Department of Mathematical Sciences

Master's Dissertation

Neural Networks for Solving Differential Equations

Author : *Isaac S. Hayden*

Supervisor : *Kasper Peeters*

Declaration

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

Acknowledgements

This work has used Durham University's NCC cluster. NCC has been purchased through Durham University's strategic investment funds, and is installed and maintained by the Department of Computer Science. I would like to thank their administration team for the use of the NCC cluster, and for their technical support.

I am indebted to my supervisor, Kasper Peeters, for providing me with the opportunity to explore this topic, and for his guidance within it.

I am also grateful to my academic advisor, Ian Jermyn, for his consistent support in every area of my undergraduate career from beginning to end.

I would like to thank Daniel Disney for his proofreading, infectious love of mathematics, and for never tiring of hearing about neural networks.

Without the continual sacrifices of my parents, I would not be where I am today. There is no end to what I owe them, and I will always be grateful for this.

Finally, thank you to Beth Presswood for more things than I could list. Every day I spend with you is better than the last.

Abstract

Finding numerical solutions to differential equations is crucial to many scientific disciplines. In the 1990s, a new method was proposed which utilises neural networks to approximate the solution function of a differential equation. In this dissertation, we explore the efficacy of this method in a variety of different examples. We also examine the effect of varying different aspects of our neural network's structure and training methodology. Finally, we consider a recent extension of the original method, and another technique by which neural networks can be used to estimate unknown parameters in a differential equation.

Contents

1 Introduction

The field of machine learning has grown exponentially since its advent in the mid-twentieth century. At its core, it seeks to find methods by which computers can learn from their experiences to perform various tasks; not just those which humans are capable of (e.g. image recognition), but also those which we find difficult (e.g. discerning patterns in large data sets) [1]. The methods by which such a goal is achieved have evolved in many different directions in the last 70 years. Now, these various manifestations of artificial intelligence are not only ubiquitous in the modern world, but are actually fundamental to many of the technological systems we rely on in everyday life.

The aim of teaching machines to 'think' naturally led early developers to take inspiration from the human brain. This produced a computing system known as an **artificial neural network**: a function whose structure mimics that of neurons and synapses in the brain, and whose parameters can be modified algorithmically based on the performance of the function's output [2]. Although these now exist in many forms, they remain at the cornerstone of machine learning, and will be the centre of our focus in this paper. In particular, we will exploit a result first proved in the 1980s known as the Universal Approximation Theorem, which states that neural networks can approximate virtually any function to arbitrary accuracy [3].

Meanwhile, the study of differential equations dates back to the seventeenth century with the invention of calculus by Newton and Leibniz. Being equations that measure the nature of change, they are crucial to understanding the relationships within any physical system. This has meant that solutions to differential equations are highly sought, and methods of finding solutions have been explored extensively since their discovery [4].

More often than not, however, it is the case that analytic solutions are impossible or impractical to obtain. In these scenarios, **numerical methods** are employed instead, which typically seek to approximate the value of the solution function at given points, rather than obtain a closed-form expression for the solution. There now exist many varieties of this idea, with the first of these methods being attributed to Euler [5].

It was then in the 1990s that the two fields of machine learning and differential equations were connected by a technique known as the **Lagaris method** [6]. This leverages the Universal Approximation Theorem to create a neural network which can be used to approximate the solution of a differential equation. Most interestingly, this differs from usual numerical methods by providing a closed-form expression of a function which serves as an approximate solution to the equation. The theorem then tells us not only that such an approximation is always possible, but that it can be made as accurate as we wish.

In this paper we will explore the capabilities of the Lagaris method. First, we will introduce the most fundamental concepts of neural networks: how they are structured and how they can be 'trained'. We will then see how they can be used for function approximation in a recent Python library designed for machine learning, known as **PyTorch** [7]. After this, we will explore a variety of methods for training a neural network, specifically examining how the algorithms for optimising the network's parameters can vary.

Then, we will see in detail how the Lagaris method can be used to solve differential equations, and how this differs from typical function approximation. We will test the method's capacity to handle several different types of differential equations, and use these examples to investigate the effect of varying parts of our network structure and training methods.

Finally, we will investigate further applications of neural networks to differential equations. Specifically, we will see a recent extension of the Lagaris method which renders it more powerful and computationally effective compared to traditional numerical methods. Beyond the Lagaris method, we will explore how neural networks can be implemented to estimate unknown parameters in a differential equation, by using sample data values of the solution function.

We will then conclude with a discussion of the scope of this paper, and indications for areas of future development which we have not explored.

Samples of the code we refer to throughout are provided in appendix C, while all files are available in full in [8].

2 Neural Networks

We begin by understanding some basic notions in the study of neural networks, with our desired applications in mind. We see the fundamental components that make up a neural network, and a common method of training them. Then, we explore **automatic differentiation**, the computational technique behind a crucial part of their training.

2.1 Structure and Definitions

Neural networks are a programming paradigm inspired by the structure and learning mechanisms of the human brain. They provide a powerful methodology for computers to learn from observed data and make further predictions, and can be trained to perform a wide variety of tasks [2]. Their basic structure is shown in figure 1:

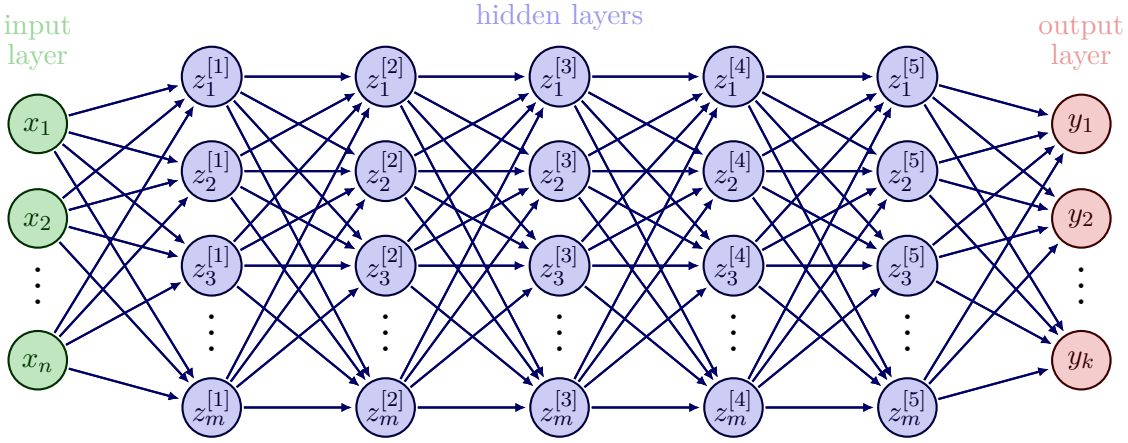

Figure 1: A fully-connected, feedforward neural network with n inputs, k outputs and five hidden layers each made up of m nodes.

A network consists of an **input layer** $x = z^{[0]} = (x_1, \dots, x_n) \in \mathbb{R}^n$, an **output layer** $y = z^{[6]} = (y_1, \dots, y_k) \in \mathbb{R}^k$ and a number of (in this case five) **'hidden' layers** $z^{[i]} = (z_1^{[i]}, \dots, z_m^{[i]}) \in \mathbb{R}^m$, $1 \leq i \leq 5$. An element of any of these column vectors is known as a **node** or **neuron** of the network. The layers are connected to each other by the following equation:

$$z^{[i]} = g_i(W^{[i]} z^{[i-1]} + b^{[i]}), \qquad 1 \leq i \leq 6, \tag{1}$$

where $W^{[i]}$ is a matrix containing the **weights** of the i^{th} layer and $b^{[i]}$ is a column vector containing the **biases** of the i^{th} layer. $g_i : \mathbb{R} \to \mathbb{R}$ is called the **activation function** of the i^{th} layer, which we apply component-wise to each element of the given vector. The purpose of activation functions will be discussed in the next section.

We define the **width** of the network to be the maximum number of nodes in any of its layers, and the network's **depth** to be the total number of layers excluding the input layer. Hence the network in figure 1 has width $\max\{n, m, k\}$ and depth 6.

Naturally, the dimensions of each weight matrix and bias vector are chosen such that equation (1) is well-defined (e.g. $W^{[1]}$, $W^{[3]}$ and $W^{[6]}$ are $m \times n$, $m \times m$ and $k \times m$ matrices respectively). Weights and biases are the main parameters of a neural network; training a network is simply the process of finding optimal values for these parameters so as to give the desired output y.

Thus our neural network is a function $N(x; \theta) : \mathbb{R}^n \to \mathbb{R}^k$, where θ denotes all the network parameters (i.e. the weights and biases).

Note that the network in figure 1 is an example of the most simple type of neural network. It is **feedforward** (there are no cycles present in the network structure) and also **fully connected** (every node in layer i is connected to every node in layer $i-1$, i.e. the value of each node depends on every node in the previous layer). This is the kind of network structure we will use throughout, but many variations exist [2].

2.2 Activation Functions

To motivate the need for activation functions, suppose we did not include them in equation (1) and consider the output of our network.

$$\begin{aligned} z^{[1]} &= W^{[1]}x + b^{[1]}, \\ z^{[2]} &= W^{[2]}z^{[1]} + b^{[2]} \\ &= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} \\ &= \tilde{W}^{[2]}x + \tilde{b}^{[2]}, \end{aligned} \tag{2}$$

where $\tilde{W}^{[2]} = W^{[2]}W^{[1]}$, $\tilde{b}^{[2]} = W^{[2]}b^{[1]} + b^{[2]}$. Thus we can replace our first two hidden layers with a single hidden layer. Repeating this iteratively for all the layers in our network, we obtain a matrix $\tilde{W}$ and a column vector $\tilde{b}$ such that:

$$y = \tilde{W}x + \tilde{b}.$$

Hence our network has simply become a linear transformation. Due to the complexity of the types of problems that neural networks are intended to solve, such a linear transformation is rarely sufficient to approximate a solution well. This creates the need for activation functions: if we choose them to be non-linear, then our network is no longer a simple linear transformation. By adding non-linearity to our neural network, we increase the complexity of the resulting function $N(x;\theta)$, and are thus capable of solving more complicated problems [2].

