Numerical Solutions of Differential Equations Using Neural Networks

Scientific Computing Case Study

Candidate Number: 1061996

1 Introduction

Many techniques exist for solving differential equations. When an analytical solution is not available or too cumbersome to derive we can often use numerical methods to find an approximate solution. Finite difference and finite element methods can be useful in certain contexts but each have their own flaws. For example, the curse of dimensionality when using these methods on higher dimensional domains.

In this essay, we will explore a different method for approximating solutions to differential equations: we will train neural networks to approximate their solutions. This can be done by the normal method of training a neural network: adjusting the weights and biases to minimise an appropriate loss function. We can apply this method to ODEs, PDEs and even systems of ODEs as we will illustrate through examples. We will then go on to investigate how changing different features of the neural network, such as the width and number of layers, effect our solution and we will compare the predictions of our neural networks to that of finite difference schemes. Lastly, we will use this method to solve partial differential equations with 3 independent variables.

2 Introduction to Neural Networks

To start with we will give a basic mathematical introduction of what a neural network is and how they can be trained to perform tasks. In turn, we will define much of the notation we will be using in this essay.

We will be looking at feedforward neural networks. The building blocks of these are called neurons, named as such as they were created to have similar properties as neurons in the brain. Neurons in the brain receive signals and output a signal with size depending on a combination of the signals received. Similarly, neurons in a neural network receive inputs, calculate a linear combination of them, apply a nonlinear function to it then output this as their "signal". For a function receiving n inputs this can be written mathematically as

$$z = b + \sum_{i=1}^{n} w_i x_i \tag{1}$$

$$a = \phi(z) \tag{2}$$

where each $x_i \in \mathbb{R}$ in an input, each $w_i \in \mathbb{R}$ is a weight, $b \in \mathbb{R}$ is the bias term and ϕ is the nonlinear function. We then call z the pre-activation and a the activation of

the neuron. If we let $w = (w_1, \ldots, w_n)^T \in \mathbb{R}^n$ and $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ then z can be written more succinctly as

$$z = b + w^T x. \tag{3}$$

Currently, we just have that each neuron is the composition of a nonlinearity and a linear map but, in a similar way to neurons in the brain, our neurons can be connected such that the output of one neuron is an input of other neurons. We call this composition of neurons a neural network and we can see a diagram of one in figure 1. Writing the function given by a neural network for a general distribution of neurons can look quite daunting so we will first look at a simple example then generalise it.

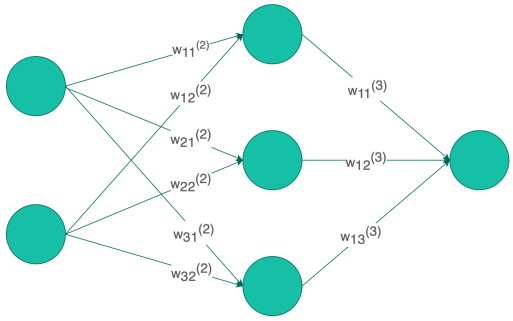


Figure 1: A diagram representing a feedforward neural network.

Suppose we have the neural network shown in figure 1. We say this network has 3 layers with 2 neurons in the first (or input) layer, 3 neurons in the second layer and 1 neuron in the output layer. As this network is still relatively small we can write out the function defined by the network fairly easily. Note that since the input vector of each neuron is left multiplied by the transpose of the weight vector of the respective neuron (as shown in equation (3)) and this happens for every neuron in the second layer and above we can represent the weights at each layer as a matrix to simplify our notation.

Let the inputs to our network be defined by $x = (x_1, x_2)^T$. We denote the weight input to a neuron in the lth row as $w_{ij}^{(l)}$ if the weight is connecting the ith neuron of the lth layer to the jth neuron of the previous layer. Then define the (i, j)th entry of the matrix $W^{(l)}$ to be $w_{ij}^{(l)}$. As we saw in equation (3) we just need to calculate the dot product of our inputs with the weights hence we can write the function defined

by the neural network as

$$\mathbf{a}^{(1)} = \mathbf{x} \tag{4}$$
$$\mathbf{z}^{(2)} = W^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \tag{5}$$
$$\mathbf{a}^{(2)} = \phi(\mathbf{z}^{(2)}) \tag{6}$$
$$\mathbf{z}^{(3)} = W^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)} \tag{7}$$
$$\hat{\mathbf{y}} = \mathbf{a}^{(3)} = \phi(\mathbf{z}^{(3)}). \tag{8}$$
$$\tag{9}$$

with $\mathbf{b}^{(2)} \in \mathbb{R}^3$ and $\mathbf{b}^{(3)} \in \mathbb{R}$ being vectors of biases for the 2nd and 3rd layers respectively and ϕ being the componentwise application of the nonlinearity. We can therefore represent an $L+1$ layer neural network as a composition of L functions $h = f_L \circ f_{L-1} \circ \cdots \circ f_1$ where each f_i is the componentwise composition of an activation function with a linear map.

There are numerous possible choices for the nonlinear function each with their own benefits. One of the most common, and the one we will use in this essay, is the sigmoid function $\sigma : \mathbb{R} \to \mathbb{R}$ where $\sigma(z) = 1/(1+e^{-z})$. This is useful as it represents a continuously differentiable approximation of the identity function. We will see the use in it being continuously differentiable when we discuss the training of the network.

The training of the network refers to finding the optimal values of the weights and the biases such that the network does the task we want it to. This is an optimisation problem as we try to minimise an objective function which measures the mismatch between the prediction of the network and the actual solution. The lower the objective function, the better our network should be at performing the task. This objective function is just used to represent how well our network performs the task and there is no optimal choice of it. One of the most common choices is the mean squared error. Suppose we have N inputs to a neural network $\mathbf{x}^i$ with $i \in \{1, \ldots, N\}$ and labels which give the actual solution and at these points are given by $\mathbf{y}^i$ with $i \in \{1, \ldots, N\}$. The mean squared loss over these points is given by

$$\frac{1}{N}\sum_{i=1}^{N}(h(\mathbf{x}^i) - \mathbf{y}^i). \tag{10}$$

Now that we have a loss function we can learn how to train our network.

