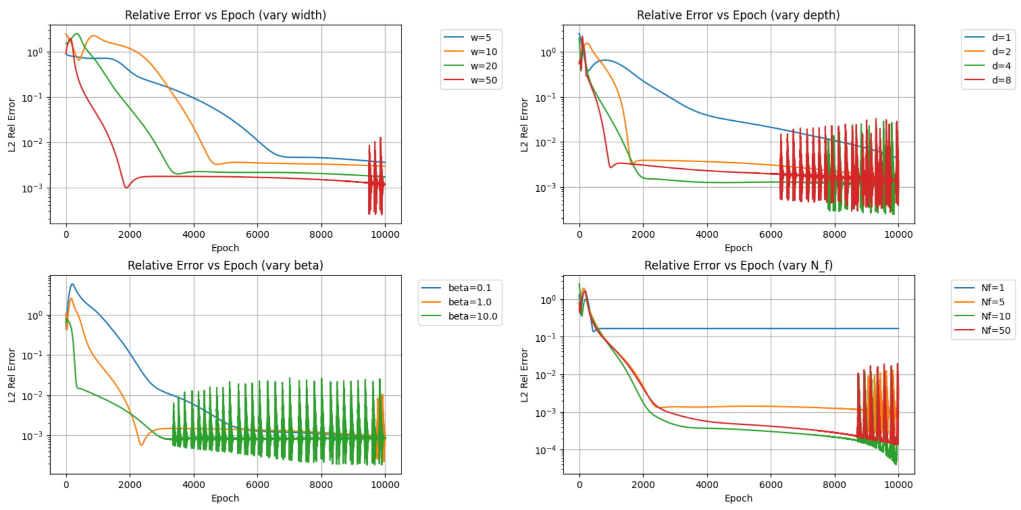

Figure 5: Relative L^2 error of u_θ against epoch for four hyperparameter variations. Top left: Varied hidden layer width. Top right: Varied number of hidden layers. Bottom left: Varied boundary loss weighting hyperparameter β. Bottom right: Varied interior collocation point quantity, equally spaced points.

Further investigating the effect of varying β, we have plotted the neural network solution u_θ on the domain at increasing epochs in Figure 6.

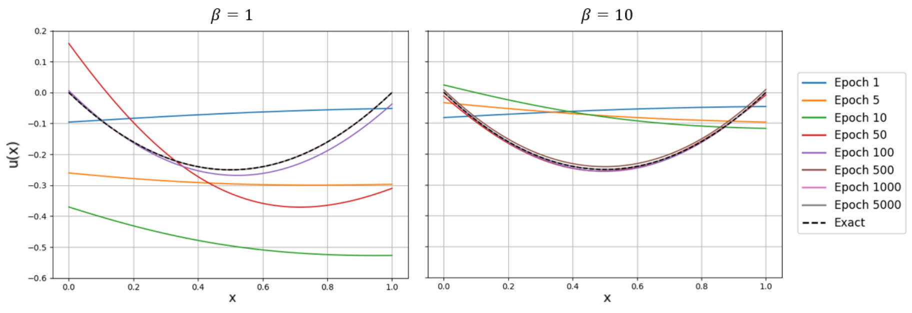

Figure 6: Neural network solution u_θ for different β values.

One can see clearly that when $\beta = 1$ the solution does not prioritise satisfaction of the boundary conditions, instead moving across the domain and correlating loosely with the shape of the exact solution before it begins to converge. When $\beta = 10$ on the hand, we see that the boundary values remain close to the boundary conditions before the solution begins to converge. Unfortunately, the drawback of using such a high β is that the solution struggles to converge in the interior of the domain. This is because the parameter β has fundamentally changed our loss landscape, increasing its anisotropy. If we define the Hessians with respect to θ, at θ^*, $H_f := \nabla^2 L_{\mathrm{f}}(\theta^\star)$ and

