Numerical Solution of Differential Equations Using Neural Networks.

MATHEMATICAL INSTITUTE, UNIVERSITY OF OXFORD

Case Study in Scientific Computing

Candidate number: 1077723

Date: September 11, 2024

1 Introduction

Differential equations (DEs) are the basis of scientific modelling. Even though most DEs lack an analytical solution, they can be approximated using classical numerical methods — e.g. finite differences or finite elements — based on the discretisation of the domain. However, mesh generation for these techniques gets increasingly expensive as the domain gets more complicated or the target function less smooth, and the accumulated error can become significant.

In this project, we explore an alternative to conventional techniques based on neural networks (NNs) [1, 2]. This method embeds the DE and its boundary conditions (BCs) into the loss function of the net. It does not require space to be discretised (i.e. it is mesh-free), being a potential solution to the curse of dimensionality [2].

With this purpose in mind, in section 2, we give a brief general introduction to NNs, how they can be built to solve DEs, and how to use automatic differentiation (AD) to differentiate the NN's outputs and loss. In section 3, we discuss how NNs can solve ordinary differential equations (ODEs). Using as examples a 1D Poisson equation and a 1D Helmholtz equation, we explore how the method for setting the BCs, the width and depth of the net, and the number of grid points affects the training of the NN. We further show how the techniques used to solve these linear ODEs can be extended to solve systems of ODEs, and non-linear ODEs. In section 4, we demonstrate the potential of NNs by solving a 2D (elliptic) Poisson equation, a 1D and 2D (parabolic) Burger's equation and a 1D (complex-valued parabolic) Schrodinger's equation. We conclude the project in section 5. Throughout this report, we will illustrate how NNs can be implemented by including Python code snippets using `PyTorch` [3].

2 Theory

2.1. Neural networks (NNs)

Consider feature variables $\boldsymbol{x} \in \mathbb{R}^m$ which map to a target variable $\boldsymbol{y} \in \mathbb{R}^k$. NNs aim to learn, from a set of training points $T = \{(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)})\}_{j=1}^{J}$ specifying the desired output $\boldsymbol{y}^{(j)}$ for each $\boldsymbol{x}^{(j)}$, what would the $\boldsymbol{y}$ of an unseen $\boldsymbol{x} \not\in \{\boldsymbol{x}^{(j)}\}_{j=1}^{J}$ be. There are many types of NNs like the convolutional or recurrent NNs. We will consider its simplest form, the feed-forward neural network (FNN), which is sufficient for most DEs [1, 2].

To make predictions, FNNs take an input $\boldsymbol{x} = \boldsymbol{h}^{(1)}$ and feed it into a net of nodes (e.g. fig. 1) where the information flows forwards (i.e. without cycles) to get an output $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(N)}$. The architecture of a FNN is usually specified by its width (maximum number of nodes per layer) and depth (number of layers). Each node applies an affine transformation followed by a nonlinear action, so at layer $l+1$ the node values are

$$\boldsymbol{h}^{(l+1)} = \Phi_l(W^{(l)}\boldsymbol{h}^{(l)} + \boldsymbol{b}^{(l)}) \qquad \text{for } l = 1, 2, ..., N-1, \tag{1}$$

where $W^{(l)} \in \mathbb{R}^{n_{l+1} \times n_l}$ and its entries $w_{i,j}^{(l)}$ are the weights, $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_{l+1}}$ is the bias, and $\Phi_l(\cdot)$ is a nonlinear activation applied in element-wise fashion to the vector arguments [4]. Hence, mathematically, FNN can be thought of as a compositional function.

To learn the parameters $\theta := \{W^{(l)}, \boldsymbol{b}^{(l)}\}_{l=1}^{N-1}$ that will minimise the error between $\hat{\boldsymbol{y}}(\boldsymbol{x}|\theta)$ and $\boldsymbol{y}(\boldsymbol{x})$, NNs minimise a loss function $L(\theta|T)$, which is the average over all training data T of a point-wise loss $l(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}|\theta)$. That is,

$$L(\theta|T) = \frac{1}{J}\sum_{j=1}^{J} l(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}|\theta). \tag{2}$$

On `PyTorch` a FNN with the same Φ_l on all layers except the last is

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, layers, activation=torch.nn.Sigmoid):
        super().__init__()
        self.depth = len(layers) - 1 # set depth of the network
        self.activation = activation # set activation function
        layer_lst = []
        for i in range(self.depth - 1): # build architecture
            layer_lst.append(torch.nn.Linear(layers[i], layers[i + 1]))
            layer_lst.append(self.activation())
        layer_lst.append(torch.nn.Linear(layers[-2], layers[-1]))
        self.net = torch. nn.Sequential(*layer_lst) # set architecture
    def forward(self, x): # define forward pass of the neural network
        return self.net(x)
```

Then, $\hat{\boldsymbol{y}}(\boldsymbol{x}|\theta)$ is obtained by

```
nn = NeuralNetwork(layers)
y = nn(x)
```

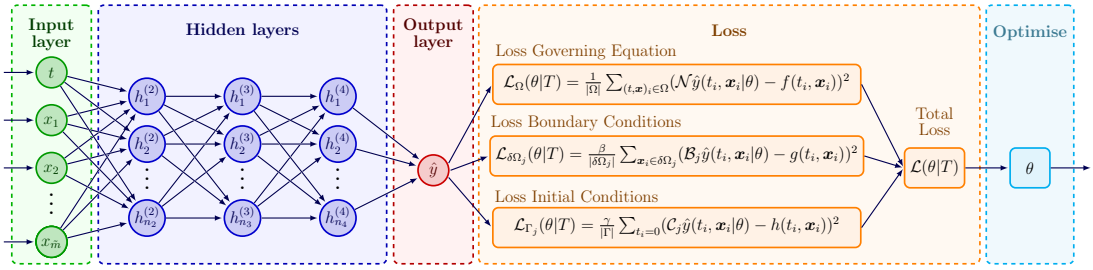

Fig. 1: Schematic of an FNN with three hidden layers approximating a DE. The input variables $(t, \boldsymbol{x}) = (t, x_1, x_2, ..., x_{\tilde{m}}) \in \mathbb{R}^{m+1}$ map into the output variable $\hat{y} \in \mathbb{R}$.

2.2. Neural networks for differential equations (DEs)

Consider solving the following DE for the function $y(t, \boldsymbol{x})$:

$$\mathcal{N}y(t, \boldsymbol{x}) = f(t, \boldsymbol{x}) \qquad \text{in } \Omega \subset \mathbb{R}^m, \tag{3a}$$

$$\mathcal{B}_j y(t, \boldsymbol{x}) = g_j(t, \boldsymbol{x}) \qquad \text{on } \delta\Omega_j \subset \delta\Omega \subset \mathbb{R}^{m-1}, \tag{3b}$$

$$\mathcal{C}_j y(0, \boldsymbol{x}) = h_j(\boldsymbol{x}) \qquad \text{in } \Omega \subset \mathbb{R}^m, \tag{3c}$$

where $\mathcal{N}$, $\mathcal{B}_j$, and $\mathcal{C}_j$ are differential operators defining the DE, the jth boundary condition (BCs), and the jth initial condition (ICs), respectively. The FNN (fig. 1) will approximate $y(t, \boldsymbol{x})$ as $\hat{y}(t, \boldsymbol{x}|\theta)$. To convert this constrained optimisation problem into an unconstrained problem we can use a loss function (or penalty function)

$$\mathcal{L}(\theta|T) = \mathcal{L}_\Omega(\theta|T) + \sum_j \mathcal{L}_{\delta\Omega_j}(\theta|T) + \sum_j \mathcal{L}_{\Gamma_j}(\theta|T), \tag{4a}$$

where $\mathcal{L}_\Omega$ is the loss at the points inside the domain

$$\mathcal{L}_\Omega(\theta|T) = \frac{1}{|\Omega|} \sum_{(t,\boldsymbol{x})_i \in \Omega} (\mathcal{N}\hat{y}(t_i, \boldsymbol{x}_i|\theta) - f(t_i, \boldsymbol{x}_i))^2, \tag{4b}$$

$\mathcal{L}_{\delta\Omega_j}$ is the loss at the jth boundary

$$\mathcal{L}_{\delta\Omega_j}(\theta|T) = \frac{\beta}{|\delta\Omega_j|} \sum_{\boldsymbol{x}_i \in \delta\Omega_j} \left(\mathcal{B}_j\hat{y}(t_i, \boldsymbol{x}_i|\theta) - g(t_i, \boldsymbol{x}_i)\right)^2, \tag{4c}$$

and $\mathcal{L}_{\Gamma_j}$ is the loss of the jth initial condition

$$\mathcal{L}_{\Gamma_j}(\theta|T) = \frac{\gamma}{|\Gamma|} \sum_{t_i=0} \left(\mathcal{C}_j\hat{y}(t_i, \boldsymbol{x}_i|\theta) - h(t_i, \boldsymbol{x}_i)\right)^2. \tag{4d}$$

Here, $|\Omega|$, $|\delta\Omega_j|$, and $|\Gamma|$ are the number of points inside the domain, at the jth boundary, and at the initial time, respectively. Moreover, β and γ are the weights given to the boundary and initial constraints, and they must be carefully chosen to balance the accuracy of the BCs, ICs, and DE.

On `PyTorch` — having defined the problem-specific functions to evaluate the DE, BCs, and ICs — the loss function for a one-dimensional DE with one BC and one IC is obtained as follows:

```
def loss_function(t,x,Ny,f,By,g,Cy,h,beta,gamma):
    # Loss domain
    t_N, x_N = select the points (t, x) inside the domain
    Ny_pred = Ny(t_N, x_N) # net approximation of the DE
    f_true = f(t_N, x_N) # true RHS of DE
    loss_PDE = torch.mean((Ny_pred - f_true)**2)
    # Loss BC
    t_B = select the points (t, x) at the boundary
    By_pred = By(t_B) # net approximation of the BC
    g_true = g(t_B) # true RHS of BC
    loss_BC = beta * torch.mean((Ny_pred - f_true)**2)
    # Loss IC
    x_C = select the points (t, x) at the initial time
    Cy_pred = Cy(x_C) # net approximation of the IC
    h_true = h(x_C) # true RHS of IC
    loss_IC = gamma * torch.mean((Cy_pred - h_true)**2)
    # Total loss
    return loss_PDE + loss_BC + loss_IC
```

Note that, unlike classical numerical methods, NNs do not guarantee convergence to a unique solution [2] because they are solving a non-convex optimization problem, which in general might have many local minima.

2.3. Automatic differentiation (AD)

Since $\mathcal{N}$ and $\mathcal{B}$ are differential operators, to evaluate them we will need to calculate derivatives of $\hat{y}(t, \boldsymbol{x}|\theta)$ with respect to t and x_i. This is done by applying the AD technique, which uses the fact that the FNN represents a compositional function. It consists of applying the chain rule to each of the paths $\mathcal{P}_r = \left\{t/x_i, h_{r_2}^{(2)}, h_{r_3}^{(3)}, ..., h_{r_{N-1}}^{(N-1)}, \hat{y}\right\}$,

connecting t/x_i to $\hat{y}(t, \boldsymbol{x}|\theta)$, and summing up. Here, $h_{r_l}^{(l)}$ is the node at layer l chosen for path r. In fig. 1, these paths are all the sets of arrows we can follow to get from t/x_i to $\hat{y}(t, \boldsymbol{x}|\theta)$. Hence,

$$\frac{\partial \hat{y}(t, \boldsymbol{x}|\theta)}{\partial t} = \sum_{r=1}^{R} \left[\frac{\partial \hat{y}(t, \boldsymbol{x}|\theta)}{\partial h_{r_{N-1}}^{(N-1)}} \prod_{n=2}^{N-2} \left(\frac{\partial h_{r_{n+1}}^{(n+1)}}{\partial h_{r_n}^{(n)}} \right) \frac{h_{r_2}^{(2)}}{\partial t} \right], \tag{5a}$$

$$\frac{\partial \hat{y}(t, \boldsymbol{x}|\theta)}{\partial x_i} = \sum_{r=1}^{R} \left[\frac{\partial \hat{y}(t, \boldsymbol{x}|\theta)}{\partial h_{r_{N-1}}^{(N-1)}} \prod_{n=2}^{N-2} \left(\frac{\partial h_{r_{n+1}}^{(n+1)}}{\partial h_{r_n}^{(n)}} \right) \frac{h_{r_2}^{(2)}}{\partial x_i} \right], \tag{5b}$$

where R is the number of possible paths. Here, the expression within the brackets is the chain rule applied to each $\mathcal{P}_r$, and all partial derivatives on the right-hand-side can be easily calculated using eq. (1). This can be similarly extended to higher-order derivatives. Hence, for the NN to work we must ensure Φ_l is differentiable at least $m+1$ times before it becomes zero everywhere, where m is the order of the DE.

AD in `PyTorch` for a one-dimensional DE is done as follows

```
y = nn(torch.cat([t, x], dim=1)) # outputs of the net
y_t = torch.autograd.grad(y, t, torch.ones_like(y),
    create_graph=True)[0] # first derivative in time
y_x = torch.autograd.grad(y, x, torch.ones_like(y),
    create_graph=True)[0] # first derivative in space
y_xx = torch.autograd.grad(y_x, x, torch.ones_like(y_x),
    create_graph=True)[0] # second derivative in space
```

Moreover, to minimise $\mathcal{L}(\theta|T)$, gradient-based optimisers will need to calculate its derivatives with respect to the weights $w_{i,j}^{(l)}$ and biases $b_i^{(l)}$. This is done similarly to eqs. (5a) and (5b), but now paths start at the hidden unit $h_i^{(l+1)}$ (not at t/x_i), i.e. $\hat{\mathcal{P}}_r = \left\{ h_i^{(l+1)}, h_{r_{l+2}}^{(l+2)}, h_{r_{l+3}}^{(l+3)}, ..., h_{r_{N-1}}^{(N-1)}, \hat{y} \right\}$.

$$\frac{\partial \mathcal{L}(\theta|T)}{\partial b_i^{(l)}} = \frac{\partial \mathcal{L}(\theta|T)}{\partial \hat{y}} \sum_{r=1}^{\hat{R}} \left[\frac{\partial \hat{y}(t, \boldsymbol{x}|\theta)}{\partial h_{r_{N-1}}^{(N-1)}} \prod_{n=l+1}^{N-2} \left(\frac{\partial h_{r_{n+1}}^{(n+1)}}{\partial h_{r_n}^{(n)}} \right) \right] \frac{h_i^{(l+1)}}{\partial b_i^{(l)}}, \tag{6a}$$

$$\frac{\partial \mathcal{L}(\theta|T)}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(\theta|T)}{\partial \hat{y}} \sum_{r=1}^{\hat{R}} \left[\frac{\partial \hat{y}(t, \boldsymbol{x}|\theta)}{\partial h_{r_{N-1}}^{(N-1)}} \prod_{n=l+1}^{N-2} \left(\frac{\partial h_{r_{n+1}}^{(n+1)}}{\partial h_{r_n}^{(n)}} \right) \right] \frac{h_i^{(l+1)}}{\partial w_{i,j}^{(l)}}, \tag{6b}$$

where all partial derivatives on the left-hand side can be easily calculated using eqs. (1) and (4a) to (4d). This instance of AD is called backpropagation.

3 Ordinary Differential Equations (ODEs) Numerical Experiments

3.1. Methods

Since there are no restrictions to the nonlinearity and nonconvexity of $\mathcal{L}(\theta|T)$, we use gradient-based optimisers to minimise it. In particular, we choose a hybrid optimisation strategy using the Adam optimiser [5] for an initial number of epochs followed by L-BFGS [6]. This strategy aims to balance computational efficiency and accuracy. Adam is a first-order optimiser that offers a highly efficient and inexpensive way of exploring the loss landscape. However, as the optimisation progresses, it tends to oscillate around the optimal solution. Then, the quasi-Newton optimiser L-BFGS is introduced, bringing improved convergence.

We specify the training points to be uniformly spaced throughout the domain and never change them during the training process, so on `Pytorch`, the training is