Some examples of activation functions are shown below. Their derivatives are also shown for later reference. Note it is easy to show that $\sigma'(x) = \sigma(x)(1-\sigma(x))$ (figure 2) and $\tanh'(x) = 1 - \tanh^2(x)$ (figure 3) from their definitions. Although ReLU (figure 4) is not differentiable at zero, by convention we set $\text{ReLU}'(0) = 1$.

Notice that the properties of these functions vary greatly. σ and tanh for example are smooth functions, and 'squash' their inputs into the ranges $[0, 1]$ and $[-1, 1]$ respectively. Meanwhile, ReLU and Linear are unbounded functions, and ReLU is not differentiable at zero. Also, the second derivatives of ReLU and Linear are identically equal to zero. The correct choice of activation function is an important factor, and depends greatly on the given application. We will explore this more in section 3.3.

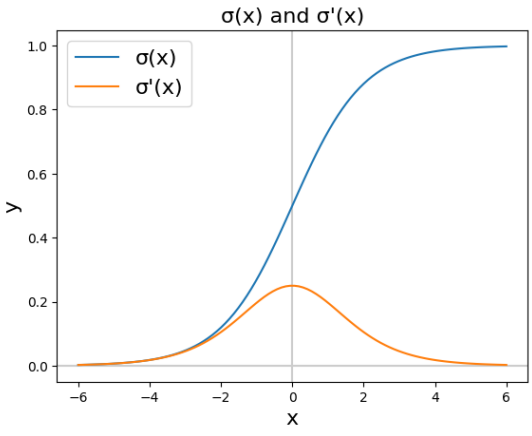

Figure 2: The sigmoid function $\sigma(x) = 1/(1+e^{-x})$ and its derivative $\sigma'(x) = \sigma(x)(1-\sigma(x))$.

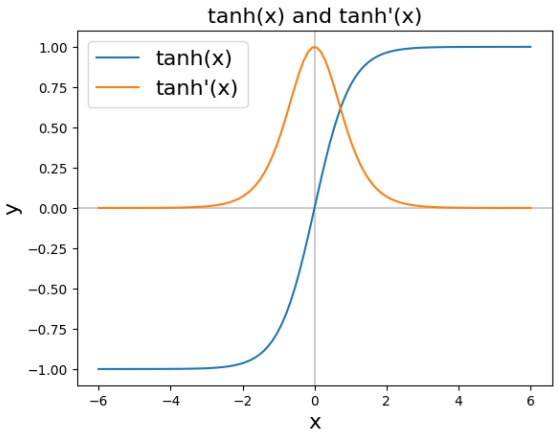

Figure 3: $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and its derivative $\tanh'(x) = 1 - \tanh^2(x)$.

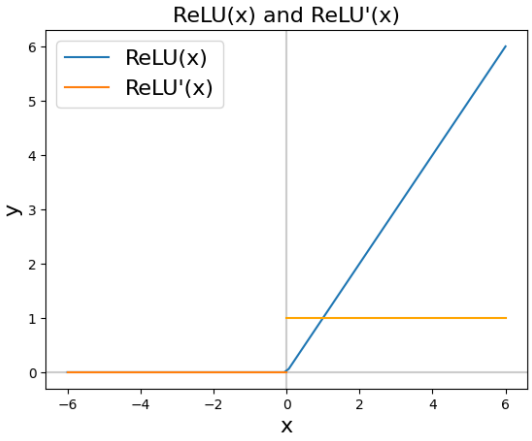

Figure 4: The ReLU function $\mathrm{ReLU}(x) = \max(0, x)$ and its derivative $\mathrm{ReLU}'(x) = \begin{cases} 1, & x \geq 0, \\ 0, & x < 0. \end{cases}$

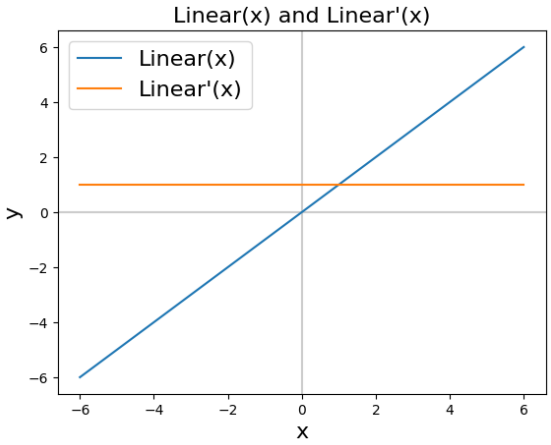

Figure 5: The Linear function $\mathrm{Linear}(x) = x$ and its derivative $\mathrm{Linear}'(x) \equiv 1$.

2.3 Training a Neural Network

There are many ways to train a neural network; here we describe a method [2] in general, while in later sections specify a few more details. We begin by initialising the network's weights and biases (typically with random values, see section 2.4), and choose M samples $\{x_1, \ldots, x_M\} \subset \mathbb{R}^n$ as training data. The training method we use then has four steps.

The first step is known as **forward propagation**. We pass all M training points through our network and store their outputs $\{\hat{y}_1, \ldots, \hat{y}_M\} \subset \mathbb{R}^k$. These values $\hat{y}_j$, $1 \leq j \leq M$, can be thought of as predicted values or **estimators** of the true values y_j we are seeking to approximate.

Next we calculate the **error** or **cost** of our network output. This can be done in several ways. If we know the true value y_j a priori, then we can simply calculate the loss of each $\hat{y}_j$, using a loss function of our choice. Some examples of loss functions are given below [2] (from left to right: absolute loss, square loss, cross-entropy loss):

$$L_{abs}(y, \hat{y}) = |\hat{y} - y|, \quad L_{squ}(y, \hat{y}) = (\hat{y} - y)^2, \quad L_{ent}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})). \tag{3}$$

An example of this scenario would be approximating a function $f(x)$ based on sample data points $\{(x_1, f(x_1)), \ldots, (x_M, f(x_M))\}$, using $y_j = f(x_j)$. We will see this in detail in section 3.3.

If we do not know the true values y_j beforehand, then we require some other metric to measure the loss of each output. When we apply networks to solving differential equations, we will see an explicit example of calculating loss in this scenario.

In either case, once each of these losses have been calculated, we aggregate all of their values into a single value, J, known as the **error** or **cost**:

$$J = \frac{1}{M} \sum_{j=1}^{M} L(y_j, \hat{y}_j). \tag{4}$$

Minimising J will bring our network closer to approximating the correct solution. If we consider J as a function of the network's parameters w, b, then we obtain a high-dimensional surface (figure 6), whose global minimum we seek to find. This will be accomplished in the final two steps of training.

The third step is known as **backpropagation**. We can calculate the partial derivative of J with respect to each of the network parameters. Since J was obtained by the composition of several functions (first in the neural network, and then in the cost function), we can repeatedly apply the chain rule to calculate these derivatives. This is done numerically by using **automatic differentiation** (see section 2.5).

Finally, we carry out **parameter optimisation**. In backpropagation, we calculated the vector ∇J, which tells us the direction in which J is increasing most steeply with respect to w, b. By updating the parameters in direction $-\nabla J$, we come closer to finding a minimum of J. By minimising J, we bring our network's output closer to the true value we wish to approximate. This is known as **gradient descent**.

We update each of our parameters using the following algorithm:

Algorithm 1 (Gradient Descent)

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w}, \qquad b \leftarrow b - \alpha \frac{\partial J}{\partial b}, \tag{5}$$

where $\alpha > 0$ is a fixed positive constant (typically very small, e.g. 10^{-3}) called the **learning rate**, which determines the magnitude of the steps we take when moving across this error landscape $J(w, b)$.

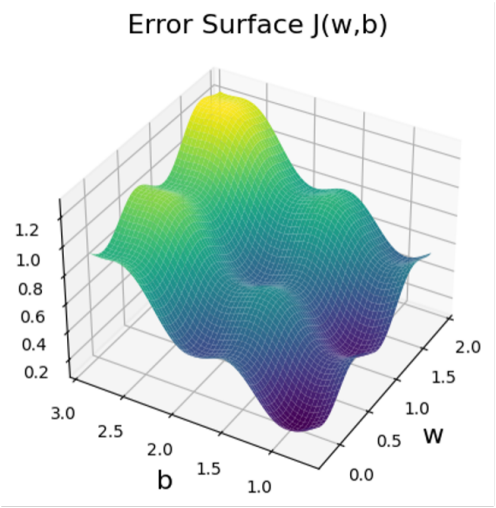

Figure 6: The error of a neural network as a function of the network parameters. We represent this as a surface in three dimensions, but note that the dimension is always much higher.

The choice of learning rate is an important factor in successfully training a neural network, and its optimal value varies greatly depending on the given task. For example, if the learning rate is too small, we could arrive at a *local* minimum (figure 7). In such a local minimum the partial derivatives will be equal to or very close to zero, and hence algorithm 1 will leave our parameters virtually unchanged. Thus we will remain stuck in a suboptimal solution.

Conversely, if α is too large, then we may continue to move over the global minimum, since our steps are too large to descend the slope. By choosing a learning rate of appropriate size, we can converge to the *global* minimum (figure 8). Often, a trial-and-error process is required to find the optimal learning rate. We will explore the effect of learning rate in more detail in section 5.5.2.