To train our network we need data such that our network can "learn" how to do the task we want it to do. That is, we need inputs to the network and the outputs corresponding to when the task has been done correctly. For example, if we wanted

our network to identify objects in images we would want images as inputs and a corresponding lists of objects in each image as outputs. We can then split this data into training and test data keeping the input-output pairs together. This allows us to train the network on the training data then test it on the "unseen" test data so that we can get an accurate prediction of how well the network will perform this task when working with data it has not been trained on.

In terms of how the training is actually done, as we mentioned earlier the training of the network just corresponds to an optimisation problem: we need to minimise the loss function over the training data. As such, we can use optimisation algorithms such as gradient descent and stochastic gradient descent which involve finding the derivative of the loss function with respect to the weights and biases when it has been applied to some collection of training points. We then use this information to change the weights and biases such that the loss function when applied to these points decreases. We hope this decrease in the loss function when applied these training points then translates to a decrease in the loss function when applied to other inputs. Next, we find another combination of training points and do the same again until we have cycled through all of our training points a prescribed number of times. We call a cycle of all training points through our training algorithm an epoch. Hence, after a prescribed number of epochs we say we have finished training. We can then apply the loss function to our test points to quantify how well our model performs the task.

Here, we only mentioned gradient descent and stochastic gradient descent but other more complex optimisation algorithms are available too, such as Adam. These tend to work on the a similar principle, finding derivatives of the loss function and taking a step in a direction which we hope decreases the loss function, however some have added layers of complexity. For example, Adam takes into account gradients at the previous step as well as the current step which can increase the speed at which the algorithm converges. Nonetheless, all the methods we will use require us to differentiate the loss with respect to the weights and biases and use it for minimisation. Using the chain rule, we find that we can calculate this derivative as a product of other derivatives, including the first order derivative of our activation function [1]. Hence, to explain what we said earlier, we need a continuously differentiable activation function so that we have a continuously differentiable loss function and can use these optimisation algorithms on it.

Now we know what neural networks are and how we can train them it is time to uncover how we can use neural networks to approximate solutions to differential

equations.

3 Solving Differential Equations Using Neural Networks

Let us start by considering a general differential equation of order k given by

$$G(\mathbf{x}, g(\mathbf{x}), \nabla g(\mathbf{x}), \ldots, \nabla^k g(\mathbf{x})) = 0, \quad \forall \mathbf{x} \in D \tag{11}$$

subject to certain boundary conditions, with $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, where $D \subset \mathbb{R}^n$ denotes the domain of definition and where $\nabla^j g(\mathbf{x})$ is a vector containing all partial derivatives of g of order j for all $j \in \mathbb{N}$ [1]. We will denote the boundary of the domain D by S and the boundary conditions by $B(\mathbf{x}, g(\mathbf{x}), \nabla g(\mathbf{x}), \ldots, \nabla^k g(\mathbf{x})) = 0, \quad \forall \mathbf{x} \in S$.

Since we are training a neural network to approximate a solution to this differential equation we have to think of a function to minimise such that minimising this function will train our network to give more accurate approximations of the solution to the differential equation. Not only this, but we also need to find training points which we know the exact solution to something for. Looking at equation (11) we see that for any point $\mathbf{x} \in D$ we know what the differential equation G should be equal to. Hence this suggests that we minimise

$$\sum_{\mathbf{x}_i \in \hat{D}} \left(G(\mathbf{x}_i, \hat{g}(\mathbf{x}_i, \mathbf{p}), \nabla \hat{g}(\mathbf{x}_i, \mathbf{p}), \ldots, \nabla^k \hat{g}(\mathbf{x}_i, \mathbf{p}))\right)^2 \tag{12}$$

where $\hat{D}$ is a finite collection of points in D and $\hat{g}(\mathbf{x}_i, \mathbf{p})$ is our trial solution at $\mathbf{x}_i$ with the parameters of the network given by $\mathbf{p}$ [1]. This gives us an idea of what we could use as a loss function but we also need the boundary conditions of our differential equation to be satisfied. There are two approaches to deal with this problem. One involves writing our trial solution in such a way that the boundary conditions are automatically satisfied and making it a function of the neural network. The other uses the neural network directly as the trial solution. First, we will discuss using the neural network directly as the trial solution, then we will consider the other method as it is a small extension to this.

In order to train the neural network to satisfy the boundary conditions in this later case we add an extra term to the loss function to encourage the boundary conditions to become satisfied. This can be expressed in a similar way to the previous term we said would be contained in the loss but with a constant scaling it so we can force the

term to be be more smaller or larger. Thus, the loss function for this method is given on the collection of points $\hat{D} \cup \hat{S}$ by

$$\begin{aligned}\mathcal{L}(\hat{D} \cup \hat{S}) \coloneqq & \sum_{\mathbf{x}_i \in \hat{D}} (G(\mathbf{x}_i, \hat{g}(\mathbf{x}_i, \mathbf{p}), \nabla \hat{g}(\mathbf{x}_i, \mathbf{p}), \ldots, \nabla^k \hat{g}(\mathbf{x}_i, \mathbf{p})))^2 \\ & + \gamma \sum_{\mathbf{x}_j \in \hat{S}} (B(\mathbf{x}_j, \hat{g}(\mathbf{x}_j, \mathbf{p}), \nabla \hat{g}(\mathbf{x}_j, \mathbf{p}), \ldots, \nabla^k \hat{g}(\mathbf{x}_j, \mathbf{p})))^2\end{aligned} \tag{13}$$

where $\hat{D}$ and $\hat{S}$ are a finite collections of points in D and S, respectively. Note that this is not forcing the boundary conditions to be satisfied in the same way the other method will, this just trains the network such that the boundary conditions should become closer to being satisfied throughout training.

We tend to select the points over which to train the network to be evenly spaced in D and S, however if the solution clearly varies much more in one area of the domain we can add more training points there to give a greater proportion of our computing power to learning the function there.

Now we have a loss function for our problem the question is again how do we train it? As we described in section 2 we need to minimise the loss over a suitable set of points using an optimisation algorithm. This requires not only the output of the neural network but also the derivatives of the output with respect to any of its inputs. We then have to differentiate these terms with respect the parameters of the network to apply our optimisation algorithms. We can follow a similar procedure to the one discussed in section 2 to write these derivatives as multiples of known functions, such as the derivatives of the activation function.