```
def train(nn,a,b,sample_size,Ny,f,By,g,beta,Adam_iter,LBFGS_iter):
    nn.train()
    x = torch.linspace(a,b,sample_size,requires_grad=True).unsqueeze(1)
    optimiser_Adam = torch.optim.Adam(nn.parameters())
    for epoch_Adam in range(Adam_iter): # train with Adam optimiser
        loss = loss_function(x,Ny,f,By,g,beta) # loss
        optimiser_Adam.zero_grad() # gradients set to zero
        loss.backward() # backpropagation
        optimiser_Adam.step() # update parameters
    optimiser_LBFGS = torch.optim.LBFGS(nn.parameters())
    def closure():
        loss = loss_function(x,Ny,f,By,g,beta) # loss
        optimiser_LBFGS.zero_grad() # gradients set to zero
        loss.backward() # backpropagation
        return loss
    for epoch_LBFGS in range(LBFGS_iter):
        optimiser_LBFGS.step(closure) # update parameters
```

Finally, we choose the sigmoid activation function $\Phi_l(\boldsymbol{z})_i = (1 + e^{-z_i})^{-1}$, which squishes a large input space $z_i \in [-\infty, \infty]$ into a small output space $\Phi_l(\boldsymbol{z})_i \in [0, 1]$. Consequently, a large change in $\boldsymbol{z}$ will cause a small change in $\Phi_l(\boldsymbol{z})$, and $\frac{\partial \Phi_l(\boldsymbol{z})}{\partial z_i}$ tends to 0 as z_i tends to $\pm\infty$. We will return to this topic when we look at NN depth.

In the following subsection, we will explore how the method for setting the BCs, the width and depth of the net, and the number of grid points affect the training of the NN when solving an ODE. We do so in terms of the convergence of the loss, the accuracy of the approximation, and the time taken for training to be completed. To quantify the accuracy, we calculate the average point-wise error of the result $\hat{y}(x|\theta)$ with respect to the exact solution $y(x)$ at 100 points,

$$\text{error} = \frac{1}{100}\sum_{i}^{100}\frac{|\hat{y}(x^{(i)}|\theta) - y(x^{(i)})|}{|y(x^{(i)})|}. \tag{7}$$

3.2. Example 1: second-order linear ODEs with constant coefficients

Consider solving the second-order linear ODE

$$\mathcal{N}y(x) = c_2\frac{\mathrm{d}^2y(x)}{\mathrm{d}x} + c_1\frac{\mathrm{d}y(x)}{\mathrm{d}x} + c_0y(x) = f(x) \qquad \text{in } a < x < b, \tag{8a}$$

$$\mathcal{B}_ay(a) = y_a, \qquad \mathcal{B}_by(b) = y_b, \tag{8b}$$

where $c_i \in \mathbb{R}$. Then $\mathcal{N}y(x)$ can be evaluated as

```
def Ny(nn,x,c0,c1,c2):
    y = nn(x) # outputs of the net
    y_x = torch.autograd.grad(y, x, grad_outputs=torch.ones_like(y),
          create_graph=True)[0] # first derivative
    y_xx = torch.autograd.grad(y_x, x, grad_outputs=torch.ones_like(y_x),
           create_graph=True)[0] # second derivative
    return c2*y_xx + c1*y_x + c0*y
```

and Dirichlet/Von Neumann BCs as