$H_b \coloneqq \nabla^2 L_\mathrm{b}(\theta^\star)$, then the total Hessian of the weighted loss is

$$H(\theta^\star) = \nabla^2 \mathcal{L}(\theta^\star) = H_f + \beta\, H_b. \tag{22}$$

By Weyl's inequalities for sums of symmetric matrices, the extreme eigenvalues obey [9]

$$\lambda_{\max}(H_f + \beta H_b) \;\le\; \lambda_{\max}(H_f) \;+\; \beta\, \lambda_{\max}(H_b), \tag{23}$$

$$\lambda_{\min}(H_f + \beta H_b) \;\ge\; \lambda_{\min}(H_f) \;+\; \beta\, \lambda_{\min}(H_b). \tag{24}$$

For simplicity, we assume that $\lambda_{\min} = 0$, which often occurs often with PINNs (or if not, $\lambda_{\max} \ll 1$. We obtain

$$\lambda_{\min}(H) \;\ge\; \lambda_{\min}(H_f). \tag{25}$$

Thus the condition number $\kappa(H)$ satisfies

$$\kappa(H) = \frac{\lambda_{\max}(H)}{\lambda_{\min}(H)} \;\le\; \frac{\lambda_{\max}(H_f) + \beta\, \lambda_{\max}(H_b)}{\lambda_{\min}(H_f)} \;\sim\; \mathcal{O}(\beta) \quad (\beta \to \infty). \tag{26}$$

Therefore, increasing β results increases the ill-conditioning of the Hessian, and the loss valley becomes highly anisotropic, forcing any first-order optimizer to adopt a very small global step-size. As shown in Figure 2, the constraints are satisfied quickly but the solution then converges slowly or at all along the relatively flat interior directions.

4.4 Partial Differential Equation Example

To extend the previous section on the 1D Poisson equation, we consider the Poisson boundary-value problem on the unit square

$$\begin{cases} -\nabla^2 u(x,y) = f(x,y), & (x,y) \in (0,1) \times (0,1), \\ u(x,y) = g(x,y), & (x,y) \in \partial\big([0,1]^2\big). \end{cases} \tag{27}$$

with source term and boundary condition

$$f(x,y) = -2\pi^2 \sin(\pi x)\sin(\pi y), \quad g(x,y) = \sin(\pi x)\sin(\pi y). \tag{28}$$

The exact solution is $u(x,y) = \sin(\pi x)\sin(\pi y)$.

For PINN architecture, we used five hidden layers of width 50, and implement using soft constraints. The input is now a vector, x_1, x_2. The PDE loss is

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_f} \sum \big(u_{xx} + u_{yy} - f(x,y)\big)^2 \tag{29}$$

on $N_f = 3000$ random interior collocation points, and the boundary loss is

$$\mathcal{L}_{\text{BC}} = \frac{1}{N_b} \sum \big(u_\theta - g\big)^2 \tag{30}$$

on $N_b = 200$ random boundary points.

The solution, following 5000 epochs, is given by Figure 7.

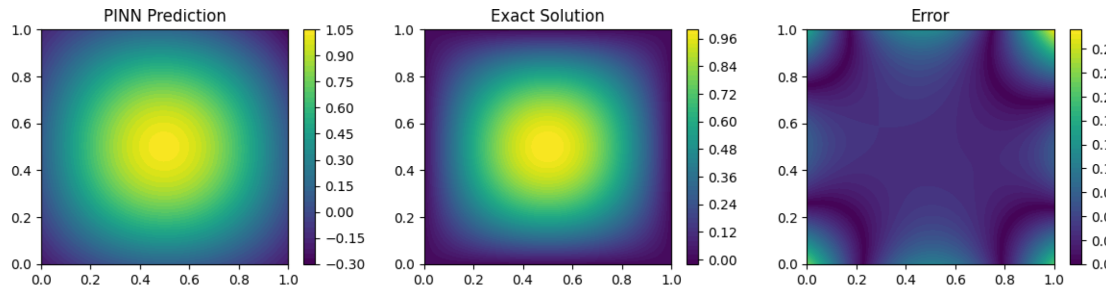

Figure 7

We see that the solution profile is very similar, but the error is quite high, given it is $\mathcal{O}(0.1)$, the same order as the solution. This is dominated by the error at the boundary of the domain, particularly in the corners. A likely explanation for this is the combination of the random sampling along the boundary along with the corners, which have less impact on the MSE, so the optimiser does not need to reduce the loss here at the same rate as the interior.