Fortunately for us, many packages exist which aid the implementation of neural networks and include autodifferentiation functions so that we do not have to write out these derivatives ourselves. Instead, using the package PyTorch, we can simply define our network, define our loss function, select an optimisation algorithm and train the network, but we will discuss this more next section. Let us first revisit the other method we mentioned to find a solution: using trial functions in a certain form such that the boundary conditions are automatically satisfied.

This can be done fairly easily by writing the trial solution as the sum of two terms, one that satisfies the boundary conditions but contains no tunable parameters and another which does not contribute to the boundary conditions but contains parameters which can be varied. We can therefore vary the parameters in the second term to minimise the loss while knowing the boundary conditions are automatically satisfied.

We can write this trial solution as

$$\bar{g}(\mathbf{x}, \mathbf{p}) = A(\mathbf{x}) + F(\mathbf{x}, N(\mathbf{x}, \mathbf{p})) \tag{14}$$

where $N(\mathbf{x}, \mathbf{p})$ is the output of the neural network at $\mathbf{x}$ with parameters given by $\mathbf{p}$. Since we know the boundary conditions are satisfied, the loss function is given on the collection of points $\hat{D}$ by

$$\mathcal{L}(\hat{D}) \coloneqq \sum_{\mathbf{x}_i \in \hat{D}} (G(\mathbf{x}_i, \bar{g}(\mathbf{x}_i, \mathbf{p}), \nabla \bar{g}(\mathbf{x}_i, \mathbf{p}), \ldots, \nabla^k \bar{g}(\mathbf{x}_i, \mathbf{p})))^2 \tag{15}$$

where $\hat{D}$ is a finite collection of points in D [1].

In reading this section, we have gone from understanding how neural networks work to understanding how they can be applied to solve differential equations. Throughout this section we have spoke generally about differential equations not specifying any specific order or whether they have to be ordinary differential equations (ODEs) or partial differential equations (PDEs). This is because, in theory, this method can be used to approximate solutions in any of these cases. In addition, we can also use it on systems of ODEs if we slightly modify the network architecture and loss function. We will see this in the next sections and learn how to construct trial solutions which automatically satisfy the boundary conditions for these equations.

4 Solving Ordinary Differential Equations

In this section, we are going to use neural networks to find solutions to ODEs. In order to gain a thorough understanding of how this method works we will vary the hyperparameters of the network and see how this affects the solutions we get. Furthermore, we want to compare the solutions given by using both forms of the trial solution. For brevity, we will call the method which forces the boundary conditions to be satisfied by giving the trial function in a certain form the modified method and we will call the other method the vanilla method. Then, as a final means to investigate the accuracy of the methods, we will compare our computed solutions with solutions given by finite difference schemes. However, first, let us look at the implementation of our methods using Python.

As mentioned earlier, there exists lots of packages to help with the implementation and training of neural networks. To implement our methods we use PyTorch. In Python all we then have to do is initialise our neural network, define our loss function and create a function to train the network. The package massively simplifies the

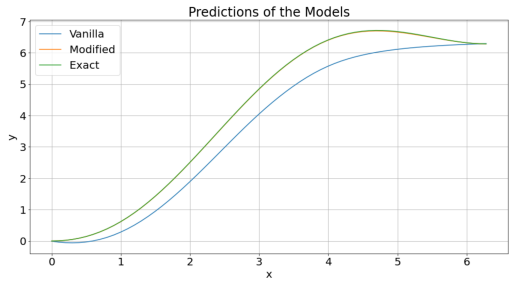
process of training as we can simply select optimisation algorithms, so we do not have to worry about the process that goes into calculating the derivatives. Instead, we can just input the appropriate variables and parameters we need to train the network, such as the set of training points and the number of epochs we want our training algorithm to run for. Similarly, initialising our neural network is simple as PyTorch has easy-to-use functions for this. We simply define a class of neural networks which inherits the functionality from another class `nn.Module`, then we can define layers of the network using the `__init__` function and control how the data will pass through the network using the `forward` function. Most of the difficulties come with defining a loss function. This is because it can be quite tricky to get to grips with the PyTorch autodifferentiation function `torch.autograd.grad`, especially since we are applying it to differentiate the neural network we are optimising. Nonetheless, this can be achieved by having the neural network and set of points we want to differentiate at as inputs to the loss function. Once these components are all working we are ready to test out our new methods.

Now we have discussed the implementation let us look at some applications of our models to a range of different types of differential equation. As mentioned before, there are quite a few parameters that you are required to input to train and use neural networks. Until we say otherwise, we will use a network with 2 hidden layers, each of 128 neurons with sigmoid activation functions and no biases. We train with a batch size of 2 for 100 epochs over a training set of 64 evenly spread points across the domain, including the endpoints and have $\gamma = 1$ in the vanilla model. Furthermore, we use the Adam optimisation algorithm with an initial learning rate of 0.01 and such that it decays by a half after every 10th epoch. After looking at these plots, we will see how varying these parameters affect the accuracy of our solutions and consider why.

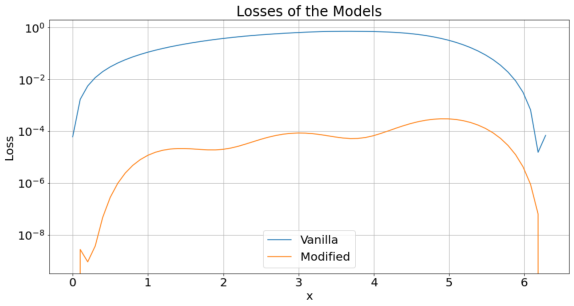
The first differential equation we want to consider is the second order ODE given by

$$\frac{\mathrm{d}^2 g}{\mathrm{d}x^2} = \sin x + \cos x \tag{16}$$

with $g(0) = 0$, $g(2\pi) = 2\pi$ and $x \in [0, 2\pi]$. This has the analytical solution $g(x) = 1 + x - \sin x - \cos x$. To give the form of the trial solution for the modified method $\bar{g}$ we need to write $\bar{g}$ as the sum of two terms, one satisfying the boundary conditions but containing no tunable parameters and another which does not contribute to the boundary conditions but contains parameters which can be varied. This can be done by writing $\bar{g}(x) = x + x(1 - x)N(x, \mathbf{p})$, where $N(x, \mathbf{p})$ is the output of the neural



(a) The prediction of our models and the exact solution against x.