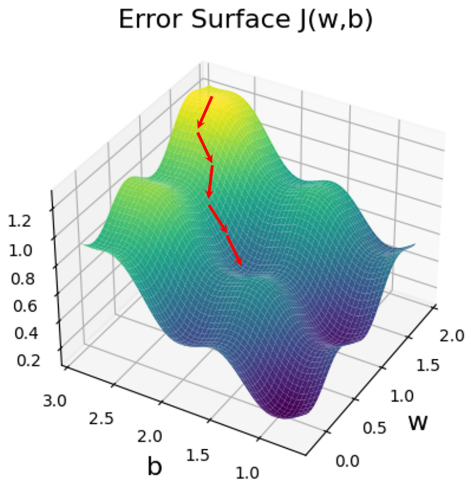

Figure 7: Convergence to a local minimum due to a small learning rate.

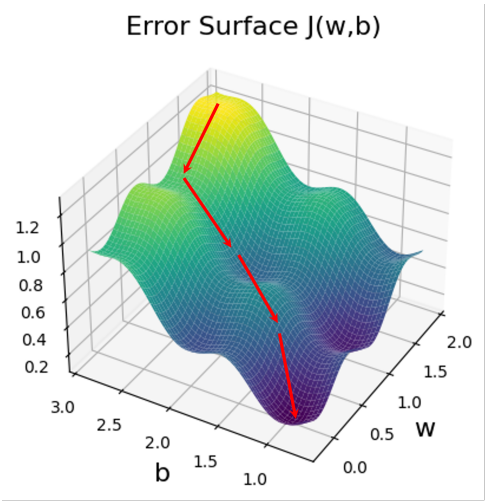

Figure 8: Convergence to the global minimum with a larger learning rate.

Note also that algorithm 1 gives gradient descent in its most naïve form. In section 4 we will see variants which perform better in practice.

Once these four steps have been carried out on all M training data points, this is known as an **epoch**. We can complete as many epochs as necessary for the cost to become sufficiently small.

Having been trained on a sample of size M, the network can now been tested on other input data. While a network is still failing to perform well on training data, this is known as **underfitting**. The network has not yet recognised the fundamental underlying relationships between x and y that we are trying to approximate, and thus the statistical **bias** of our estimator $\hat{y}$ is high. Hence more training is required to minimise J further.

Conversely, if a network performs well on training data, but poorly on test data, this is **overfitting**. It is indicative in a high **variance** in our estimator $\hat{y}$ for y, and can be attributed to the network learning features of our training set that are not generally true of all our input data.

Trying to reduce the error introduced by both high bias and variance, and thus prevent both underfitting and overfitting, is known as the **bias-variance dilemma** [2].

2.4 Weight Initialisation

When constructing any neural network, an unavoidable question is how to initialise the network's weights and biases. This can be an incredibly important factor: referring again to the error surface in figure 6, starting at different points on this surface can make convergence to the global minimum significantly easier or harder. Since this surface in reality has much higher dimension, and since it is difficult to know much about its shape a priori, it is challenging to determine optimal initial values for w and b beforehand. We highlight some potential problems that can come about from poor choice of initial parameter values, and indicate how they are resolved in practice.

2.4.1 Constant Initialisation

One idea might be to take fixed values, w_i and b_i, and initialise every weight and bias in the i^{th} layer to w_i and b_i respectively. This creates redundancy in our network however, since every node in this layer now takes the same inputs (all values from the previous layer) and will produce the same output. Hence each weight in the layer will have the same derivative, and similarly for each bias. They will then be updated identically in the gradient descent algorithm, and hence all parameter values across the layer will always be the same as each other.

The problem with this is that it reduces the complexity of our network. The purpose of having multiple layers (and multiple neurons in each layer) is to allow our network to distinguish multiple features (and subfeatures) of the underlying data set from which our training data is drawn. By not allowing nodes in a given layer to deviate from each other, we hinder the network's ability to solve more complicated problems. From another perspective, it forces our position on the error surface to have identical values on several parameter axes. Naturally, this restricts our ability to find a global minimum.

Hence we can conclude that the parameters of a given layer should be initialised with varying values. Given this, a natural suggestion would be to initialise our parameters with *random* values. A wide variety of random initialisation methods exist [9], dependent on the specific type and application of the neural network.

2.4.2 Exploding and Vanishing Gradients

There is another important consideration to be made when initialising parameters. To illustrate it, consider the network below:

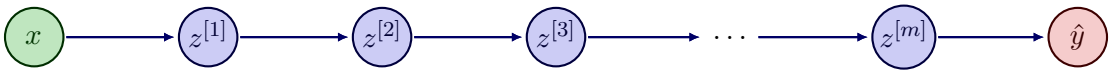

Figure 9: A neural network with 1 input, 1 output and m hidden layers each made up of 1 node.

Since all weight matrices and bias vectors in this network are 1×1, let w_i and b_i denote the weight and bias of the i^{th} layer respectively. Also, let $a_i = w_i z^{[i-1]} + b_i$, and fix an activation function g for every layer. Let L be the square loss function (see equation (3)); then, taking just one input sample, x, we have cost function $J = L$. Consider the derivative $\frac{\partial J}{\partial w_j}$, for some

$1 \leq j \leq m$. From equations (1) and (3) and repeated application of the chain rule we obtain:

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} \\
&= 2(\hat{y} - y) \cdot \frac{\partial}{\partial w_j} \left(g(a_{m+1}) \right) \\
&= 2(\hat{y} - y) \cdot g'(a_{m+1}) \cdot \frac{\partial}{\partial w_j} \left(w_{m+1} z^{[m]} + b_{m+1} \right) \\
&= 2(\hat{y} - y) \cdot g'(a_{m+1}) \cdot w_{m+1} \cdot \frac{\partial}{\partial w_j} \left(z^{[m]} \right) \\
&= 2(\hat{y} - y) \cdot g'(a_{m+1}) \cdot w_{m+1} \cdot \frac{\partial}{\partial w_j} \left(g(a_m) \right) \\
&\vdots \\
&= 2(\hat{y} - y) \cdot g'(a_{m+1}) \cdot w_{m+1} \cdot \ldots \cdot g'(a_{j+1}) \cdot w_{j+1} \cdot \frac{\partial}{\partial w_j} \left(g(w_j z^{[j-1]} + b_j) \right) \\
&= 2(\hat{y} - y) \cdot g'(a_{m+1}) \cdot w_{m+1} \cdot \ldots \cdot g'(a_{j+1}) \cdot w_{j+1} \cdot g'(a_j) \cdot z^{[j-1]} \\
&= 2(\hat{y} - y) \cdot g'(a_j) \cdot z^{[j-1]} \cdot \prod_{i=j+1}^{m+1} w_i \cdot g'(a_i). \qquad (6)
\end{aligned}$$

To understand this equation better, we consider the derivatives of the activation functions mentioned in section 2.2, shown in figures 2 - 5.

We can see that for all $x \in \mathbb{R}$ we have $|\sigma'(x)| \leq 0.25$, $|\tanh'(x)| \leq 1$, $|\text{ReLU}'(x)| \in \{0, 1\}$ and $|\text{Linear}'(x)| \equiv 1$. From equation (6) we can see:

$$\left| \frac{\partial J}{\partial w_j} \right| \leq 2 \left| z^{[j-1]} \right| |\hat{y} - y| \prod_{i=j+1}^{m+1} |w_i|, \qquad (7)$$

taking g to be any of these four activation functions.

For large m and small j, the product of the magnitude of the weights dominates the right-hand side of this inequality. Hence if $|w_i| \ll 1$ for all i, this product will be close to zero, and the magnitude of $\frac{\partial J}{\partial w_j}$ will be very small. This is known as the **vanishing gradient problem**, and is exacerbated for deeper networks (i.e. larger m). It has the effect of bringing the learning algorithm to a halt before it has found an optimal solution.

Note that this can occur more easily when using tanh or σ, since their derivatives are always small, and tend towards zero for $|x|$ large. Hence their product in equation (6) will also contribute to this effect. Furthermore, if σ (respectively tanh) has been used on the output layer, then $|\hat{y} - y| \leq 1$ (respectively $|\hat{y} - y| \leq 2$), further limiting the size of $\frac{\partial J}{\partial w_j}$.

The reverse issue also occurs: the **exploding gradient problem**. If $|w_i| \gg 1$ for all i, then the size of $\frac{\partial J}{\partial w_j}$ can increase exponentially. Similar to the effect of having too large a learning rate, this problem can cause erratic movements around the error landscape and a failure to converge to a solution. This occurs less frequently for tanh and σ, for the same reasons by which they exacerbated the vanishing gradient issue.

In the case of Linear, whose gradient is identically 1, its gradients offer no dampening to the growth of the product of weights in equation (6), and $|\hat{y} - y|$ is unbounded. The same can be said for ReLU if $a_i \geq 0$ for all i. If this is not the case, then the right-hand side of equation (6) vanishes. In a network of greater width however, $\frac{\partial J}{\partial w_j}$ will be a sum of expressions similar to the right-hand side of equation (6), and so it is less likely that $\frac{\partial J}{\partial w_j}$ will vanish completely. Hence the exploding gradient problem can still occur, and both Linear and ReLU can be prone to this issue.

For ease of notation, we demonstrated this issue for a network of width 1 at every layer and using only one input sample, but the generalisation to any (deep) network is clear. $\frac{\partial J}{\partial w}$, for any

weight w, will always be the sum and product of weights and the derivative of the activation functions, and the loss function will contribute at most one term to each product.

Thus to avoid exploding or vanishing gradients, we must consider carefully the magnitude of our weights, especially in relation to which activation functions we are using.