5 Extension: Convection-dominated problems

5.1 Background

So far we only considered diffusion-dominated problems. Convection-dominated problems have historically been a challenge even for traditional numerical methods. The convection-diffusion is particularly problematic, because thin boundary layers can form which have very large gradients. We consider the problem

$$-\epsilon u'' + u' = 1, \quad x \in (0,1), \tag{31}$$

with boundary conditions $u(0) = u(1) = 0$. $\epsilon \ll 1$ is the diffusion coefficient.

To see how boundary layers form we first investigate (31) analytically. This type of equation is referred to as 'singularly perturbed' because the perturbation $\epsilon y''$ fundamentally changes the structure of the equation from the case where $\epsilon = 0$. If $\epsilon = 0$, (31) becomes $y' = 0$, which has the general solution $y = Ax$ and will only satisfy both boundary conditions in special cases. Now, when we perturb this equation by letting $\epsilon > 0$, a solution now exists for any reasonable forcing function f. If ϵ also happens to be small, the solution will remain similar to the pure convection solution $y = Ax$ over most of the domain, and the diffusive term will only take over close to $x = 1$. Here, the function value must change rapidly to meet the $y(1) = 0$ boundary condition, leading to very large gradients, known as the 'boundary layer'.

Another way to see this is through the analytical solution

$$u = x + \frac{e^{-(1-x)/\epsilon} - e^{-1/\epsilon}}{e^{-1/\epsilon} - 1}. \tag{32}$$

The gradient $y' \to \infty$ at $x = 1$ as $\epsilon \to 0$.

Traditional numerical methods, such as the first-order finite difference method and the finite element method with linear elements, struggle to approximate the solution accurately when convection dominates, leading to oscillatory solutions. This is because the approximations introduce negative diffusion, which works to amplify infinitesimal differences in the approximation [2]. PINNs offer a new approach entirely.

With this particular equation, the central-difference finite difference scheme

$$-\epsilon \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \frac{u_{i+1} - u_{i-1}}{2h} = 0, \tag{33}$$

with $N = 200$ begins to break down solutions begin to break down when $\epsilon \lesssim 0.005$[3]. Figure 8 and Figure 9 compare the finite difference and PINN solution with 200

[3]In convection-diffusion theory, one defines the Péclet number $\mathcal{P}_h = \frac{h}{2\epsilon}$, where h is the largest element length. Finite difference schemes begin to oscillate at $\mathcal{P}_h \approx 2$.

[2]

nodes/collocation points. The PINN architecture consisted of two hidden layers of 50 neurons, and it was run for 8000 epochs.

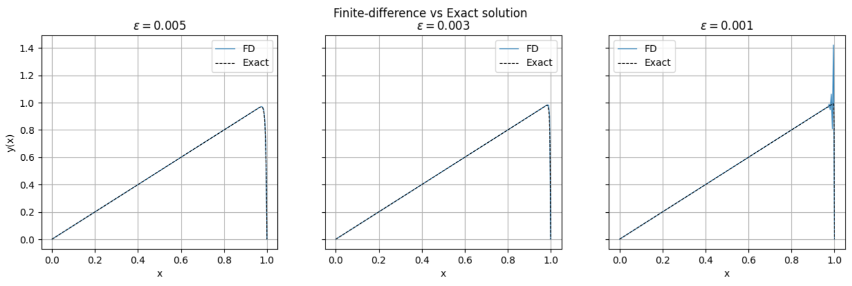

Figure 8: Finite difference solution to (31).