(b) The loss of our models against x.

Figure 2: The predictions and losses of our models for equation (16).

network at $x \in [0, 2\pi]$ with parameters $\mathbf{p}$.

In our figures, we will plot the prediction of each of our models and the exact solution against x so we can see how well our predictions approximate the true solution. We will then also plot the mean squared loss a for each individual point in the domain against x using equation (10) on a single point. This allows us to see the magnitude of our error more clearly.

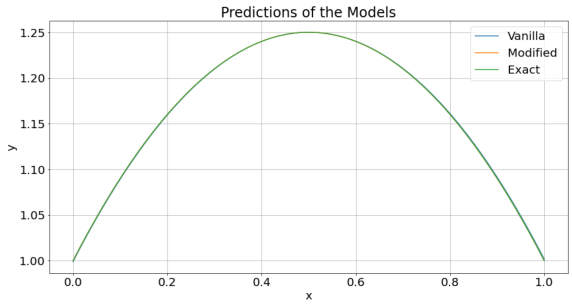
Figure 2 shows our predictions for equation (16). On one hand, we see that the modified model provides an extremely accurate prediction of the solution being practically indistinguishable from the exact solution in figure 2a. We can however, see a small error for most values in figure 2b, but note that the loss is 0 at the boundaries because of the form of out trial solution. On the other hand, the vanilla model provides a good, but not great solution. Both the endpoints look very accurate in figure 2a and the shape follows that of the exact solution reasonably well, but it is far from perfect. Let us see if we find similar things when applying our methods to other differential equations.

We will now consider the nonlinear second order ODE given by

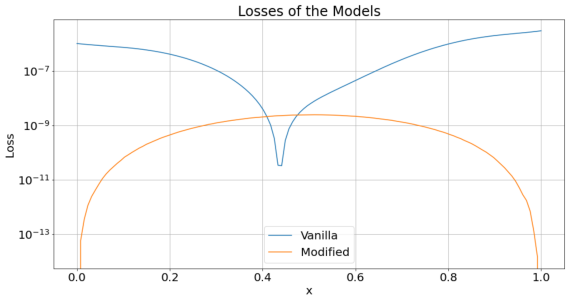
$$-\frac{\mathrm{d}^2 g}{\mathrm{d}x^2} + g^2 = 3 + 2x(1-x) + x^2(1-x)^2 \tag{17}$$

with $g(0) = 1$, $g(1) = 1$ and $x \in [0, 1]$. Since the g is equal to 1 at both of the boundaries we can write the trial solution for the modified form as $\bar{g}(x) = 1 + x(1-x)N(x, \mathbf{p})$, with N defined analogously to the previous example. The analytic solution is given by $g(x) = 1 + x(1-x)$ so we now have everything we need to calculate and plot our solutions.

Figure 3 shows our predictions for equation (17). We see that both of our models provide accurate predictions, however the vanilla model is again less accurate when we



(a) The prediction of our models and the exact solution against x.



(b) The loss of our models against x.

Figure 3: The predictions and losses of our models for equation (17).

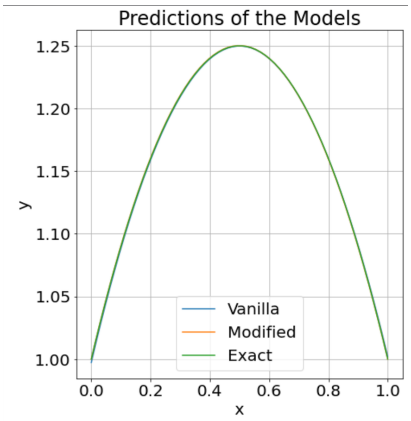
consider the mean squared loss over a grid of 64 equally spaced points on the domain, as shown in figure 3b. Nonetheless, this shows the models' ability to approximate nonlinear ODEs.

Finally, let us see how our models perform on a system of ODEs. More specifically, let

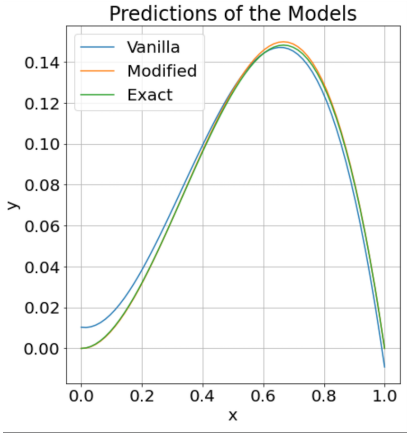
$$\begin{aligned} -\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + xu - v &= 2 + x \\ -\frac{\mathrm{d}^2 v}{\mathrm{d}x^2} + v &= 6x - 2 + x^2(1-x) \end{aligned} \tag{18}$$

with $u(0) = u(1) = 1$, $v(0) = v(1) = 0$ and $x \in [0, 1]$. As the solution of this system is two functions we train a neural network with two outputs. The loss function we minimise during training can then be calculated by summing the losses of u and v if their losses were calculated in the normal way, using equation (13). Since u has the same boundary conditions as the last equation we can use the same form for the modified solution $\bar{u}(x) = 1 + x(1-x)N_1(x, \mathbf{p})$ and for v we analogously have $\bar{v}(x) = x(1-x)N_2(x, \mathbf{p})$, where N_1 and N_2 represent the first and second outputs of our neural network. The last thing we need to make our plots is the exact solution given by $u(x) = 1 + x(1-x)$ and $v(x) = x^2(1-x)$.