Consider now an expression for $\frac{\partial J}{\partial b_j}$. With identical calculations as in the derivation of equation (6) we have:

$$\begin{aligned}\frac{\partial J}{\partial b_j} &= 2(\hat{y}-y)\cdot g'(a_{m+1})\cdot w_{m+1}\cdot \ldots \cdot g'(a_{j+1})\cdot w_{j+1}\cdot \frac{\partial}{\partial b_j}\left(g(w_j z^{[j-1]}+b_j)\right)\\ &= 2(\hat{y}-y)\cdot g'(a_{m+1})\cdot w_{m+1}\cdot \ldots \cdot g'(a_{j+1})\cdot w_{j+1}\cdot g'(a_j)\cdot 1\\ &= 2(\hat{y}-y)\cdot g'(a_j)\cdot \prod_{i=j+1}^{m+1} w_i\cdot g'(a_i). \end{aligned} \tag{8}$$

Note that $\frac{\partial J}{\partial b_j}$ depends much more on the weights of the network than its biases. Again generalising to a wider network, if two biases in the j^{th} layer are equal, $b_j=\tilde{b}_j$, but two weights in different rows differ, $w_j \neq \tilde{w}_j$, the values $g'(a_j)=g'(w_j z^{[j-1]}+b_j)$ and $g'(\tilde{a}_j)=g'(\tilde{w}_j z^{[j-1]}+\tilde{b}_j)$ will differ. The gradient of each bias will be similar to equation (8), and since $g'(a_j)\neq g'(\tilde{a}_j)$, the gradients will not be equal. Thus the biases will be updated to different values during gradient descent, and we can avoid the problem of constant initialisation described previously.

This has shown us that weight initialisation has a much larger impact on the success of a network than bias initialisation. In fact, all biases are typically initialised to zero [10].

2.4.3 Xavier and Kaiming Initialisations

There are many possible strategies to avoid both exploding and vanishing gradients, which vary depending on the activation functions used. We give the general idea behind a popular method, with a full explanation provided in appendix A.

The crux of the method is to assume that the gradients with respect to all weights, $\frac{\partial J}{\partial w}$, have mean zero. Then the variance of the gradients with respect to the weights in the l^{th} layer, $\text{Var}\left[\frac{\partial J}{\partial W^{[l]}}\right]$, determines the *magnitude* of these weights. If we can then keep these variances equal across all layers, i.e. keep $\text{Var}\left[\frac{\partial J}{\partial W^{[l]}}\right]$ constant as l varies, then the magnitude of our gradients will neither explode nor vanish.

It can then be shown that $\text{Var}\left[\frac{\partial J}{\partial W^{[l]}}\right]$ depends on $\text{Var}\left[W^{[l]}\right]$, and hence by sampling the weights of each layer from an appropriate distribution, we can control the magnitude of our gradients. Recall that the gradients are also dependent on the activation function used, and hence the exact choice of $\text{Var}\left[W^{[l]}\right]$ depends on the activation function as well.

For example, if we are using tanh, it can be shown that:

$$\text{Var}\left[W^{[l]}\right]=\frac{2}{n_l+n_{l-1}} \quad \forall\, 1\leq l\leq L, \tag{9}$$

achieves this, where n_l is the width of the l^{th} layer, and L is the total number of layers. We can then sample these weights from the Normal distribution, $W^{[l]}\sim\mathcal{N}\left(0,\frac{2}{n_l+n_{l-1}}\right)$, for example. This is known as **Xavier initialisation** [10].

Similarly, when using the ReLU function, a solution can be found in **Kaiming initialisation** [11], for which:

$$\text{Var}\left[W^{[l]}\right]=\frac{2}{n_l} \quad \forall\, 1\leq l\leq L. \tag{10}$$

It is important to note that the initial assumption we made is certainly not always true, essential as it is to the proof (and similarly for several other assumptions required in the full proof of the method). Nonetheless, Xavier initialisation has proven to be effective in practice, in a wide variety of use-cases [10].

Another important observation is that initialisation methods cannot necessarily prevent the exploding / vanishing gradient problems from occurring later on during the training of the network. There are many other methods to solve this problem; one example is **gradient clipping** [12]. Here, upper and lower limits are chosen for the gradient values, such that they can never become too large or too small.

2.5 Automatic Differentiation

Recall that our parameter optimisation algorithm is based on calculating the gradient of the error function J with respect to each of the network parameters θ. Since it would not be practical to derive analytic expressions for each of these gradients manually and evaluate them, we require an effective way of calculating them numerically.

One approach would be to calculate these using **finite-difference methods** (see appendix B), which use evaluations of the function itself to estimate the gradient. However, these can be computationally inefficient, numerically unstable and prone to error (which can then grow exponentially as it is propagated through the network) [13].

Another possible method is to utilise **symbolic programming**. Here, a computer algebra system can be used to derive explicit expressions for derivatives automatically, using only the expression of the function itself and basic differentiation laws. Although this would give a much more accurate value than finite-difference methods, it can also be computationally expensive. This occurs since, in the case of high-order derivatives or functions with many components, the expressions calculated can become large and complicated [13]. In the case of neural networks this is particularly true, since the error function J consists of the composition of many functions.

In practice, a third approach is generally used for neural networks, known as **automatic differentiation** [13], which combines aspects of the previous two methods. As in symbolic programming, the laws of differentiation (e.g. product rule, chain rule) are implemented algebraically to calculate derivatives. However, gradients are only ever evaluated at given points; we never attain a closed expression for the function's derivative, as in finite-difference methods.

To give an outline of automatic differentiation, consider a differentiable function with n inputs and k outputs:

$$\begin{aligned} f &: \mathbb{R}^n \to \mathbb{R}^k \\ \mathbf{x} = (x_1, \dots, x_n) &\mapsto f(\mathbf{x}) = (f_1(\mathbf{x}), \dots, f_k(\mathbf{x})) = (y_1, \dots, y_k). \end{aligned}$$

Suppose we would like to calculate all of its nk first-order derivatives, i.e. its Jacobian matrix:

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_k}{\partial x_1} & \dots & \frac{\partial y_k}{\partial x_n} \end{bmatrix} \tag{11}$$

There are two ways of calculating this full Jacobian using automatic differentiation. The first is known as **forward mode**, in which we fix an input x_i and calculate each partial derivative with respect to this input: $\frac{\partial y_j}{\partial x_i}$ for all $1 \leq j \leq k$. Repeating this for each $1 \leq i \leq n$, we calculate each *column* of the Jacobian.

Alternatively, we can calculate the *rows* of the Jacobian, via **reverse mode**. Here, we fix an output y_j and calculate all of its partial derivatives $\frac{\partial y_j}{\partial x_i}$ for all $1 \leq i \leq n$. We then repeat this for each $1 \leq j \leq k$.

As we shall see shortly, each row in reverse mode (respectively each column in forward mode) can be calculated in a single pass. Hence reverse mode requires k passes, while forward mode requires n, and so for computational efficiency we prefer reverse mode when $k \ll n$, and forward mode when $n \ll k$.

In the case of neural networks, our cost function J has only one output but a large number of inputs. For example, the network in figure 1 has $nm + 4m^2 + mk$ weights and $5m + k$ biases, hence its error function will have $nm + 4m^2 + mk + 5m + k$ inputs and only one output. This makes reverse mode the natural choice for implementing backpropagation, and so we will describe how it works in detail. An explanation of forward mode can be found in [13].

Automatic differentiation leverages a **computation graph** [14] to trace the dependency of each output on each input. Each node in the graph represents an input variable, an output variable, or an intermediate variable obtained by performing basic operations on previous (input or intermediate) variables. The (directed) edges of the graph represent the dependence of subsequent variables on previous ones. See figure 11 below as an example.

During the forward pass (i.e. the forward propagation step), as inputs are passed through the computation graph, these intermediate variables and their dependencies are stored. This allows us to calculate the gradients on the backward pass. Having fixed an output variable, y_j, we 'augment' every input variable x_i, and intermediate variable, v_l, with an **adjoint** $\bar{x}_i = \frac{\partial y_j}{\partial x_i}$, $\bar{v}_l = \frac{\partial y_j}{\partial v_l}$. Using the chain rule repeatedly, we can traverse the computational graph backwards to evaluate all of these adjoints (since earlier adjoints will depend on later ones). In fact, we have the following relationships (directly from the chain rule):

$$\bar{x}_i = \frac{\partial y_j}{\partial x_i} = \sum_{v_m} \bar{v}_m \frac{\partial v_m}{\partial x_i}, \qquad \bar{v}_l = \frac{\partial y_j}{\partial v_l} = \sum_{v_p} \bar{v}_p \frac{\partial v_p}{\partial v_l}, \tag{12}$$

where v_m (respectively v_p) are all nodes directly dependent on x_i (respectively v_l).

To illustrate, consider the simple neural network and cost function below:

Figure 10: A neural network with output $\hat{y}$ and cost function $J = L_{squ}$.

For simplicity, we pass only one sample input, x, through the network and hence the cost function J is equal to the loss function, which we take to be squared error loss $L_{squ} = (\hat{y} - y)^2$. We let $z = \tanh(w_1 x + b_1)$ and $\hat{y} = w_2 z + b_2$. This network has the following computation graph:

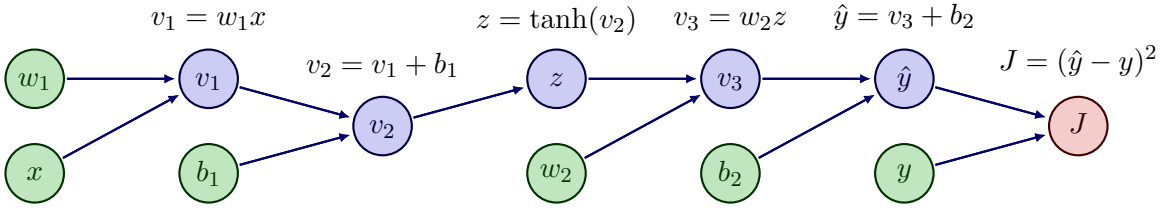

Figure 11: Computation graph of the network shown in figure 10. Input variables are shown in green, intermediate variables in purple, and the output variable in red.