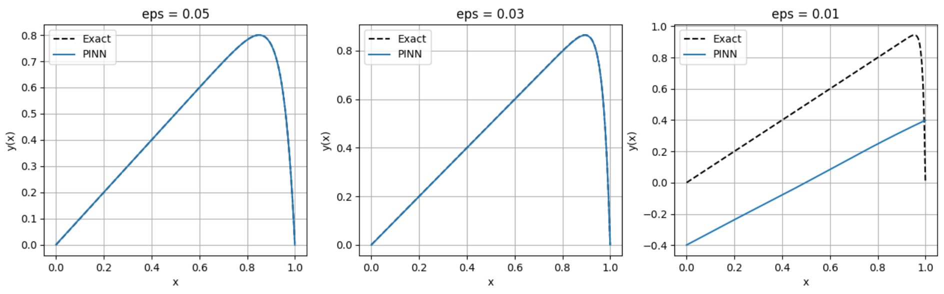

Figure 9: PINN solution to (31).

Finite Difference		PINN	
ϵ	Time (ms)	ϵ	Time (ms)
0.005	0.97	0.05	61758.0
0.003	1.80	0.03	56270.6
0.001	0.82	0.01	59107.3

Table 1: Comparison of compute time between Finite Difference and PINN method for solving (31).

We see that the PINN solution can handle problems of up to one order of magnitude greater ϵ with the current network configuration and training setup. Table 1 shows that the compute time is also significantly slower. Although, we note that the time could be taken to a set loss value, say 10^{-3}, since one may continue convergence indefinitely. Figure 10 shows the points epochs at which convergence begins to take place.

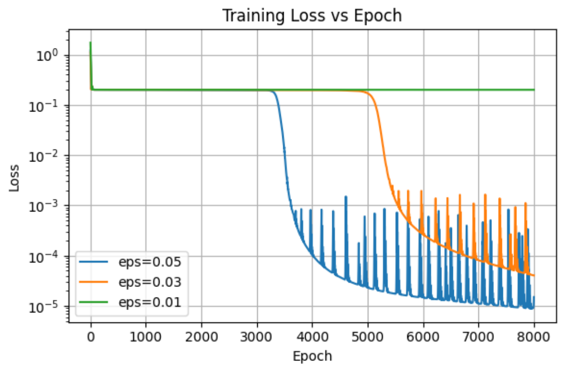

Figure 10: Training loss against epoch for varying $eps = \epsilon$.

Standard PINNs struggle with the convection diffusion equation, and more generally with any equation that exhibit high-frequency features, because of the 'vanishing gradient problem'. This phenomenon arises when the loss landscape is ill-conditioned. Wang et al show that when high-frequency components (such as our steep boundary layer) are present in the solution, $\|\nabla L_b\|$ may be much smaller than $\|\nabla L_f\|$, which biases the neural network training to neglect the contribution of the boundary term [18]. In other words, the loss landscape is so ill-conditioned that the optimiser struggles to descend along these directions.

Figure 9 and Figure 10 offer exceptional insight into these issues. We see in Figure 9 that at $\epsilon = 0.01$ the gradient of u_θ, u'_θ is very close to the gradient of the convection-dominated section of the solution, so the solution closely satisfies the residual $-\epsilon u''_\theta + u'_\theta \approx 1$ in the domain interior. The entire solution is shifted to minimise the boundary residual as if the solution was constrained to this 'linear' shape, with $u_\theta(0) \approx -u_\theta(1)$. What we are seeing is an optimiser which is struggling to locate the global minimum, which would require a combination of parameters that dramatically increase y'' close to $x = 1$. As $\epsilon \to 0$, the contribution to ∇L_f of the second derivative, $\frac{\partial u''}{\partial \theta} \to 0$.