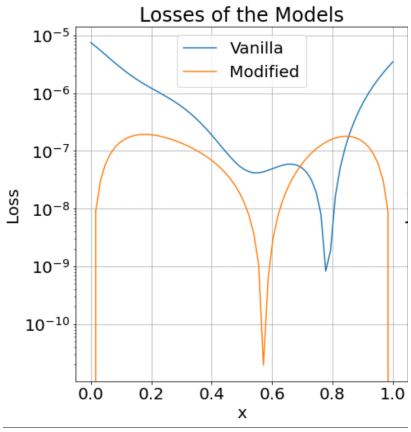
Figure 4 shows our predictions for equation (18). Again, we have the modified method outperforming the vanilla method when you consider the mean squared loss over a grid of 64 equally spaced points on the domain. It seems the greatest loss from the vanilla model when predicting v comes from predicting v at the endpoints of the domain. We have the parameter γ which controls how much the error at the boundary conditions affect the loss function hence increasing γ from 1 to 10 may increase the accuracy of our prediction. Figure 5 shows the result and clearly we have a more accurate approximation of the solution of v.



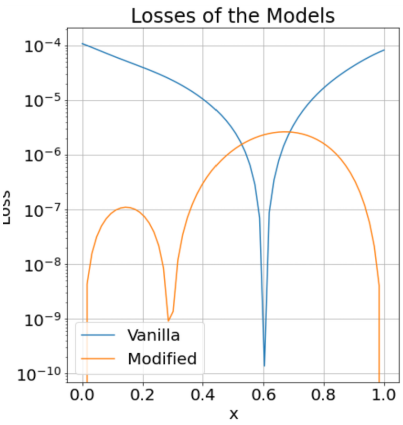
(a) The prediction of our models and the exact solution for u against x.



(b) The prediction of our models and the exact solution for v against x.



(c) The loss of our models for u against x.

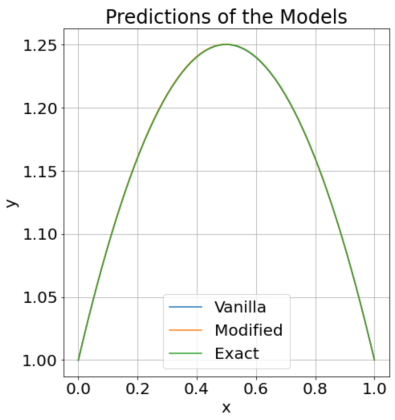


(d) The loss of our models for v against x.

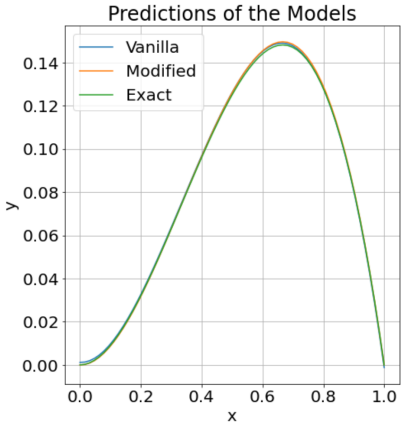
Figure 4: The predictions and losses of our models for equation (18) with $\gamma = 1$.

We have seen how changing the parameters of our network can increase the accuracy of our predictions. This opens the door to the problem of hyperparameter tuning: trying to select the optimal hyperparameter values to enable our network to learn. The hyperparameters are parameters used to control the learning process, such as the architecture of the network and the number of grid points used for training. Now, we will investigate how varying certain hyperparameters affects the approximations of solutions we get. To do this, we will use equation (16) and fix all the parameters at the values given at the start of section 4, aside from the one we are varying. To compare our solutions we will calculate the mean squared loss over 64 equally spaced points on our domain (including the boundaries) by using equation (10).

First, we will look at varying γ. Figure 6 shows the result, but it is not as



(a) The prediction of our models and the exact solution for u against x.



(b) The prediction of our models and the exact solution for v against x.

Figure 5: The predictions of our models for equation (18) with $\gamma = 10$.

we expected. In the previous example when we increased γ the mean squared loss seemed to decrease however here it tends to increase with γ. This may be because the solution was already relatively well approximated at the boundaries for lower values of γ as shown in figure 2. Hence, increasing γ will have caused the boundary terms to become greater in the training loss function and therefore the optimisation will have been more focused on the boundary values despite the function being more poorly approximated on the interior of the domain. These past two examples show how there is no perfect set of parameters for all problems therefore making this method unreliable when you do not know the exact solution you are looking for.

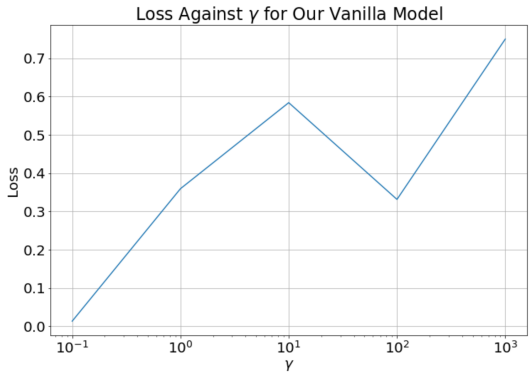


Figure 6: The loss of our vanilla model against γ for equation (16).

Second, we will investigate how varying the number of training points affects the loss of our computed solutions. The plots we get are shown in figure 7 where both

plots are given by running identical code. On one hand, we get what we expected here since, in general, as the number of training points increases the loss decreases for both of our models. On the other hand, we see quite a bit of variability in terms of which model gives the lowest loss and also in terms of the loss of individual models as loss sometimes increases as the number of training points increases. This is a result of randomness in the initialisation of neural networks and in the training, hence when we are looking at plots given by our models it is always important to remember randomness plays a role.

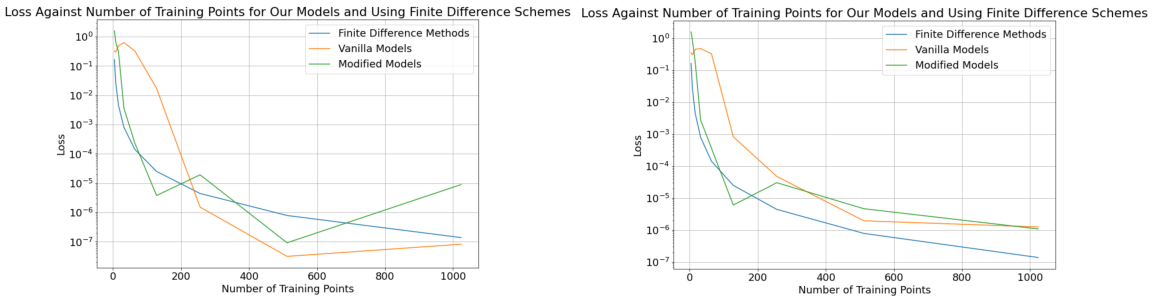


Figure 7: Two plots given by identical code showing the losses of our models and finite difference schemes against the number of training points for equation (16).