```
def By(nn,x,BCs): # where x = [a, ..., b]
    y.nn(x)
    if BCs == "dirichlet": # Set Dirichlet BCs
        return torch.tensor([y[0], y[-1]])
    elif BCs == "neumann": # Set Neumann BCs
        y_x = torch.autograd.grad(y, x, grad_outputs=torch.ones_like(y),
              create_graph=True)[0]
        return torch.tensor([y_x[0], y_x[-1]])
    else: raise ValueError("Invalid BCs")
```

In the upcoming numerical experiments, we will try to approximate a 1D Poisson equation with Dirichlet BCs

$$\frac{\mathrm{d}^2 y(x)}{\mathrm{d}x} = -2 \quad \text{in } 0 < x < 1, \quad y(0) = y(1) = 1, \tag{9}$$

and exact solution $y(x) = 1 + x(1-x)$, and a 1D Helmholtz equation with mixed BCs

$$\frac{\mathrm{d}^2 y(x)}{\mathrm{d}x} + y(x) = 0 \quad \text{in } -\pi < x < \pi, \quad y(-\pi) = -1, \quad \frac{\mathrm{d}y(\pi)}{\mathrm{d}x} = 1. \tag{10}$$

and exact solution $y(x) = \cos(x) - \sin(x)$.

3.2.1 Boundary conditions

The most common approach to applying the BCs is the one described until now, where the boundary constraints are added to the loss function (eq. (4c)) with a certain weight β. However, there are alternative approaches, like post-processing the net outputs to automatically satisfy the BCs. For Dirichlet BCs (eq. (9)) a way of doing so is

$$\bar{y}(x) = \hat{y}(x) + \frac{(b-x)}{(b-a)}(y_a - \hat{y}(a)) + \frac{(x-a)}{(b-a)}(y_b - \hat{y}(b)), \tag{11}$$

so $\bar{y}(a) = y_a$ and $\bar{y}(b) = y_b$ automatically. Similarly for mixed BCs (eq. (10))

$$\bar{\bar{y}}(x) = \hat{y}(x) + (y_a - \hat{y}(a)) + (x-a)\left(y_b - \frac{\partial \hat{y}(b)}{\partial x}\right), \tag{12}$$

so $\bar{\bar{y}}(a) = y_a$ and $\frac{\partial \bar{\bar{y}}(b)}{\partial x} = y_b$ automatically. Note that the boundary terms in eq. (4c) for $\bar{y}(x)$ and $\bar{\bar{y}}(x)$ vanish and we no longer need to define a β.

Figure 2 compares the classical approach with different values of β and the post-processing method. For eq. (9), both are similar in terms of accuracy and speed. However, for eq. (10), the postprocessing approach faced some convergence issues that forced us to reduce the learning rate of Adam and abandon L-BFGS. As a result, it converges to a local minimum that is not exactly the correct solution. Thus, we will use the first method for the rest of the section. While any choice of β between 1 and 70 gives similar accuracy, larger values increase the training time and the fluctuations in the convergence. Hence, we choose to set $\beta = 1$.

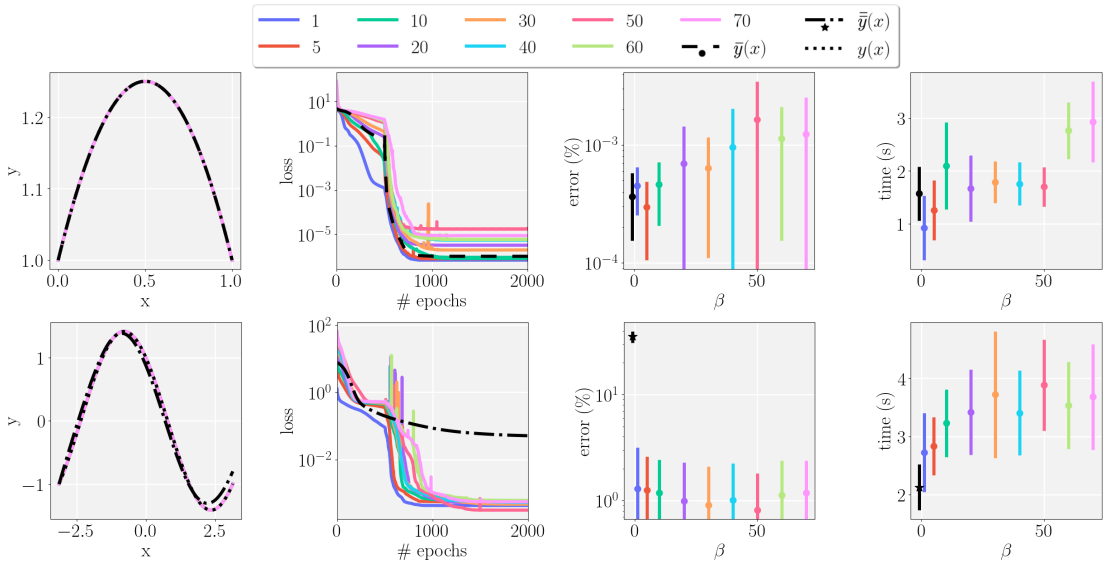

Fig. 2: Effect of β on the training of the net. The NN consisted of one hidden layer with five nodes. The problems learnt were eq. (9) (top row) and eq. (10) (bottom row). The results of the classical (solid lines) and post-processing (dashed lines) methods are the average of 20 trial runs. These are verified against the exact solutions (dotted-black lines). The error bars correspond to one standard deviation. The number of grid points was 100. The optimiser switched from Adam (learning rate of 0.02) to L-BFGS at epoch 500 except for $\bar{\bar{y}}(x)$, for which we only used Adam (learning rate of 0.005).

3.2.2 Width and Depth

Width. Figure 3 shows that increasing the number of nodes per hidden layer improves the convergence and accuracy of the NN. This is equivalent to increasing the number of parameters θ. At first, θ are so few that the NN does not have enough expressivity to learn the function, and convergence and accuracy improve significantly with each node added. However, when the NN has enough representational power, increasing the width does not substantially improve the results. It is also clear that the number of nodes needed to reach enough expressivity increases with the complexity of the problem. In this case, around 5 nodes for eq. (9) and around 9 for eq. (10). A trend between the width and the training time is unclear from these graphs.

Depth. Figure 4 shows that increasing the number of hidden layers of the NN makes the training time longer, and can improve but generally worsens convergence and accuracy. Recall from section 3.1 that the derivatives of the sigmoid function with respect to its input can become very small. This is not an issue with a few layers, but

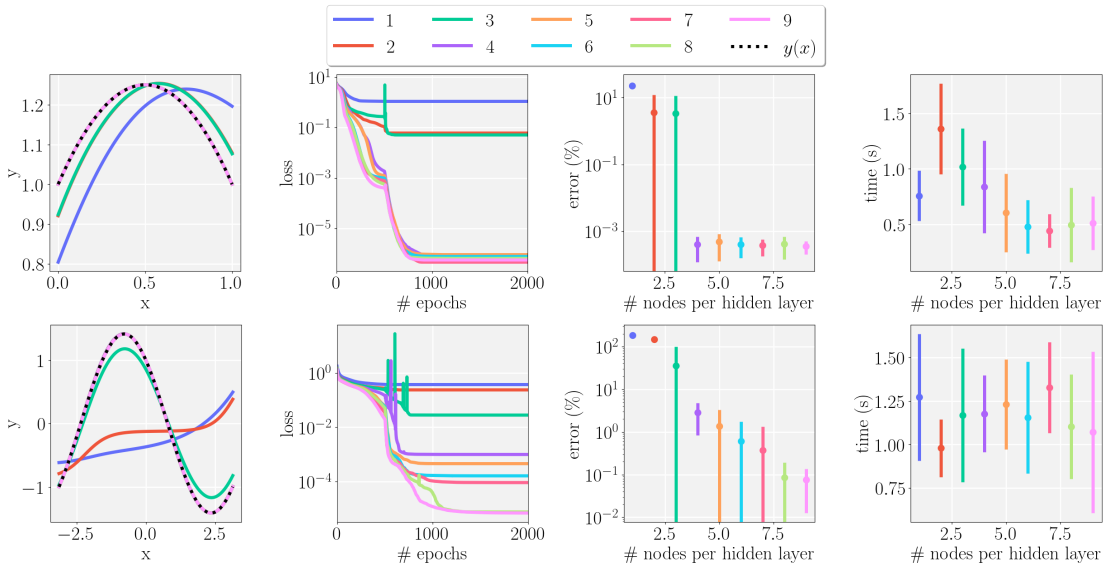

Fig. 3: Effect of the number of nodes per hidden layer (width) on the training of the net. The NN consisted of one hidden layer with a varying number of nodes. The problems learnt were eq. (9) (top row) and eq. (10) (bottom row). The results for each number of nodes (solid lines) are the average of 20 trial runs. These are verified against the exact solutions (dotted-black lines). The error bars correspond to one standard deviation. The number of grid points was 100 and $\beta = 1$. The optimiser switched from Adam (learning rate of 0.02) to L-BFGS at epoch 500.