5.2 Remedy: Shishkin grids

A typical remedy in finite element methods is the use of Shishkin grids [6]. A Shiskin grid is a grid composed of two uniform, sub-grids of different spacings. Let I^N denote an arbirary mesh (set of collocation points) with mesh points x_i, $i = 0, 1, ..., N$, such that $0 = x_0 < x_1 < ... < x_N = 1$ [14]. For our problem (31), the domain is split at

$$\sigma = \min\left\{\tfrac{1}{2}, \alpha\, \epsilon \ln N\right\}, \quad J = \lfloor \sigma N \rfloor, \quad x_J = \sigma. \tag{34}$$

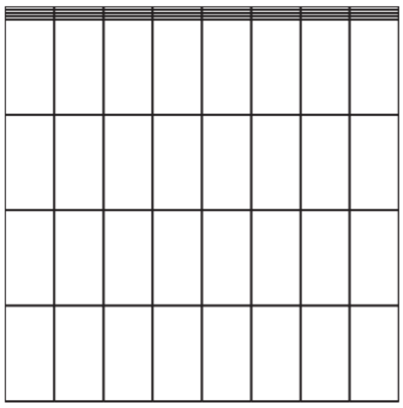
Figure 11: Shishkin grid for 2D geometry with a boundary layer on the top side, taken from [6].

α is a parameter which can be used to scale the width of the inflation layer to ensure it correlates with the width of the boundary layer of the solution. Then we let each sub-grid have uniform spacing, so that

$$0 = x_0 < x_1 < \cdots < x_J = \sigma \quad \text{and} \quad \sigma = x_J < x_{J+1} < \cdots < x_N = 1,$$

with the first J intervals covering $[0, \sigma]$ coarsely and the remaining $N - J$ intervals covering $[\sigma, 1]$ finely. Figure 11 illustrates this Shishkin grid.

The fine layer is intended to cover the boundary layer region. Once this is achieved, for traditional numerical methods the fine grid enables better approximation of the sharp changes in gradients which occur in this layer. With regards to PINNs, our motivation for using this method is to concentrate minimisation of our PDE residual in this region.

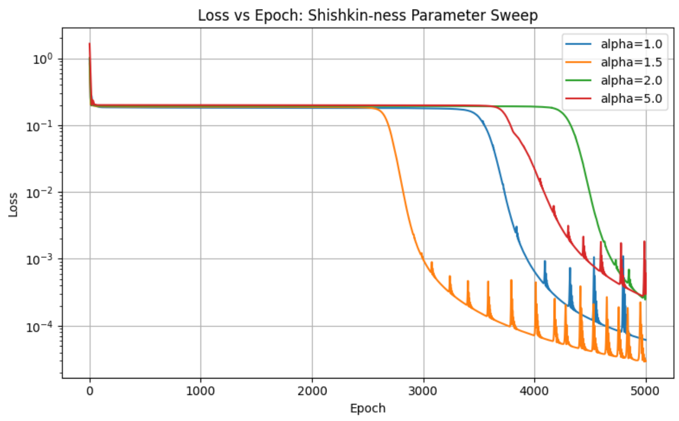

Figure 12: Loss vs epoch for varying Shishkin grid parameter α.

Figure 12 shows the point where convergence begins to occur is earlier than the standard method in Figure 10. Varying the parameter α - the 'Shishkiness' - we see that the convergence begins earlier at $\alpha = 1.5$, but decreases in effectiveness beyond that. This is likely because we are capturing more of the boundary layer within our fine sub-grid when we increase α to 1.5, but beyond that the spacing becomes greater, diminishing the effect of the density of the collocation points.

6 Conclusion

To conclude, we have formulated PINNs for a number of example problems and investigated the effect of changing network hyperparameters on the convergence rate of the solution. For the 1D ODE, hard constraints appeared to win over soft constraints by orders of magnitude, but this relies on simple geometries and differential equations to have a reliable ansatz. With the 2D Poisson equation, we observed that random collocation may lead to larger errors near complex boundary regions, suggesting that sampling strategies are important considerations when developing a PINN model.

When applied to convection–diffusion problems with steep boundary layers, standard PINNs struggled due to ill-conditioning of the loss landscape. Computational time was significantly greater than classical methods, and the diffusion coefficient had to be an order of magnitude higher to correct this. Shishkin grids did offer some minor convergence benefits but did little to bring together the gap between classical and PINN methods for this problem.