The final parameters we will vary will be the number of hidden layers and the widths of the hidden layers of our neural network. We will vary each of these things individually but discuss them together as they are inherently linked. We found in creating these plots that there was quite large variation in what the plot showed each time we ran the code. In general, we saw that as the number of layers and width increased the loss decreased initially and then eventually started to increase again as shown in figure 8. This likely happens as increasing size of the network gives us more weights and biases to optimise, hence initially we find that we can fit our network better to more complex functions. However, as the size continues to increase we find that we have too many parameters to optimise effectively so the loss starts to increase again.

In this section, we have varied each of these parameters individually and compared the result. In practice, we would vary all the parameters and try to find some sort of balance for which the network trains best.

The last thing we have left to do in this section is compare our approximate solutions to the approximate solutions given by finite difference schemes. It is difficult to decide what a fair comparison is between the methods since if we use a finite

Figure 8: The losses of our models against the width of the hidden layers (left) and against the number of hidden layers (right) for equation (16).

difference scheme on the same grid of points we use to train the network it will run far quicker. This is because to find a solution to the finite difference scheme you simply have to solve a linear system of equations, whereas training a neural network requires us to run time-consuming optimisation algorithms. Nonetheless, using all methods on the same grid seems like the natural way to compare them.

We will again use equation (16) to compare the methods. For the finite difference scheme, we will simply use the second divided difference operator to approximate the second order derivative and then fix the solution at the boundaries. We choose to vary the number of grid points and plot the loss of our computed solution against the number of grid points as shown in figure 7. We see that the solutions of the finite difference schemes have similar losses to both of the models, but more so to the modified model. However, if we have problems which require lots of computation finite difference methods are more practical as they require much less processing to compute a solution with similar mean squared loss.

At the start of section, we discussed how neural networks could be implemented to find numerical solutions of ODEs then proceeded to use neural networks to find these solutions for a variety of types of ODE. A natural question would then be to ask does the implementation or the accuracy of the solutions change when considering PDEs? This is what we will discuss in the next section.

5 Solving Partial Differential Equations

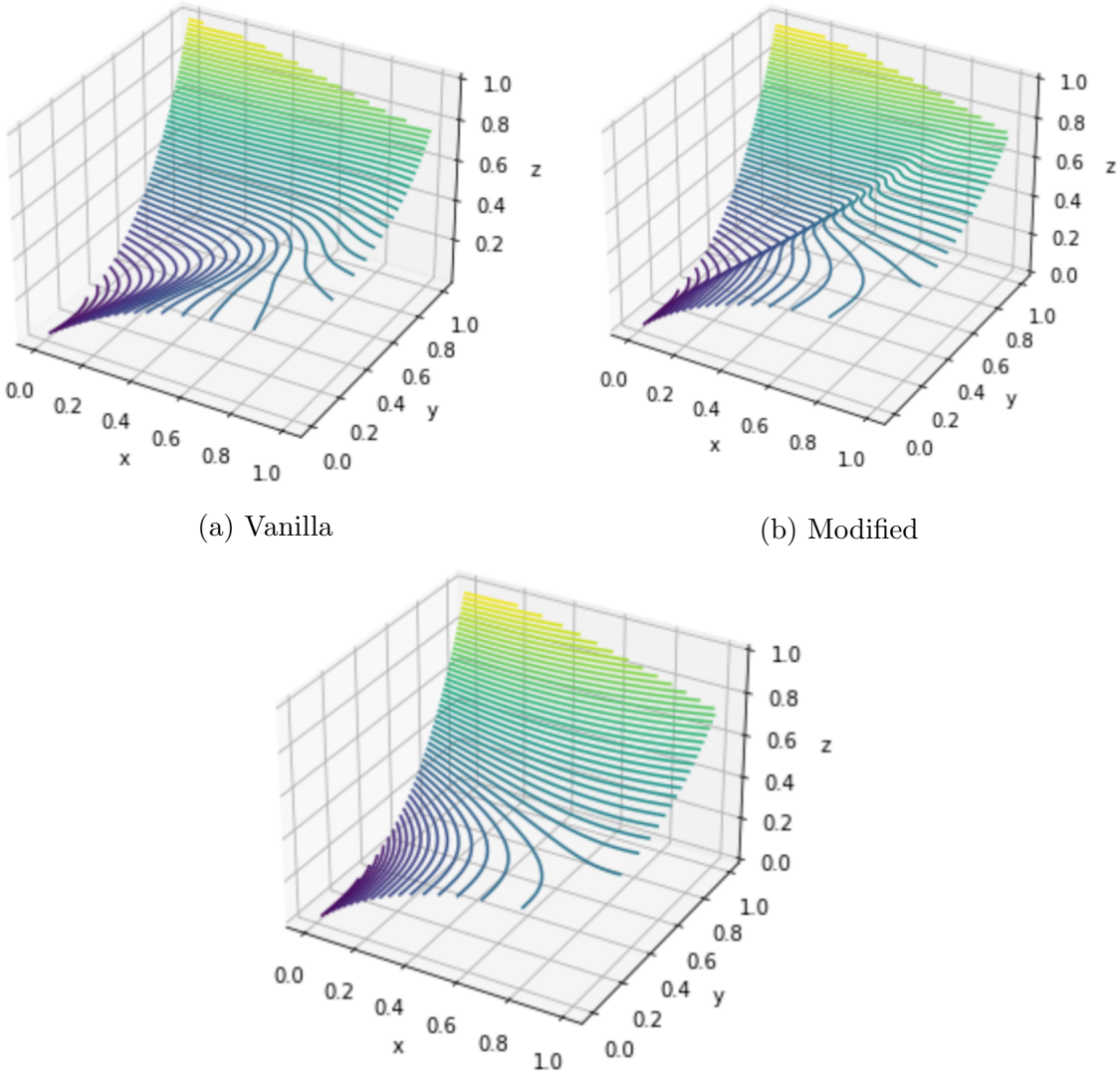
We first want to investigate the implementation of neural networks to solve PDEs and how it differs from the implementation for ODEs. The first thing to note is we now need 2 inputs to our network rather than 1 but this is simply a matter of changing a parameter. A less trivial difference is that we can no longer give our training points as a line of equally spaced points. Instead, we will use a higher dimensional equivalent, such as a grid of points if we have 2 independent variables. This sounds quite simple to implement however it can cause difficulties in the training of our network. This is because we want to shuffle the order of the points before we train the network so we do not just train the network on points from one side of the domain to the other as by the end of training the network will likely not be accurate for points in the first part of the domain anymore. Suppose we have two independent variables. Shuffling a two dimensional grid of points can be difficult, however if we write all m training points in an $m \times 2$ dimensional array we can then permute the columns using the `torch.randperm` function and use this permuted vector as the order of points to train the neural network with. Another problem is in implementing the loss function used during training. We need the correct boundary conditions to be applied on our training points but the boundaries are no longer points, they are higher dimensional objects. Hence we use a function to determine which boundary the point is on so we can subtract the correct value from it in the loss function. The final, and most obvious, difference is that we now need to be able to partially differentiate. This does not end up being a big problem as we simply change an input in the `torch.autograd.grad` function from batches of points in 1 dimension to batches of points in the higher dimension.