Note that although y is technically an input variable (of the computation graph), we do not require $\frac{\partial J}{\partial y}$ for backpropagation. Although the same can be said for x, we will require $\frac{\partial \hat{y}}{\partial x}$ when working with differential equations, and will calculate it using exactly this method.

The chain rule then gives us, for example, the following relationships:

$$\begin{aligned}
\bar{\hat{y}} &= \frac{\partial J}{\partial \hat{y}} \\
&= 2(\hat{y} - y), \\
\bar{v}_3 &= \frac{\partial J}{\partial v_3} \\
&= \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_3} + \frac{\partial J}{\partial y} \frac{\partial y}{\partial v_3} \\
&= \bar{\hat{y}} \frac{\partial \hat{y}}{\partial v_3} \\
&= 2(\hat{y} - y) \cdot 1,
\end{aligned}$$

$$\begin{aligned}
\bar{b}_2 &= \frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} + \frac{\partial J}{\partial y} \frac{\partial y}{\partial b_2} = \bar{\hat{y}} \frac{\partial \hat{y}}{\partial b_2} \\
&= 2(\hat{y} - y) \cdot 1, \\
\bar{w}_2 &= \frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} + \frac{\partial J}{\partial y} \frac{\partial y}{\partial w_2} \\
&= \frac{\partial J}{\partial \hat{y}} \left(\frac{\partial \hat{y}}{\partial v_3} \frac{\partial v_3}{\partial w_2} + \frac{\partial \hat{y}}{\partial b_2} \frac{\partial b_2}{\partial w_2} \right) \\
&= \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_3} \frac{\partial v_3}{\partial w_2} = \frac{\partial J}{\partial v_3} \frac{\partial v_3}{\partial w_2} = \bar{v}_3 \frac{\partial v_3}{\partial w_2} \\
&= 2(\hat{y} - y) \cdot z.
\end{aligned}$$

Notice that these verify the general formulae given in equation 12. Taking some initial values for our six input variables, we can calculate all derivatives in one pass, as shown in table 1 below.

Forward Pass	Backward Pass
$x = 0.2$	$\bar{J} = \frac{\partial J}{\partial J} = 1$
$w_1 = 0.4$	$\bar{\hat{y}} = \bar{J}\frac{\partial J}{\partial \hat{y}} = 1 \cdot 2(\hat{y} - y) = -1.482$
$b_1 = 0.6$	$\bar{b}_2 = \bar{\hat{y}}\frac{\partial \hat{y}}{\partial b_2} = -1.482 \cdot 1 = -1.482$
$w_2 = 0.1$	$\bar{v}_3 = \bar{\hat{y}}\frac{\partial \hat{y}}{\partial v_3} = -1.482 \cdot 1 = -1.482$
$b_2 = 0.1$	$\bar{w}_2 = \bar{v}_3\frac{\partial v_3}{\partial w_2} = -1.482 \cdot z = -0.876$
$y = 0.9$	
$v_1 = w_1 \cdot x = 0.08$	$\bar{z} = \bar{v}_3\frac{\partial v_3}{\partial z} = -1.482 \cdot w_2 = -0.1482$
$v_2 = v_1 + b_2 = 0.68$	$\bar{v}_2 = \bar{z}\frac{\partial z}{\partial v_2} = -0.1482 \cdot (1 - \tanh^2(v_2)) = -0.0963$
$z = \tanh(v_2) = 0.592$	$\bar{b}_1 = \bar{v}_2\frac{\partial v_2}{\partial b_1} = -0.0963 \cdot 1 = -0.0963$
$v_3 = w_2 \cdot z = 0.0592$	$\bar{v}_1 = \bar{v}_2\frac{\partial v_2}{\partial v_1} = -0.0963 \cdot 1 = -0.0963$
$\hat{y} = v_3 + b_2 = 0.1592$	$\bar{w}_1 = \bar{v}_1\frac{\partial v_1}{\partial w_1} = -0.0963 \cdot x = -0.0193$
$J = (\hat{y} - y)^2 = 0.549$	$\bar{x} = \bar{v}_1\frac{\partial v_1}{\partial x} = -0.0963 \cdot w_1 = -0.0385$

Table 1: Calculation of one forward and backward pass of reverse mode automatic differentiation, carried out for the neural network in figure 10.

Hence, in its simplest form, an automatic differentiation library requires only the capacity to implement the basic rules of differentiation (chain rule, product rule etc.), as well as the derivative of common functions (e.g. exponential, logarithm, trigonometric functions). Since the adjoints are computed and stored node by node as we traverse the computation graph backwards, we avoid the 'expression swell' produced by symbolic differentiation.

The main drawback of automatic differentiation is the memory cost of storing all of these adjoints; reducing the storage required is an ongoing area of research [15]. However, it can also be shown that automatic differentiation is computationally more efficient than finite-difference methods and symbolic differentiation [13]. This has made it is the preferred method when working with (especially large) neural networks, where many epochs of training are required and thus computational speed is of great importance.

There is one final benefit to automatic differentiation, which will be fundamental when we apply neural networks to differential equations. This is the ability to efficiently calculate the following product of a vector $r = (r_1, \dots, r_k) \in \mathbb{R}^k$ with the transposed Jacobian [13]:

$$\mathbf{J}_f^{\mathbf{T}} \cdot r = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_k}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_k}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_k \end{bmatrix} = \begin{bmatrix} r_1\frac{\partial y_1}{\partial x_1} + & \cdots & + r_k\frac{\partial y_k}{\partial x_1} \\ & \vdots & \\ r_1\frac{\partial y_1}{\partial x_n} + & \cdots & + r_k\frac{\partial y_k}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n \tag{13}$$

Notice that, for any fixed output y_j, $1 \leq j \leq k$, all its partial derivatives $\frac{\partial y_j}{\partial x_i}$, $1 \leq i \leq n$, are multiplied by r_j in this resulting column vector. Recall also that $\frac{\partial y_j}{\partial x_i}, 1 \leq i \leq n$, can all be obtained by performing one pass of reverse mode automatic differentiation from y_j with adjoint $\bar{y}_j = 1$. Hence, if we were to set the adjoint $\bar{y}_j = r_j$ instead of equal to 1, we could calculate all n of these values $r_j\frac{\partial y_j}{\partial x_i}$, $1 \leq i \leq n$, in one backward pass. Repeating this for all $j = 1, \dots, k$, we can

calculate the entire vector product in equation (13) above, *without ever calculating the Jacobian itself* (or its tranpose).

This is a powerful method to speed up the calculation of Jacobian-vector products, and another significant advantage of using automatic differentiation. We shall make explicit use of it in our later applications.

3 Function Approximation

We explain why neural networks are suited to the task of function approximation, and then give an introduction to **PyTorch**, the machine learning library used throughout this project. We then demonstrate a basic example of using a neural network in PyTorch to approximate $\sin(x)$.

3.1 The Universal Approximation Theorem

The fundamental application of neural networks which we will focus on is that of function approximation. In particular, we seek to leverage a result known as the **Universal Approximation Theorem**. The general implication of the theorem is that neural networks can be used to approximate virtually any function to arbitrary accuracy. Equivalently, it establishes the density of neural networks (as a space of functions) in various other function spaces. Many different versions of the theorem exist, dependent on the specific details of the statement, for example: the properties of the function we seek to approximate, the properties of the activation functions used, the limitations on the width and depth of the neural network.

Two of the earliest versions were results of Kurt Hornik. In 1989 he showed that a feedforward neural network with one hidden layer (with sufficiently many neurons) can approximate any (L^p-)measurable function on $\mathbb{R}^n$ to arbitrary accuracy, as long as the activation function used is bounded and non-constant [3]. In 1991 he then established the same result for any function continuous on a compact subset $K \subset \mathbb{R}^n$, as long as the activation function is continuous (as well as bounded and non-constant) [16].

We state in full a version of the theorem which followed these two results in 1992 [17]:

Theorem 1 (Universal Approximation Theorem) *Let $\epsilon > 0$, let $n, k \in \mathbb{N}$ and let $K \subset \mathbb{R}^n$ be compact. Let $f : K \to \mathbb{R}^k$ and $g : \mathbb{R} \to \mathbb{R}$ both be continuous.*

Then there exist a natural number $m \in \mathbb{N}$, matrices $W^{[1]} \in \mathbb{R}^{m \times n}$, $W^{[2]} \in \mathbb{R}^{k \times m}$ and column vectors $b^{[1]} \in \mathbb{R}^m$, $b^{[2]} \in \mathbb{R}^k$ such that, for $N(x) \coloneqq W^{[2]} \cdot g\left(W^{[1]} \cdot x + b^{[1]}\right) + b^{[2]}$ (where g is applied component-wise to the vector in $\mathbb{R}^m$), we have:

$$\sup_{x \in K} \|f(x) - N(x)\| < \epsilon$$

if and only if g is not polynomial.

In other words, a feedforward neural network with one hidden layer (of sufficiently many nodes) can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary accuracy, so long as the activation function used on the hidden layer is not polynomial. We use this version of the Universal Approximation Theorem as reference since, when using neural networks to solve differential equations, most of the functions f we seek to approximate will be of this nature.

Observe that this theorem further highlights the importance of the role of activation functions in neural networks. We notice also that the statement is of the existence of such a network, and the proof (see [17]) is not constructive. Thus we are still left with the task of finding optimal values for m and our weights and biases.

Furthermore, although we know such an approximation is always possible with only one hidden layer, this does not mean that a network with more layers will not converge *faster* in practice. In fact, the ideal choice of network structure, initial weight and bias values, activation functions, optimisation algorithm and many other hyperparameters remain to be determined. Hence approximating a given function f with a neural network continues to be an experimental and open-ended challenge.

Before we see an example of function approximation, we introduce the method by which we will implement neural networks in Python.