Future work could explore adaptive loss weighting [**wang2021understanding**] to remedy the convergence issues we saw throughout this report. Adaptive collocation, taking inspiration from adaptive meshing from finite elements, could also be a worthwhile avenue to pursue. Overall, PINNs offer a lot of potential to complement to classical solvers but much development is required to bridge the gap.

7 References

References

[1] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey.* 2018. arXiv: 1502.05767 [cs.SC]. URL: https://arxiv.org/abs/1502.05767.

[2] Alexander N. Brooks and Thomas J.R. Hughes. “Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations”. In: *Computer Methods in Applied Mechanics and Engineering* 32.1 (1982), pp. 199–259. ISSN: 0045-7825. DOI: https://doi.org/10.1016/0045-7825(82)90071-8. URL: https://www.sciencedirect.com/science/article/pii/0045782582900718.

[3] George F. Corliss. “Applications of Differentiation Arithmetic11This work was supported in part by IBM Deutschland GmbH.” In: *Reliability in Computing.* Ed. by Ramon E. Moore. Academic Press, 1988, pp. 127–148. ISBN: 978-0-12-505630-4. DOI: https://doi.org/10.1016/B978-0-12-505630-4.50013-4. URL: https://www.sciencedirect.com/science/article/pii/B9780125056304500134.

[4] George Cybenko. “Approximation by Superpositions of a Sigmoidal Function”. In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314.

[5] D. A. R. Dissanayake and Nguyen Phan-Thien. “Neural-network-based approaches for the numerical solution of partial differential equations”. In: *Computer Methods in Applied Mechanics and Engineering* 125.1-4 (1995), pp. 143–155.

[6] Howard Elman, David Silvester, and Andy Wathen. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics.* Oxford University Press, June 2014. ISBN: 9780199678792. DOI: 10.1093/acprof:oso/9780199678792.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780199678792.001.0001.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* Cambridge, MA: MIT Press, 2016.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* Cambridge, MA: MIT Press, 2016.

[9] Roger A. Horn and Charles R. Johnson. *Matrix Analysis.* 2nd. Cambridge: Cambridge University Press, 2012.

[10] Max E. Jerrell. “Automatic Differentiation and Interval Arithmetic for Estimation of Disequilibrium Models”. In: *Computational Economics* 10.3 (Aug. 1997), pp. 295–316. ISSN: 1572-9974. DOI: `10.1023/A:1008633613243`. URL: `https://doi.org/10.1023/A:1008633613243`.

[11] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: (2015).

[12] Daniel B. Meade and Claudio P. Fernandez. “Solving Ordinary Differential Equations Using Neural Networks”. In: *Mathematical Biosciences* 105 (1991), pp. 37–57.

[13] Tom M. Mitchell. *Machine Learning.* New York, NY: McGraw-Hill, 1997.

[14] Thai Anh Nhan and Relja Vulanovic. “A note on a generalized Shishkin-type mesh”. In: *Novi Sad Journal of Mathematics* Accepted (May 2018). DOI: `10.30755/NSJOM.07880`.

[15] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* 2nd. New York, NY: Springer, 2006.

[16] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems.* Vol. 32. 2019, pp. 8024–8035.

[17] M. Raissi, P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2018.10.045`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999118307125`.

[18] Sifan Wang, Yujun Teng, and Paris Perdikaris. “Understanding and Mitigating Gradient Flow Pathologies in Physics-Informed Neural Networks”. In: *SIAM Journal on Scientific Computing* 43.5 (2021), A3055–A3081. DOI: `10.1137/20M1318043`. eprint: `https://doi.org/10.1137/20M1318043`. URL: `https://doi.org/10.1137/20M1318043`.

[19] Olek C. Zienkiewicz, Robert L. Taylor, and Jian Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals.* 7th. Oxford: Butterworth-Heinemann, 2013.