Now we have seen the difference in the implementation, let us see if the models can approximate solutions for PDEs. For these next plots we use the same parameters as those given in section 4, aside from setting the interior batch size to 16 and boundary batch size to 4 in the vanilla model and setting the batch size to 16 in the modified model. We will apply our method to the PDE given by

$$\nabla^2 g(x, y) = e^{-x}(x - 2 + y^3 + 6y) \tag{19}$$

with $x, y \in [0, 1]$ and boundary conditions $g(0, y) = y^3$, $g(1, y) = (1+y)^3 e^{-1}$, $g(x, 0) = xe^{-x}$ and $g(x, 1) = e^{-x}(x + 1)$. In the same way as we do for ODEs, we can write the form of the trial function for the modified solution as the sum of two term, one which satisfies the boundary conditions and another that does not affect them. We

obtain this by subbing the general form of solution for a PDE on a unit square $g(x, y) = A(x, y) + x(1 - x)y(1 - y)N(x, y, \mathbf{p})$ into equation (19) and rearranging for A [1]. The exact form is not important for the purposes of this essay but we will use it in our code for the modified method. Finally, the analytic solution to this problem is given by $g(x, y) = e^{-x}(x + y^3)$; we can now make our plots.



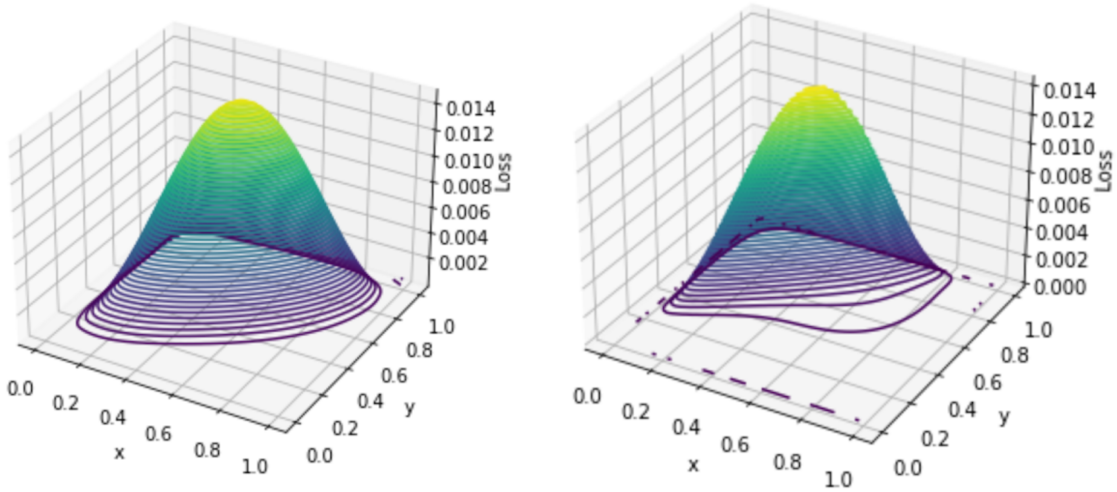
(a) Vanilla

(b) Modified

(c) Exact

Figure 9: The predictions of our models and the exact solution for equation (19) against x and y displayed as contour plots.

Looking at figures 9 and 10 we see that both methods look like they provide good approximations of the actual solution. It seems most of the loss is a result of poorer approximations in the centre of the domain. This would be expected for the modified



(a) Vanilla

(b) Modified

Figure 10: The loss of our models for equation (19) against x and y displayed as contour plots.

method, however for the vanilla method this shows that we have a good γ value as the boundary conditions are well approximated. If we decrease the value of γ we find the boundaries are no longer as well approximated and the loss of the whole function suffers as a result.

It is difficult to tell which model approximates the function better but we can take a grid of 4096 equally spaced points over the domain $[0, 1]^2$ and calculate the mean squared loss using equation (10). We find that when using the vanilla method we get a loss of 0.00309 and when using the modified method we get a loss of 0.00165. Hence, if we used the mean squared loss as a metric to decide which model approximated the solution better we would find that it was the modified method, as was the case with all our ODE examples.

In this section, we spoke generally at the start about using our models to find solutions to PDEs but we did not go into too much detail, especially for PDEs with more than 2 independent variables. Furthermore, we only gave an example of a PDE with 2 independent variables. In the next section, we will try to solve PDEs with 3 independent variables using our models.