as depth increases, eqs. (6a) and (6b) obtained via backpropagation become sums of products of very small derivatives. This means that $\frac{\partial \mathcal{L}(\theta|T)}{\partial b_i^{(L)}}$ and $\frac{\partial \mathcal{L}(\theta|T)}{\partial w_{i,j}^{(L)}}$ will be very small for the first layers and the weights and biases of these initial layers will not be updated efficiently leading to overall network inaccuracy. Hence, as soon as the NN is expressive enough to learn the function, increasing the number of hidden layers only leads to a higher risk of vanishing gradients, hindering the training of the NN.

Discussion. This section shows that a small network consisting of a few layers and nodes is sufficient to approximate an ODE. In fact, we have seen that increasing θ can only improve convergence and efficiency until a certain point dependent on the complexity of the problem. Moreover, if θ is increased by making the NN deeper, it can quickly worsen the approximation. These results align with the general view that wider and shallower networks work better for these problems.

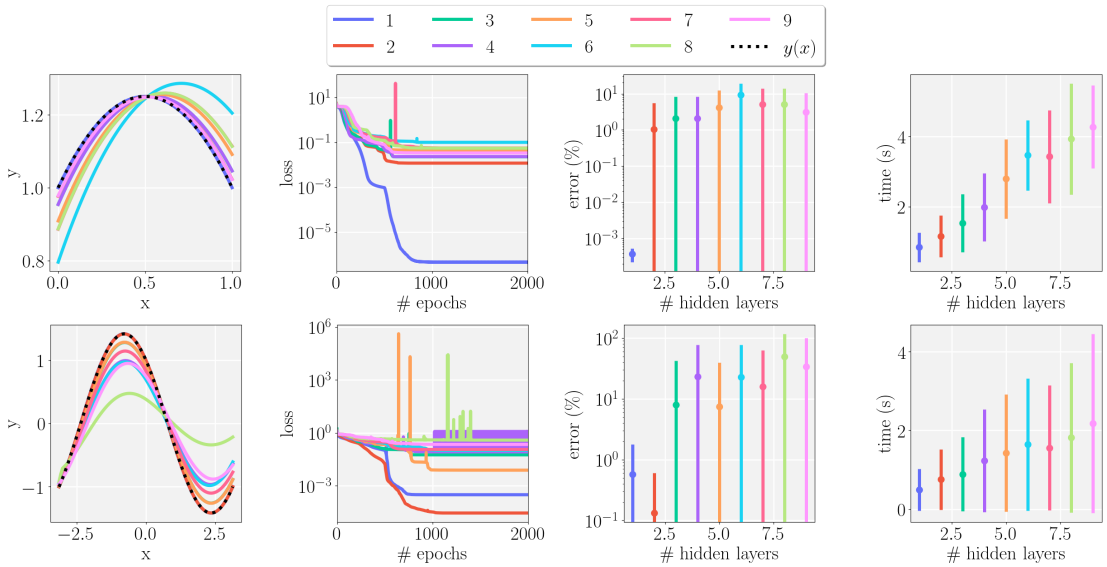

Fig. 4: Effect of the number of hidden layers (depth) on the training of the net. The NN consisted of five nodes per hidden layer and a varying number of layers. The problems learnt were eq. (9) (top row) and eq. (10) (bottom row). The results for each number of layers (solid lines) are the average of 20 trial runs. These are verified against the exact solutions (dotted-black lines). The error bars correspond to one standard deviation. The number of grid points was 100 and $\beta = 1$. The optimiser switched from Adam (learning rate of 0.02) to L-BFGS at epoch 500.

3.2.3 Grid points

Figure 5 shows that increasing the number of grid points leads to more accurate results at the expense of slower training. Note that the apparent worsening of the convergence does not result from a poorer approximation but from the loss having to be minimised at a larger number of grid points. Hence, this is an instance where the values of the loss misrepresent the accuracy of the final solution, and we should instead refer to the error to compare solutions. Nonetheless, the behaviour of the loss is still relevant to know whether the NN converges (the loss decreases) or not (the loss remains flat).

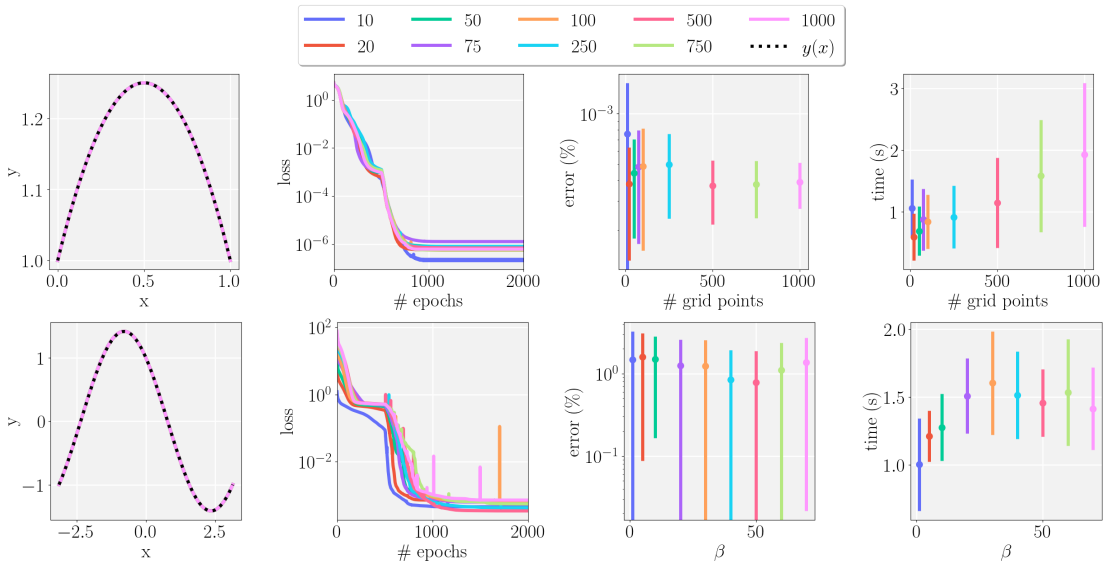

Fig. 5: Effect of the number of grid points on the training of the net. The NN consisted of one hidden layer with ten nodes. The problems learnt were eq. (9) (top row) and eq. (10) (bottom row). The results for each number of layers (solid lines) is the average of 20 trial runs. These are verified against the exact solutions (dotted-black lines). We set $\beta = 1$. The optimiser switched from Adam (learning rate of 0.02) to L-BFGS at epoch 500.

3.3. Example 2: linear system of second-order ODEs

The previous techniques can be easily extended to a system of linear ODEs

$$\mathcal{N}(\boldsymbol{y}) = \begin{cases} c_2^{(1)}(x)\frac{\mathrm{d}^2 y_1(x)}{\mathrm{d}x} + c_1^{(1)}(x)\frac{\mathrm{d}y_1(x)}{\mathrm{d}x} + c_0^{(1)}(x)y_1(x) = f_1(x, y_2) \\ c_2^{(2)}(x)\frac{\mathrm{d}^2 y_2(x)}{\mathrm{d}x} + c_1^{(2)}(x)\frac{\mathrm{d}y_2(x)}{\mathrm{d}x} + c_0^{(2)}(x)y_2(x) = f_2(x, y_1) \end{cases} \quad \text{in } a < x < b, \tag{13a}$$

$$\mathcal{B}_{1a}y_1(a) = y_{1a}, \qquad \mathcal{B}_{1b}y_1(b) = y_{1b}, \qquad \mathcal{B}_{2a}y_2(a) = y_{2a}, \qquad \mathcal{B}_{2b}y_2(b) = y_{2b}. \tag{13b}$$

where $c_i(x)$ are differentiable functions. The `PyTorch` implementation is exacly the same as before but with an extra dimension in the output of the net ($\hat{\boldsymbol{y}} = (y_1, y_2)$), `Ny` ($\mathcal{N}y_1$ and $\mathcal{N}y_2$), and `By` ($[\mathcal{B}_{1a}y_1(a), \mathcal{B}_{1b}y_1(b)]$ and $[\mathcal{B}_{2a}y_1(a), \mathcal{B}_{1b}y_1(b)]$). Moreover, the loss is now the sum of the losses for y_1 and y_2.

Figure 6 shows the results of approximating the example problem

$$\begin{cases} \frac{\mathrm{d}^2 y_1(x)}{\mathrm{d}x^2} + xy_1(x) - y_2(x) = 2 + x, & y_1(0) = y_1(1) = 1, \\ -\frac{\mathrm{d}^2 y_2(x)}{\mathrm{d}x^2} + y_2(x) = 6x - 2 + x^2(1 - x), & y_2(0) = y_2(1) = 0. \end{cases} \tag{14}$$

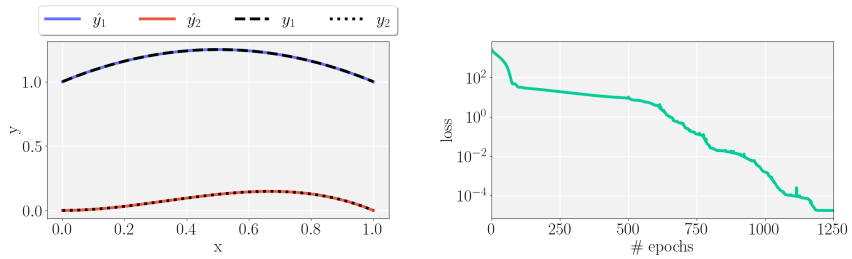

Fig. 6: NN approximation of eq. (14). The NN consisted of one hidden layer with ten nodes. The result of one run is verified against the exact solutions (black lines). The number of grid points was 100 and $\beta = 1$. The optimiser switched from Adam (learning rate of 0.02) to L-BFGS at epoch 500.

3.4. Example 3: second-order non-linear ODEs

Again, the previous techniques can be easily extended to solve non-linear ODEs. In this case, we must change the function `Ny` to be the respective non-linear left-hand side. Figure 7 shows the results of trying to approximate the problem

$$\frac{\mathrm{d}^2 y(x)}{\mathrm{d}x^2} - \left(\frac{\mathrm{d}y(x)}{\mathrm{d}x}\right)^2 y(x) = 10\cos(10x^2)e^{5x^2} \qquad \text{in } x \in [-1, 1], \tag{15}$$

with Dirichlet BCs ($y(-1) = y(1) = 0$) and mixed BCs ($y(-1) = \frac{\partial y(1)}{\partial x} = 0$). Since the problem was harder, we required more width, depth, and trial point density to solve it, and the solution was not as accurate as before.

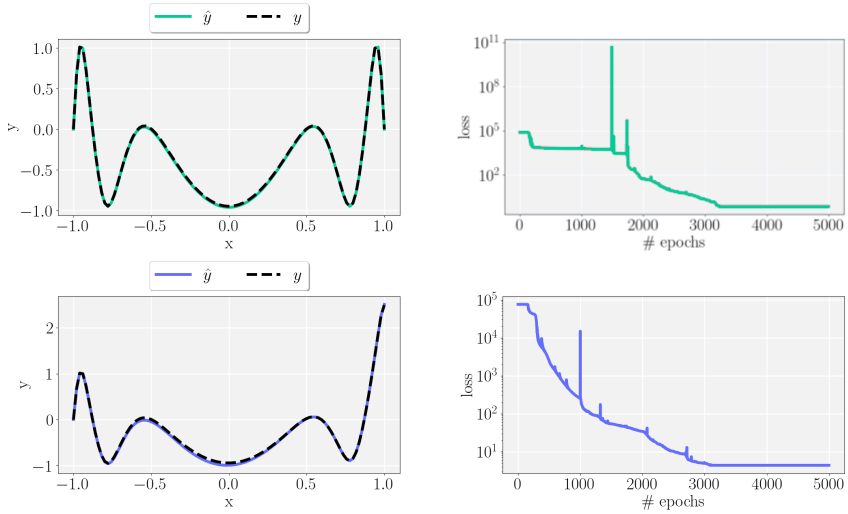