3.2 PyTorch

PyTorch is a relatively recent, open-source Python library (first released in 2016), written mostly in C++ and developed for machine learning. It is generally designed to balance user-friendliness with computational speed. In particular, it has many built-in memory optimisation features (making it ideal for use with a GPU) and a highly efficient automatic differentiation package, **autograd** [18].

The main data structure used in PyTorch is known as a **tensor**: this is very similar to a multidimensional array in NumPy [19]. In fact, PyTorch is designed to be compatible with NumPy, such than any PyTorch tensor has a method ('numpy') to convert it to a NumPy array of the same shape, data type and input values. PyTorch also shares much of the same functionality as NumPy, including broadcasting and **vectorisation** [20].

Vectorisation (also known as **SIMD: Single Instruction, Multiple Data**) provides a powerful method for applying a given process to multiple inputs [21]. For example, in the neural network outlined below, we have M scalar sample data points which we want to pass through our network. This could be achieved by passing each one through individually in a for loop. However, a CPU is generally able to perform a small number, say p, of such scalar operations simultaneously (i.e. in parallel). Hence carrying out one at a time is computationally inefficient.

Functions in PyTorch are able to take advantage of this small-scale parallelisation. We format all M inputs into one vector (tensor) of shape $(M, 1)$, and all subsequent operations are applied component-wise to each element of this tensor. All M outputs can then be calculated in M/p passes, instead of M. This significantly reduces computation time, particularly with a large number of inputs.

Vectorising a given function also requires no change in code. For example, when implementing the example below (figure C.2), the activation function $\tanh : \mathbb{R} \to \mathbb{R}$ is applied to a tensor of shape ('batchSize', 'numHiddenNodes'). Regardless of the values of these two integers, tanh will be applied component-wise to each element of the tensor. This also generalises to tensors of higher dimensions.

3.3 Example: Function Approximation

To illustrate, suppose we wanted to approximate an unknown function:

$$\begin{aligned} f : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto y, \end{aligned}$$

based on M sample data points $\{(x_1, y_1), \ldots, (x_M, y_M)\}$ (where $y_j = f(x_j)$, $1 \leq j \leq M$). We can achieve this by training a neural network with parameters θ:

$$\begin{aligned} N(x; \theta) : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto N(x; \theta), \end{aligned}$$

with cost function (having chosen a loss function L):

$$J(\theta) = \frac{1}{M} \sum_{j=1}^{M} L\left(y_j, N(x_j; \theta)\right). \tag{14}$$

We implement this method in PyTorch, taking $f(x) = \sin(x)$, $M = 30$, and $\{x_1, \ldots, x_M\}$ to be evenly-spaced points across the interval $[-3, 3]$. This can be created as a subclass of the Dataset class in PyTorch (code shown in figure C.1).

We then create a neural network with one input, one output, and one hidden layer made up of sixteen nodes. We use a tanh activation function on the hidden layer, and a Linear activation

function on the outer layer, so as to not restrict the range of our network's output. In PyTorch this is done by creating a subclass of the PyTorch Module class (see figure C.2). Note that by default, weights are initialised according to a variant of Kaiming initialisation (equation (10)), and biases are set to zero [20]. We found this to perform well in most examples, especially where the network has only one hidden layer.

Finally, we choose a loss function (in this case squared error loss) and an optimisation algorithm (here we use the gradient descent described in section 2.3, with learning rate 10^{-2}; see section 4 for more variants), both of which are inbuilt functions in PyTorch. We then train the network for 10,000 epochs, recording the cost after each epoch (see figure C.3). We use a DataLoader object to load batches of our training data and specify a batch size; this will be explained in section 4.1.

Figure 12 shows the output of this network after 10,000 epochs of training, while figure 13 shows the total cost value on each epoch. The final cost value is approximately 1.34×10^{-2}.

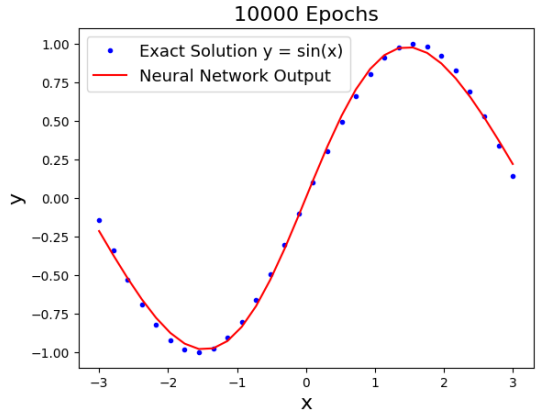

Figure 12: Output of a network trained to approximate $\sin(x)$ with tanh activation function in the range $[-3, 3]$, after 10,000 epochs of training.

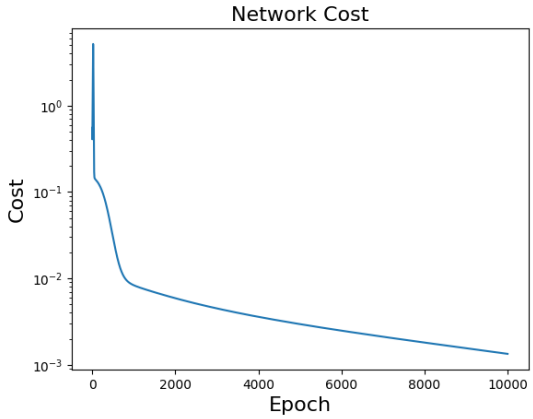

Figure 13: Cost values during training of network trained for 10,000 epochs to approximate $\sin(x)$ with tanh activation function.

At this point, is interesting to consider the effect of the activation function on our network's output. By changing only one line of code in our neural network class (figure C.2), we can compare tanh to the sigmoid function, ReLU and Linear (figures 14, 15, and 16 respectively). We fix all other hyperparameters (number of epochs, learning rate, sample data points etc.).

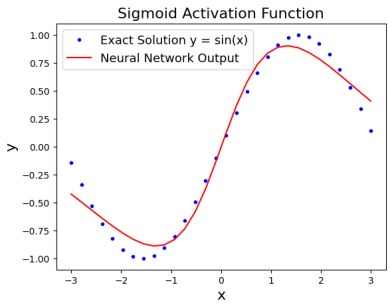

Figure 14: Network trained to approximate $\sin(x)$ with $\sigma(x)$ activation function. The final cost is approximately 0.013.

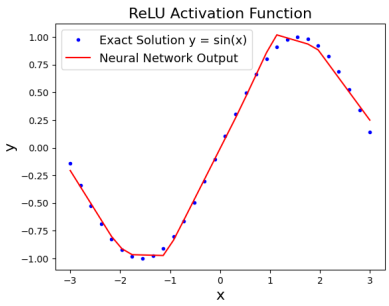

Figure 15: Network trained to approximate $\sin(x)$ with the ReLU activation function. The final cost is approximately 0.002.

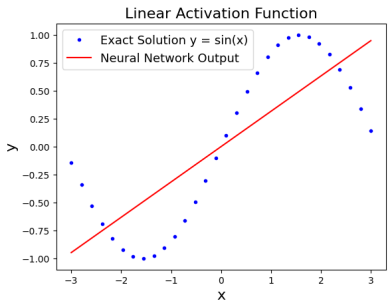

Figure 16: Network trained to approximate $\sin(x)$ with the Linear activation function. The final cost is approximately 0.186.

As seen in equation 2, using the Linear activation function has produced a linear output, and thus a poor approximation of $\sin(x)$, with the highest cost value.

ReLU, as a non-linear function, has provided a better approximation than this, and has attained a smaller cost than σ after the same number of epochs. This can be attributed to its larger gradient for positive x (see figures 2 and 4 for comparison) speeding up the gradient descent

algorithm. However, since it is not a smooth function, it has produced an output which is not smooth.

σ has produced a smooth output, but after 10,000 epochs has a cost value approximately 10 times larger than tanh. Again, this is due to tanh accelerating the gradient descent algorithm with a larger gradient than σ, particularly close to zero (see figures 2 and 3).

This gives an indication as to why tanh will be our activation function of choice when we apply neural networks to differential equations. The solutions we seek to approximate will generally be smooth, and so fast convergence to a smooth output is desirable.

4 Optimisation Algorithms

One of the most important features in the process of training a neural network is the optimisation algorithm. In section 2.3, we saw the basic gradient descent algorithm, with only one hyperparameter, the (fixed) learning rate α. Naturally, there are many ways to alter this algorithm to improve its performance.

4.1 Batch Size

Firstly, notice that in section 2.3 we took a training set of size M and updated our parameters w and b after evaluating our cost function J on *all* elements of this training set. This is known as **batch** gradient descent. To better represent the underlying data set, we generally require M to be large. This means that the computational cost of performing forward propagation, error calculation and backpropagation on all M samples is expensive. This can slow down learning.

One alternative to this is **stochastic** gradient descent [2]. Here, we select just one point at random from our training set, pass it through the network and update our parameters. We then continue this process, randomly selecting a different point each time, until all M points have been used. Hence we have M parameter updates per epoch, instead of just one. This has the advantage of updating our parameters quickly, while requiring much less memory and computational power. It also introduces a random element to our training, meaning our progress through the error landscape becomes 'noisy'. This means the network can more easily avoid local minima, since its path is no longer deterministic, as in the case of batch gradient descent.

Conversely, the solutions found quickly by stochastic gradient descent can often be suboptimal, since they do not simultaneously optimise the network for multiple inputs. The noise in the network's progress can also lead to an oscillation around the global minimum, rather than convergence to it. Furthermore, training only on single data points prevents us from taking advantage of vectorisation (mentioned in section 3.2) or parallelisation on GPU (see section 6.3), which can speed up learning for a large number of inputs.