6 Extension

In this section, we want to extend our code so we can use our models to compute solutions to PDEs with 3 independent variables. As usual, we will start by discussing how we would implement our model using code. The first thing we do is change our neural network to have 3 inputs. As mentioned in the previous section, one of the biggest problems is trying to create a grid of points to train the network on then getting these into a list such that we can permute the order of them so we are not training over the domain in any specific order. This can be done using the `torch.linspace`, `torch.meshgrid`, `torch.flatten`, `torch.reshape` and `torch.cat` functions. Due to the nature of our loss function for the vanilla method, training points in the interior and on the boundary need to be treated differently when training the vanilla method. As such, we divide our training points into boundary and interior points and have a different batch size for each of them so that when training we can apply the appropriate function to each of them in the training loss. We just need to set the batch sizes correctly so that during training we run out of training points in both sets at similar times. Once we have done this we need to define a function such that given a boundary point it finds which boundary the point lies on and gives an appropriate boundary condition. This can then be subtracted from our the vanilla models prediction in the loss function. The last thing we need to do is find how to differentiate with respect to our new variable. This has not changed from the previous step aside from now an input into the `torch.autograd.grad` function is batch size by 3 dimensional rather than batch size by 2 dimensional as it was for PDEs with 2 independent variables. The function to calculate the gradients of the approximate solution at the batch size by 3 dimensional input `batch_all` given its batch size by 1 dimensional approximate solution `output` is `grad = torch.autograd.grad(output, batch_all, grad_outputs=torch.ones_like(output), retain_graph=True, create_graph=True)[0]`. We can then select the derivative with respect to any of the three independent variables using `grad[:,0]`, `grad[:,1]` and `grad[:,2]`. The function to calculate the second derivatives at `batch_all` is given in a similar way by `hess =torch.autograd.grad(grad, batch_all, grad_outputs=torch.ones_like(grad), retain_graph=True, create_graph=True)[0]` and we can select the second order derivative with respect to any of the three independent variable using `hess[:,0]`, `hess[:,1]` and `hess[:,2]`. These can then be used in our loss function as we will show now by doing some examples.

We will use the parameters as described in section 4 but we will change the hidden

layer width to 256, the number of training points to 4096 and instead of just having one batch size we will have an interior batch size of 16 and an boundary batch size of 4 for the vanilla model. We will have just one batch size of 16 for the modified model. We will create a grid of 4096 equally spaced out points in the unit cube to be our training set.

The first equation we want to solve is Laplace's equations with 3 independent variables. More specifically, we want to solve

$$\Delta g(x, y, z) = 0 \tag{20}$$

with $x, y, z \in [0, 1]$ and homogeneous Dirichlet boundary conditions. This has the unique solution $g = 0$ and we can write the trial solution for the modified model as $\bar{g}(x, y, z) = x(1-x)y(1-y)z(1-z)N(x, y, z, \mathbf{p})$.

Since we have homogeneous Dirichlet boundary conditions, the loss function for our vanilla model is given by `loss = (((hess[:, 0] + hess[:, 1] + hess[:, 2] )**2).mean() + (gamma*((y_predict_boundary)**2)).mean())`, where `y_predict_boundary` is the prediction found by subbing our training batch into the network and `hess` is defined as above. The loss for the modified method is the same just without the final term. Running our code and calculating the mean squared loss over our set of grid points gives 3.33×10^{-9} and 3.047×10^{-5} for the vanilla and modified methods, respectively. Thus we have found accurate solutions using our model.

We now want to solve Poisson's equation given by

$$\Delta g(x, y, z) = -2x(1-x)y(1-y) - 2y(1-y)z(1-z) - 2z(1-z)x(1-x) \tag{21}$$

with $x, y, z \in [0, 1]$ and homogeneous Dirichlet boundary conditions. This has the unique solution $g(x, y, z) = x(1-x)y(1-y)z(1-z)$ and the modified model's solution has the same form as the previous equation's modified solution.

We can give the training loss for the vanilla method as `loss = ((hess[:, 0] + hess[:, 1] + hess[:, 2] -(-2*batch_interior[:,1]*(1-batch_interior[:, 1])* batch_interior[:,2]* (1-batch_interior[:, 2]) -2*batch_interior[:, 0]* (1-batch_interior[:, 0]) *batch_interior[:,2]*(1-batch_interior[:,2]) -2*batch_interior[:,0]*(1-batch_interior[:, 0])*batch_interior[:, 1]* (1-batch_interior[:, 1])))**2).mean() + (gamma*((y_predict_boundary - y_target_boundary)**2)).mean()` and similarly for the modified method. Training our networks then calculating the mean squared losses in the same way as the previous example we find that the vanilla model has a loss of 1.73×10^{-5} whereas the

modified model has a loss of 9.93×10^{-6}. These losses sound good but if we calculate the average value of g on our grid we find it is 0.0038 and if we calculate the average of the magnitudes of our prediction minus the true solution over our grid points we get 0.0027 and 0.0017 for the vanilla and modified method, respectively. Nonetheless, if we look at the exact solution for a range of points and our models over the same range we can often see the models following a similar pattern just without varying as much. Furthermore, if we increase the number of training points we find that our loss decreases. If we use 8 times more training points we find that the vanilla model now has a mean squared loss of 1.43×10^{-5} and the modified model has a mean squared loss of 6.22×10^{-6}. Hence, it looks as though if we continue to increase the number of training points the models may converge to a solution.

Our problem is now that we cannot increase the number of training points any more without the program on my laptop crashing: it seems as though running the code takes up too much memory. Hence, it seems that the neural network would be able to get an accurate approximation the solution of the PDE however it would require training with more training points. This is likely because as the number of independent variables increases, the solution can vary in more ways and therefore the network becomes more difficult to train.

7 Summary

In this essay, we started by introducing neural networks. We described them then gave a brief description of how they can be trained such that they perform desired tasks before we adapted this description to see how they could be trained to solve differential equations. We then implemented two different methods to solve differential equations using neural networks and solved a variety of ODEs. Once we were able to solve a few ODEs we spent some time investigating how varying parameters affects the computed solution and also compared our models' computed solutions with a finite difference scheme's. We then went on to use our models to compute solutions of a PDE with 2 independent variables before going on to our extension which was computing solutions of PDEs with 3 independent variables. We succeeded in getting an accurate solution for Laplace's equation but we found that our solution to Poisson's equation was not great, however we suspect it is because we need more computing power. This is because in general, the more independent variables a PDE has, the more variable it can be and therefore the more difficult it is to train.

References

[1] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *Institute of Electrical and Electronics Engineers (IEEE)*, 1:1–25, May 1997. URL `\url{https://arxiv.org/abs/physics/9705023}`.