Fig. 7: NN approximation of eq. (15). The NN consisted of three hidden layers with 20 nodes. The results of one run for Dirichlet (green) and mixed (blue) BCs are verified against the exact solutions (black lines). The number of grid points was 250 and $\beta = 750$. The optimiser switched from Adam (learning rate of 0.02) to L-BFGS at epoch 1500 and 1000, respectively.

4 Partial Differential Equations (PDEs) Numerical Experiments

4.1. Methods

For PDEs, we use a slightly different training strategy where, instead of switching to using L-BFGS after a certain amount of Adam epochs, we reduce the learning rate λ_k of the Adam optimiser at each iteration k as

$$\lambda_k = \kappa \lambda_{k-1}, \tag{16}$$

where $\kappa \in [0, 1]$. This strategy aims to keep the oscillations of this optimiser in check by reducing its learning rate fast enough. Moreover, we specify different training points at each optimisation iteration by selecting them randomly throughout the domain (24/25ths of the points) and boundaries (1/25th of the points).

Since PDEs usually require deeper nets, we choose a hyperbolic-tangent activation function $\Phi_l(\boldsymbol{z})_i = \frac{e^{z_i} - e^{-z_i}}{e^{z_i} + e^{-z_i}}$. This activation function is very similar to the classical sigmoid function but it squishes the input space $z_i \in [-\infty, \infty]$ into a slightly larger output space $\Phi_l(\boldsymbol{z})_i \in [-1, 1]$. Consequently, its derivatives will be larger than the sigmoid activation function, reducing the danger of vanishing gradients.

4.2. Example 1: 2D Poisson equation

Consider solving the 2D Poisson equation

$$-\nabla^2 y(\boldsymbol{x}) = f(\boldsymbol{x}) \qquad \text{in } \Omega. \tag{17}$$

The `PyTorch` implementation is analogous to that of ODEs but with an extra dimension in the input of the net ($\hat{\boldsymbol{x}} = (x_1, x_2)$). Additionally, one must watch the dimensions of the torch arrays, and that they are taking the right points when evaluating the PDE and BCs.

In particular, consider the specific problem

$$-\nabla^2 y(x_1, x_2) = 2\pi^2 \sin(\pi x_1) \sin(\pi x_2) \qquad \text{in } (x_1, x_2) \in ([0, 1], [0, 1]), \tag{18a}$$

$$y(0, x_2) = y(1, x_2) = 0 \qquad \text{on } x_2 \in [0, 1], \tag{18b}$$

$$y(x_1, 0) = y(x_1, 1) = 0 \qquad \text{on } x_1 \in [0, 1], \tag{18c}$$

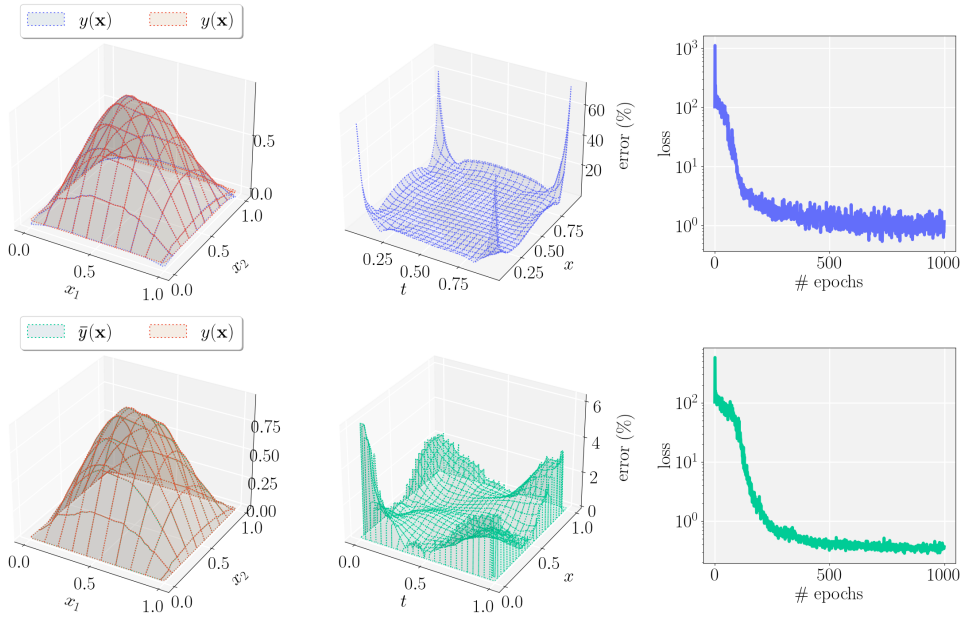

Fig. 8: Solving the 2D Poisson equation. The NN had three hidden layers with 20 nodes each. The result of one run (blue for $\hat{y}$, green for $\bar{y}$) is verified against the exact solution (red). The number of training points was 10000, and the Adam optimiser had $\lambda_0 = 0.05$, which decayed at each iteration following eq. (16) with $\kappa = 0.995$. For $\hat{y}$, $\beta = 200$ and the average error was 27.5%. For $\bar{y}$, the average error was 0.68%.

with exact solution $y(\boldsymbol{x}) = \sin(\pi x_1)\sin(\pi x_2)$.

Figure 8 (top row) shows that the biggest contribution to the error comes from the points at the edges and, if they are not considered, the average error quickly reduces below 1%. A solution to this problem in the approximation is to post-process $\hat{y}$ to automatically satisfy the BCs in a similar way to what was discussed in section 3.2.1. In the case of this PDE, $\bar{y}$ would be obtained as follows

$$\begin{aligned}\bar{y}(x_1, x_2) =& \hat{y}(x_1, x_2) - (1 - x_1)\hat{y}(0, x_2) - x_1\hat{y}(1, x_2) - (1 - x_2)\hat{y}(x, 0) - x_2\hat{y}(x, 1) \\ &+ (1 - x_1)(1 - x_2)\hat{y}(0, 0) + x_1(1 - x_2)\hat{y}(1, 0) \\ &+ (1 - x_1)x_2\hat{y}(0, 1) + x_1x_2\hat{y}(1, 1). \end{aligned} \tag{19}$$

The results for $\bar{y}(\boldsymbol{x})$ are also shown in fig. 8. This post-processing notably reduces the error at the edges giving an excellent approximation to the problem.

Overall, the loss fluctuates significantly more than with ODEs, reflecting the increased complexity of the problem and its loss landscape. This complexity makes it challenging for the optimiser to find the path towards the minimum.

4.3. Example 2: Burger's equation

In this section, we will solve Burger's equation; a convection-diffusion equation that arises in various areas of applied mathematics.

4.3.1 One-dimensional Burger's equation

In one spatial dimension, this equation is

$$\frac{\partial y(t,x)}{\partial t} + y(t,x)\frac{\partial y(t,x)}{\partial x} - \upsilon\frac{\partial^2 y(t,x)}{\partial x^2} = 0 \qquad \text{in } \Omega. \tag{20}$$

where υ is a parameter we refer to as viscosity. Classical numerical methods struggle to approximate this equation for small values of υ as they can lead to shock formation. We will examine the specific problem

$$\frac{\partial y(t,x)}{\partial t} + y(t,x)\frac{\partial y(t,x)}{\partial x} - \frac{0.01}{\pi}\frac{\partial^2 y(t,x)}{\partial x^2} = 0 \qquad \text{in } (t,x) \in ([0,1],[-1,1]), \tag{21a}$$

$$y(t,-1) = y(t,1) = 0, \qquad \text{in } t \in [0,1], \tag{21b}$$

$$y(0,x) = -\sin(\pi x) \qquad \text{in } x \in [-1,1], \tag{21c}$$

with exact solution

$$y(t,x) = \frac{-\int_{-\infty}^{\infty} \sin\pi(x-\eta)\exp\left[-\cos\frac{\pi(x-\eta)}{2\pi\upsilon} - \frac{\eta^2}{4\upsilon t}\right]\mathrm{d}\eta}{\int_{-\infty}^{\infty}\exp\left[-\cos\frac{\pi(x-\eta)}{2\pi\upsilon} - \frac{\eta^2}{4\upsilon t}\right]\mathrm{d}\eta}, \tag{22}$$