A balance between the two methods can be found in **mini-batch** gradient descent [2]. Here we fix a number $m \ll M$, and randomly select m inputs to pass through our network before updating our parameters. By selecting m effectively (the exact value would depend on the given example and computational power available), we can attempt to balance the advantages of the batch and stochastic methods, while limiting their disadvantages. We can introduce enough noise to avoid local minima, but not too much to hinder convergence. Similarly, we can implement vectorisation or parallelisation, while not slowing down our learning by training on too much input data.

We will see a comparison of these three types of gradient descent in section 5.2.2. An indication of their paths through the error landscape is given in figure 17.

4.2 Momentum

A typical problem with (all three kinds of) simple gradient descent is that the convergence can be slow, especially for larger networks. One reason for this can be that the gradients across parts of our error surface are small, i.e. the surface has relatively flat regions. This means that updates to our parameters in algorithm 1 will also be small. Meanwhile, another reason could be that our surface has noisy gradients, i.e. the surface contains irregular 'bumps'. This will lead to the

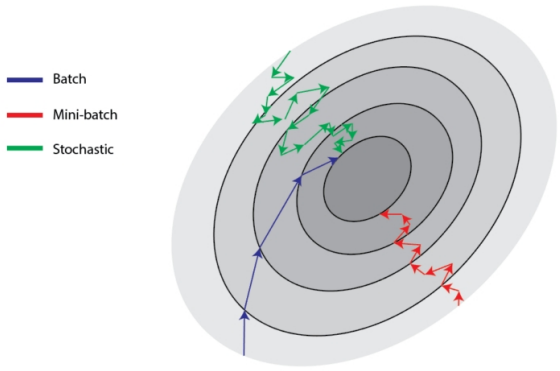

Figure 17: Paths through the error landscape (shown as a contour plot) of each type of gradient descent. Processing fewer data points before each parameter update leads to a noisier path towards the minimum (image from baeldung.com/batch-size).

parameters being updated too noisily, in varying directions, and slow down convergence to the minimum.

One method of speeding up convergence is known as **momentum** [2]. In this method we update our parameters θ not just with the gradient of our current iteration, $\frac{\partial J}{\partial \theta}$, but with an **exponentially weighted moving average** of previous gradients, denoted by m_θ:

Algorithm 2 (Gradient Descent with Momentum)

$$\begin{aligned} m_\theta &\leftarrow \beta m_\theta + (1-\beta)\frac{\partial J}{\partial \theta}, \\ \theta &\leftarrow \theta - \alpha m_\theta, \end{aligned}$$

where $\beta \in [0, 1)$ is a fixed constant, and m_θ is set to zero during the first iteration. β determines the weight that we give to the previous gradients; typically, it is taken to be close to 1.

The way m_θ is updated is known as an exponentially weighted moving average since, as we perform further iterations, earlier values of $\frac{\partial J}{\partial \theta}$ have exponentially less influence. To illustrate, fix a parameter θ and let m_t denote the value of m_θ on iteration (i.e. parameter update) t. Similarly, let d_t denote the value of $\frac{\partial J}{\partial \theta}$ on iteration t. Then:

$$\begin{aligned} m_t &= (1-\beta)d_t + \beta m_{t-1} \\ &= (1-\beta)d_t + \beta((1-\beta)d_{t-1} + \beta m_{t-2}) \\ &= (1-\beta)d_t + \beta(1-\beta)d_{t-1} + \beta^2 m_{t-2} \\ &= (1-\beta)d_t + \beta(1-\beta)d_{t-1} + \beta^2((1-\beta)d_{t-2} + \beta m_{t-3}) \\ &\vdots \\ &= (1-\beta)(d_t + \beta d_{t-1} + \beta^2 d_{t-2} + \beta^3 d_{t-3} + \ldots + \beta^{t-1} d_1). \end{aligned} \tag{15}$$

Since $\beta \in [0, 1)$, we have $\lim_{n\to\infty} \beta^n = 0$, and hence m_t depends only on the most recent values of $\frac{\partial J}{\partial \theta}$. In fact, for β close to 1, we have that $\beta^{\frac{1}{1-\beta}} \approx e^{-1} \approx 0.37$, since:

$$\begin{aligned} \lim_{\beta\to 1} \beta^{\frac{1}{1-\beta}} &= \lim_{\gamma\to 0} (1+\gamma)^{\frac{1}{-\gamma}} && \text{(Substitution } \beta = 1+\gamma) \\ &= \lim_{\gamma\to 0} \left((1+\gamma)^{\frac{1}{\gamma}}\right)^{-1} && \text{(Since } \lim_{\gamma\to 0}(1+\gamma)^{\frac{1}{\gamma}} \text{ exists)} \\ &= e^{-1}. \end{aligned}$$

Hence after $1/(1-\beta)$ iterations, the influence of a given gradient is reduced by almost two thirds. For this reason, we can roughly think of $1/(1-\beta)$ as the number of preceding gradients that

we factor into our moving average. E.g. if $\beta = 0.9$ (a typical choice), then the average depends heavily on the previous $1/(1-0.9) = 1/0.1 = 10$ gradients. While if $\beta = 0.999$, the average depends on the previous $1/0.001 = 1000$ gradients, and if $\beta = 0$ we consider only the current gradient, and recover the standard gradient descent algorithm (algorithm 1).

The name 'momentum' is inspired by the idea of our physical movement around this error surface. m_θ measures our cumulative speed across the surface: if $\frac{\partial J}{\partial \theta}$ always points in the same direction, this speed will remain relatively constant, while if the gradient changes direction frequently we will be slowed. Thus m_θ can be thought of as our velocity. Since our mass does not play a role, we can presume to have unit mass and thus our velocity and momentum coincide.

Momentum speeds up our learning algorithm in a few ways [2]. By giving greater weight to the average of our gradients, we dampen the movement along parameter axes where gradients are fluctuating greatly. Their oscillations will be cancelled in the averaging process, and thus our path towards the minimum will be smoothed. On the other hand, for parameters with relatively constant gradients, movement along their axes will remain unchanged by the averaging process. This is illustrated in figure 18.

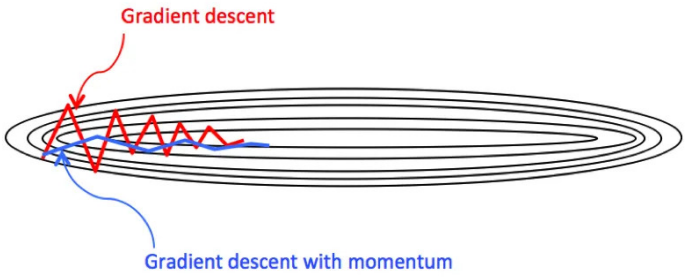

Figure 18: Smoothing effect of momentum on gradient descent. Movement is dampened along parameter axes where gradients fluctuate more, while progress is unchanged for parameters with more constant gradients (taken from towardsdatascience.com/gradient-descent).

Furthermore, if we enter a shallow minimum or a plateau of our error surface, the momentum factor will delay the slowing effect of the small gradients found here. This increases our chances of exiting this neighbourhood before these small gradients can bring our learning algorithm to a halt.

4.2.1 Bias Correction

Note that momentum in this form creates a different problem for early values of t, i.e. early parameter updates. For example (using again the notation of equations (15)), we have (taking $\beta = 0.9$ and noting $m_0 = 0$):

$$\begin{aligned} m_1 &= (1-\beta)d_1 + \beta m_0 \\ &= 0.1 d_1. \end{aligned}$$

Hence early on in our training, the momentum algorithm will introduce a bias towards much smaller values of m_θ, and thus towards smaller updates to θ also. Before we have accumulated enough gradients into our velocity term, applying momentum to our learning algorithm will slow our progress by a significant factor.

To counteract this we introduce a **bias correction** term. Recall from equation (15) that:

$$\begin{aligned} m_t &= (1-\beta)(d_t + \beta d_{t-1} + \beta^2 d_{t-2} + \beta^3 d_{t-3} + \ldots + \beta^{t-1} d_1). \\ &= (1-\beta)\sum_{i=0}^{t-1} \beta^i d_{t-i} \end{aligned}$$

Then noting that:

$$\sum_{i=0}^{t-1} \beta^i = \frac{1-\beta^t}{1-\beta} \qquad \Longrightarrow \qquad (1-\beta)\sum_{i=0}^{t-1} \beta^i = 1-\beta^t,$$

we can see that dividing m_t by $(1-\beta^t)$ will normalise m_t with respect to the factors introduced by β. m_t is now simply the weighted average of the previous gradients, and we have removed this initial bias towards smaller values. For example, $m_1 = d_1$ under this correction. Since $\lim_{t\to\infty}(1-\beta^t) = 1$, this bias correction only influences early values of m_t, as desired.

Our gradient descent algorithm with bias-corrected momentum (on iteration t) now becomes:

Algorithm 3 (Gradient Descent with Bias-Corrected Momentum)

$$\begin{aligned} m_\theta &\leftarrow \beta m_\theta + (1-\beta)\frac{\partial J}{\partial \theta}, \\ \widehat{m}_\theta &\leftarrow \frac{m_\theta}{1-\beta^t}, \\ \theta &\leftarrow \theta - \alpha \widehat{m}_\theta. \end{aligned}$$

4.3 RProp

Although momentum can greatly speed up convergence, it comes with other problems. Choosing the correct value of β can be as difficult as finding the optimal value of the learning rate, α. Meanwhile, since gradients in each parameter direction might vary greatly in magnitude, fixing one learning rate for all parameters can make convergence to the global minimum more difficult. Another optimisation algorithm was developed to avert this, known as **RProp** (short for **Resilient Propagation**) [22].

The idea behind RProp is to have a separate learning rate for each parameter, and to dynamically update these throughout training. For a given parameter θ on a given iteration, we measure the sign of its gradient and compare it to the sign of the gradient on the previous iteration. If the gradient has changed sign, then we have moved over the minimum (i.e. taken too large a step in this parameter direction) and so should undo our last step and reduce the learning rate for this parameter. Otherwise, we can increase the learning rate to accelerate towards the minimum.