which can solved numerically using Hermite integration [7]. This solution becomes a sawtooth wave at $x = 0$ at $t \simeq \pi$, reaching a maximum value for the gradient at $t \simeq 0.5$ [1, 7]. Hence, to approximate this problem, we chose to force at least 1/3rd of the trial points in the domain to be within $x \in [-0.25, 0.25]$. Figure 9 shows very reasonable results where, regardless of our sampling method, the biggest contribution to the error still comes from the sawtooth crest.

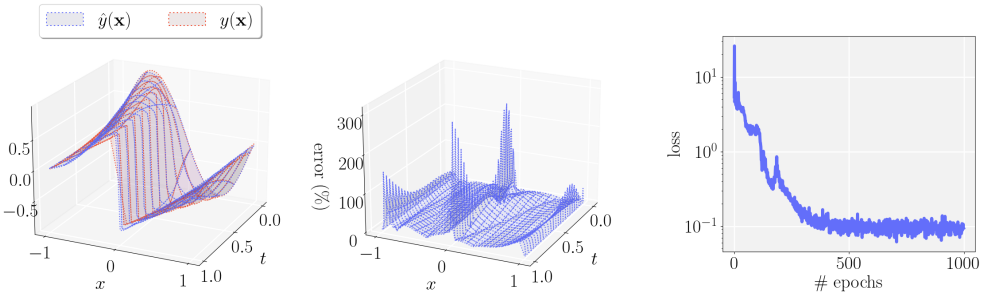

Fig. 9: Solving the 1D Burger's equation. The NN consisted of eight hidden layers with 20 nodes. The result of one run (blue) is verified against the data provided by [1] (red). The number of training points was 10000, $\beta = \gamma = 10$, and the Adam optimiser had $\lambda_0 = 0.05$, which decayed at each iteration following eq. (16) with $\kappa = 0.99$. The average error was 16.7%.

4.3.2 Two-dimensional Burger's equation

In two spatial dimensions, this equation is

$$\begin{cases} \frac{\partial y_1}{\partial t} + y_1 \frac{\partial y_1}{\partial x_1} + y_2 \frac{\partial y_1}{\partial x_2} = \frac{1}{\mathrm{Re}} \left(\frac{\partial^2 y_1}{\partial x_1^2} + \frac{\partial^2 y_1}{\partial x_2^2} \right) \\ \frac{\partial y_2}{\partial t} + y_1 \frac{\partial y_2}{\partial x_1} + y_2 \frac{\partial y_2}{\partial x_2} = \frac{1}{\mathrm{Re}} \left(\frac{\partial^2 y_2}{\partial x_1^2} + \frac{\partial^2 y_2}{\partial x_2^2} \right) \end{cases} \quad \text{in } \Omega, \tag{23}$$

where we left the $(\boldsymbol{x}, t)$ dependencies implicit for readibility. In the domain $(x_1, x_2) \in ([0,1],[0,1])$ this equation admits the solution [8]

$$\tilde{y}_1(\boldsymbol{x}, t) = \frac{3}{4} - \frac{1}{4\left[1 + e^{\mathrm{Re}(4x_2 - 4x_1 - t)/32}\right]}, \qquad \tilde{y}_2(\boldsymbol{x}, t) = \frac{3}{4} + \frac{1}{4\left[1 + e^{\mathrm{Re}(4x_2 - 4x_1 - t)/32}\right]}. \tag{24}$$

Taking the BCs and ICs from eq. (24) and setting $\mathrm{Re} = 150$ we get the problem

$$\begin{cases} \frac{\partial y_1}{\partial t} + y_1 \frac{\partial y_1}{\partial x_1} + y_2 \frac{\partial y_1}{\partial x_2} = \frac{1}{150} \left(\frac{\partial^2 y_1}{\partial x_1^2} + \frac{\partial^2 y_1}{\partial x_2^2} \right) \\ \frac{\partial y_2}{\partial t} + y_1 \frac{\partial y_2}{\partial x_1} + y_2 \frac{\partial y_2}{\partial x_2} = \frac{1}{150} \left(\frac{\partial^2 y_2}{\partial x_1^2} + \frac{\partial^2 y_2}{\partial x_2^2} \right) \end{cases} \quad \text{in } x_1, x_2, t \in [0,1], \tag{25a}$$

$$y_i(t, 0, x_2) = \tilde{y}_1(t, 0, x_2), \quad y_i(t, 1, x_2) = \tilde{y}_i(t, 1, x_2) \quad \text{in } x_2, t \in [0,1], \tag{25b}$$

$$y_i(t, x_1, 0) = \tilde{y}_i(t, x_1, 0), \quad y_i(t, x_1, 1) = \tilde{y}_i(t, x_2, 1) \quad \text{in } x_1, t \in [0,1], \tag{25c}$$

$$y_i(0, x_1, x_2) = \tilde{y}_i(0, x_1, x_2) \quad \text{in } x_1, x_2 \in [0,1], \tag{25d}$$

where $i = 1, 2$. This can be solved using an NN with three inputs $\boldsymbol{x} = (t, x_1, x_2)$ and two outputs $\hat{\boldsymbol{y}} = (y_1, y_2)$.

Figure 10 shows that, while the NN manages to capture the initial condition, the

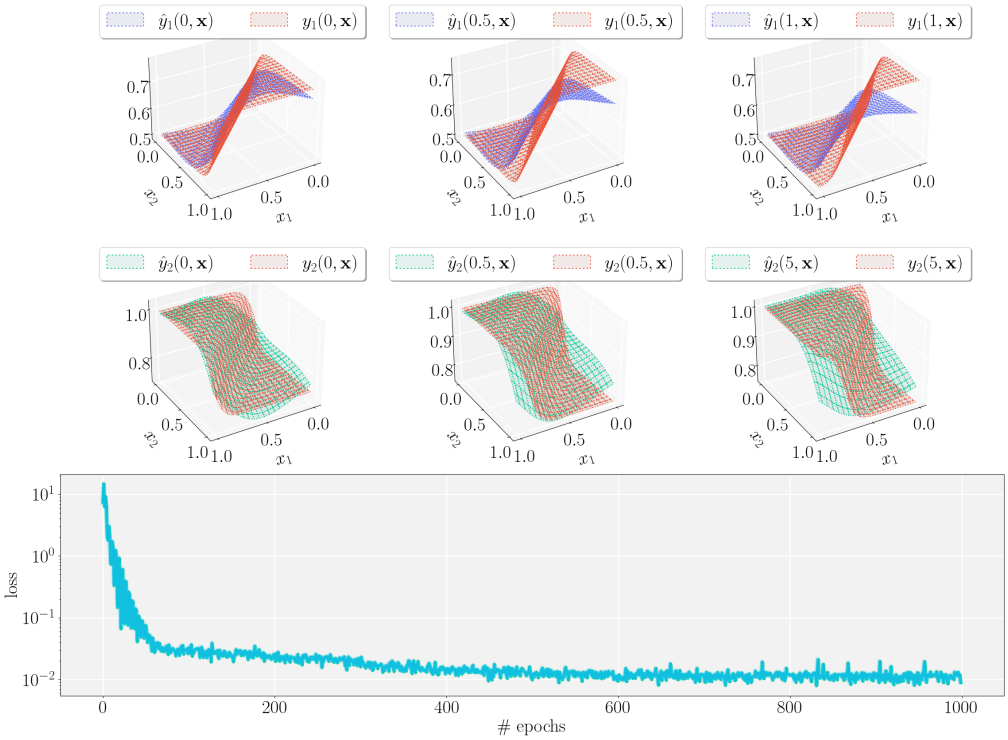

Fig. 10: Solving the 2D Burger's equation. The NN consisted of two hidden layers with 20 nodes. The result of one run (blue for $\hat{y}_1$, green for $\hat{y}_2$) is verified against the exact solution (red). The number of training points was 10000, $\beta = \gamma = 1$, and the Adam optimiser had $\lambda_0 = 0.05$, which decayed at each iteration following eq. (16) with $\kappa = 0.995$. The average errors (of $\hat{y}_1$ and $\hat{y}_2$ together) were 3.9%, 5.6%, and 6.7% for $t = 0$, $t = 0.5$, and $t = 1$, respectively.

quality of the solution deteriorates over time, making our results unsatisfactory. The main reason for our poor approximation lies in the computational cost associated with performing an exhaustive exploration of the NN hyperparameters — net depth and width, β, γ, $\Phi_i(\cdot)$, etc. —, the optimiser — Adam, L-BFGS, etc. — and its hyperparameters — λ_0, κ, etc. —, as well as the sampling procedure, which proved unfeasible. Instead, we tried only a few configurations from which the best one gave the results shown. In future research, a more thorough inspection should be carried out, or more sophisticated optimisation and sampling techniques that admit a wider range of hyperparameters should be used.

4.4. Example 3: Derivative nonlinear Schrodinger's equation

In this section we will solve the derivative nonlinear Schrödinger equation; used in the study of quantum mechanical systems, especially in space plasma physics and nonlinear optics. In one spatial dimension, this equation is

$$\frac{\partial h(t,x)}{\partial t} + i\frac{\partial^2 h(t,x)}{\partial x^2} + \frac{\partial(|h|^2 h)}{\partial x} = 0, \tag{26}$$

which gives complex-valued solutions $h(t,x)$. It admits the solution [9]

$$\tilde{h}(t,x) = \frac{4e^{2i(2t-x)}(4i(4t-x)-1)^3}{(16(4t-x)^2+1)^2}, \tag{27}$$

so we consider the problem

$$\frac{\partial h(t,x)}{\partial t} + i\frac{\partial^2 h(t,x)}{\partial x^2} + \frac{\partial(|h|^2 h)}{\partial x} = 0 \qquad \text{in } (t,x) \in ([-0.08, 0.08], [-5,5]), \tag{28a}$$

$$h(t,-5) = \tilde{h}(t,-5) \qquad \text{in } t \in [-0.08, 0.08], \tag{28b}$$

$$h(t,5) = \tilde{h}(t,5) \qquad \text{in } t \in [-0.08, 0.08], \tag{28c}$$

$$h(-0.08,x) = \tilde{h}(-0.08,x) \qquad \text{in } x \in [-5,5]. \tag{28d}$$

Complex-valued solutions can be decomposed into two real functions $h(t,x) = u(t,x) + iv(t,x)$. Hence, eq. (28a) is equivalent to solving a system of PDEs

$$\begin{cases} \frac{\partial u}{\partial t} - \frac{\partial^2 v}{\partial x^2} + (3u^2+v^2)\frac{\partial u}{\partial x} + 2vu\frac{\partial v}{\partial x} = 0, \\ \frac{\partial v}{\partial t} + \frac{\partial^2 u}{\partial x^2} + (3v^2+u^2)\frac{\partial v}{\partial x} + 2vu\frac{\partial u}{\partial x} = 0, \end{cases} \tag{29}$$

where we left the (t,x) dependencies implicit for readability, and with the respective BCs corresponding to eqs. (28b) to (28d). This can be solved using a multi-output NN giving $\hat{h} = [\hat{u}, \hat{v}]$ and the results are shown in fig. 11. The NN manages to capture the points at the edges of the space domain but it has trouble depicting the sharp crest in the middle, even though we force at least 1/3rd of the trial points in the domain to be within $x \in [-1,1]$. This suboptimal approximation arises from similar obstacles to the ones we faced when solving the 2D Burger's equation.

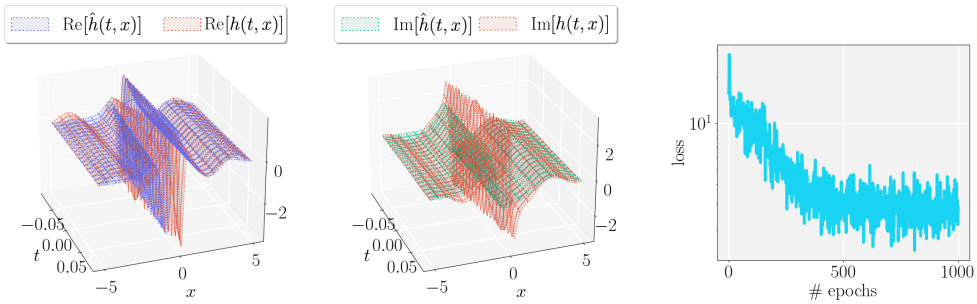

Fig. 11: Solving the derivative nonlinear Schrödinger equation. The NN consisted of four hidden layers with 25 nodes. The result of one run (blue for $\mathrm{Re}[\hat{h}(t,x)]$, green for $\mathrm{Im}[\hat{h}(t,x)]$) is verified against the exact solution (red). The number of training points was 10000, $\beta = \gamma = 10$, and the Adam optimiser had $\lambda_0 = 0.05$, which remained constant eq. (16). The average error was 176% for $\mathrm{Re}[\hat{h}(t,x)]$ and 70% for $\mathrm{Im}[\hat{h}(t,x)]$.

5 Conclusion

In this project, we have embarked on a comprehensive exploration of using neural networks (NN) for the numerical solution of differential equations (DEs).

Taking ordinary differential equations (ODEs) as an entry point, we observed that compact NN architectures are sufficient to approximate a diverse set of problems. In particular, using as examples the 1D Poisson equation and the 1D Helmholtz equation, we showed that adhering to the classical method of setting boundary conditions (BCs) promotes stable convergence, shallow and wide networks are optimal, and increasing the training point density improves accuracy at the expense of computational efficiency. Subsequently, we extended the techniques we developed to solve a system of ODEs, and a non-linear ODE.

Moving into more intricate partial differential equation (PDE) problems, we initially tackled the 2D Poisson equation, achieving notable performance with a modest network architecture, especially when using post-processing techniques to enforce BCs. Our investigation extended to Burger's equation, where satisfactory solutions were obtained in 1D but not 2D due to the computational cost of exhaustive hyperparameter exploration. Finally, we demonstrated attempts to solve the Schrödinger equation, encountering similar obstacles.

Overall, this project illustrates the basic ideas and techniques of using NN for approximating DEs. These are sufficient for ODEs and simple PDEs but more sophisticated optimisation and sampling methods must be used for intricate PDEs.

References

[1] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations". *arXiv preprint arXiv:1711.10561* (2017).

[2] L. Lu et al. "DeepXDE: A deep learning library for solving differential equations". *SIAM review* **63**.1 (2021), pp. 208–228.

[3] A. Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". *Advances in neural information processing systems* **32** (2019).

[4] J. Tanner. *Three ingredients of deep learning: LeNet-5, MNIST, and backprop.* Lecture Notes of C6.5 Theories of Deep Learning. Mathematical Institute, University of Oxford, 2024. URL: https://courses.maths.ox.ac.uk/pluginfile.php/94812/mod_resource/content/4/Lecture%201%20Slides.pdf.

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

[6] R. H. Byrd et al. "A limited memory algorithm for bound constrained optimization". *SIAM Journal on scientific computing* **16**.5 (1995), pp. 1190–1208.

[7] C. Basdevant et al. "Spectral and finite difference solutions of the Burgers equation". *Computers & fluids* **14**.1 (1986), pp. 23–41.

[8] V. Kumar, S. Singh, and M. E. Koksal. "A composite algorithm for numerical solutions of two-dimensional coupled Burgers' equations". *Journal of Mathematics* **2021** (2021), pp. 1–13.

[9] J. Pu, W. Peng, and Y. Chen. "The data-driven localized wave solutions of the derivative nonlinear Schrödinger equation by using improved PINN approach". *Wave Motion* **107** (2021), p. 102823.