As in the previous section, fix a parameter θ and let d_t denote $\frac{\partial J}{\partial \theta}$ on iteration t. Let $0 < \epsilon^- < 1 < \epsilon^+$ denote the two multiplicative factors we will use to respectively decrease and increase the learning rates. Let $\alpha_t > 0$ and $\Delta_t \in \mathbb{R}$ respectively denote the learning rate and (signed) step size for θ on iteration t (we will see in the algorithm why there is a distinction between α_t and Δ_t in this case). Then to calculate Δ_t we perform the following steps [22].

Algorithm 4 (RProp)

First, we determine whether to increase or decrease the magnitude of the learning rate based on the signs of d_t and d_{t-1}:

$$\alpha_t = \begin{cases} \epsilon^+ \alpha_{t-1}, & \text{if } d_t d_{t-1} > 0, \\ \epsilon^- \alpha_{t-1}, & \text{if } d_t d_{t-1} < 0, \\ \alpha_{t-1}, & \text{if } d_t d_{t-1} = 0. \end{cases} \tag{16}$$

Then we check which direction the step should take, using the sign of the current gradient d_t:

$$\Delta_t = \begin{cases} 0, & \text{if } d_t = 0, \\ -\operatorname{sgn}(d_t) \cdot \alpha_t, & \text{if } d_t d_{t-1} \geq 0, \\ -\Delta_{t-1}, & \text{if } d_t d_{t-1} < 0. \end{cases} \tag{17}$$

In the case $d_t d_{t-1} < 0$, we want to backtrack to our previous position in the error landscape, with our learning rate for θ having been reduced by equation (16). This creates a problem on iteration $t+1$ however: since d_t and d_{t+1} will now have opposing signs, the next iteration will decrease the learning rate for θ once again. To avoid this, we set $d_t \leftarrow 0$ after equation (17), such that on the next iteration we have $\alpha_{t+1} = \alpha_t$ and $\Delta_{t+1} = -\operatorname{sgn}(d_{t+1}) \cdot \alpha_{t+1}$ (note this is *only* in the case where $d_t d_{t-1} < 0$). This is the reason for the distinction between Δ_t and α_t.

We then update θ appropriately:

$$\theta \leftarrow \theta + \Delta_t.$$

Note we would also typically set minimum and maximum values for α_t, and compare with these values before updating α_t in equation (16). This leaves fives choices of hyperparameters in RProp: ϵ^+, ϵ^-, Δ_0, α_{max} and α_{min}. Through experimentation in [22], they found the following to be optimal in a wide variety of examples: $\epsilon^+ = 1.2$, $\epsilon^- = 0.5$, $\Delta_0 = 0.1$, $\alpha_{max} = 1.0$ and $\alpha_{min} = 10^{-6}$, although they noted that the algorithm was generally resilient to a wide range of choices for these values.

In fact, the main problem with the RProp algorithm is not the number of hyperparameters it introduces, but its poor performance when training with mini-batches. Here, gradient size and direction for a given parameter can vary a great deal between successive mini-batches. When using RProp, this noisiness can cause too many updates to parameter learning rates, and thus slow convergence. We will see an example of this in section 5.2.4.

Another issue is that, by considering only the sign of the gradient, RProp does not factor in the gradient's magnitude appropriately. For example, if the gradient has value 0.1 on seven successive mini-batches, and then -0.7 on the eighth, we would expect our optimisation to approximately cancel these out. In RProp however, the learning rate would be increased seven times and decreased only once, and so we will continue to move too quickly in this direction. This scenario is very typical of mini-batch training, and much less so of batch training, since our path in batch training is entirely deterministic, and hence changes in gradient are much more uniform.

Since mini-batch training is often necessary for large datasets, this is a serious limitation of RProp. Hence a modified version was created to remedy this issue.

4.4 RMSProp

The successor to RProp is known as **RMSProp** (short for **Root Mean Squared Propagation**). It seeks to combine the best features of gradient descent and RProp, while limiting their drawbacks.

To begin to see how RMSProp works, notice there is another way we can view the RProp algorithm. By considering only the sign of the gradient, instead of its magnitude, we are effectively considering the *normalised* value of the gradient. Then, the problem when using RProp on mini-batches is that the magnitude of the gradient may vary greatly between successive mini-batches. By normalising, we lose this information and thus slow convergence.

RMSProp addresses this by keeping an exponentially weighted moving average of the squared gradients (similar to the method used in momentum) and dividing the current gradient with this average instead [23]. Implementing this within the usual gradient descent algorithm allows us to successfully train on mini-batches, while retaining the adaptive learning rate of RProp.

Fixing a parameter θ and a learning rate α, let v_θ denote the moving average of the squared gradients, where we initialise v_θ to zero. Fix a smoothing factor $\mu \in [0, 1)$ for the moving average (analogous to β in momentum; a typical value is $\mu = 0.99$). Then on each iteration of RMSProp we have:

Algorithm 5 (RMSProp)

$$v_\theta \leftarrow \mu v_\theta + (1 - \mu) \left(\frac{\partial J}{\partial \theta} \right)^2,$$

$$\theta \leftarrow \theta - \left(\frac{\alpha}{\sqrt{v_\theta} + \epsilon} \right) \frac{\partial J}{\partial \theta}.$$

Here $\epsilon > 0$ is a small constant (typically 10^{-8}) used solely for numerical stability (i.e. to prevent division by a float close to zero). Note that the squaring of $\frac{\partial J}{\partial \theta}$ and the division by the square root of v_θ is what gives RMSProp its name.

Thus the learning rate for θ is $\alpha/(\sqrt{v_\theta} + \epsilon)$, and we have found a way of incorporating RProp's dynamically-updating, parameter-specific learning rates into our usual gradient descent algorithm.

The visual impact of RMSProp's effect on our progress through the error landscape is similar to momentum's, as shown in figure 18. By normalising each parameter's gradient with its running average, we dampen oscillations in parameter directions with varying gradients. Meanwhile, for parameters whose gradients are relatively constant, we leave progress in these directions unchanged.

4.5 Adam

In 2015, the ideas from all of the preceding adaptations to gradient descent were incorporated together in a single learning algorithm, known as **Adam** [24]. In essence, Adam combines RMSProp with momentum, while also correcting the bias of v_θ from the RMSProp algorithm. This combination has proved Adam to be a resilient algorithm which performs well in an extensive range of use-cases of neural networks [25].

The name Adam is short for **Adaptive Moment Estimation**, since the moving averages m_θ and v_θ (see algorithm 6 below) are in fact estimates of the mean and variance of $\frac{\partial J}{\partial \theta}$, known respectively as the **first** and **second moments** of $\frac{\partial J}{\partial \theta}$.

The full Adam algorithm is as follows. We fix:

- $\alpha > 0$: the learning rate,
- $\beta \in [0, 1)$: the smoothing factor for the weighted average of the gradients,
- $\mu \in [0, 1)$: the smoothing factor for the weighted average of the *square* gradients,
- $\epsilon > 0$: a small positive number for numerical stability when performing division.

We initialise the average of the gradients $m_\theta = 0$, and similarly the average of the square gradients $v_\theta = 0$. Then on iteration t of the Adam algorithm we have, for each parameter θ:

Algorithm 6 (Adam)

$$
\begin{aligned}
m_\theta &\leftarrow \beta m_\theta + (1-\beta)\frac{\partial J}{\partial \theta}, && \textit{(Update average of gradients, } m_\theta\textit{)} \\
v_\theta &\leftarrow \mu v_\theta + (1-\mu)\left(\frac{\partial J}{\partial \theta}\right)^2, && \textit{(Update average of square gradients, } v_\theta\textit{)} \\
\widehat{m}_\theta &\leftarrow \frac{m_\theta}{1-\beta^t}, && \textit{(Correct bias in } m_\theta\textit{)} \\
\widehat{v}_\theta &\leftarrow \frac{v_\theta}{1-\mu^t}, && \textit{(Correct bias in } v_\theta\textit{)} \\
\theta &\leftarrow \theta - \alpha\frac{\widehat{m}_\theta}{\sqrt{\widehat{v}_\theta}+\epsilon}. && \textit{(Update parameter } \theta\textit{)}
\end{aligned}
$$

Typically, default values $\beta = 0.9$, $\mu = 0.999$ and $\epsilon = 10^{-8}$ perform well in the Adam algorithm, and rarely need to be changed [24]. The learning rate, α, is generally the only hyperparameter in the algorithm which requires manual tuning to find an optimal value.

We will see a comparison of all of these optimisation algorithms in section 5.2.4.

5 Application to Differential Equations

Differential equations are found in virtually all areas of science and mathematics; they are fundamental to understanding the behaviour of any physical system. Although an analytic solution to a (system of) differential equations is always preferred, in reality this is often not attainable or practical. The system of equations could be incredibly difficult (or even impossible) to solve analytically, or it could be too large to solve in a realistic time frame. This is where numerical solutions to differential equations become critical.

Many traditional methods exist for solving differential equations numerically, for example the Euler or Runge-Kutta methods (see appendix B for a reminder of these). These are known as

finite-difference methods, since they rely on discretising the continuous domain of a differential equation and approximating the solution function at these discrete points. Then, to evaluate this approximation, they utilise the Taylor expansion of a function in terms of its derivatives. Due to the ubiquitous nature of differential equations, new methods or improvements on these traditional methods are always being sought.

We outline the general method for solving differential equations using neural networks, which differs greatly from finite-difference methods. We then apply this to a few different examples to illustrate its efficacy. Later, we will see a more detailed comparison of this method with finite-difference methods (section 6.6).

5.1 The Lagaris Method

We describe the method detailed in [6]. Suppose we have a differential equation of order m:

$$D\left(x, f(x), \nabla f(x), \nabla^2 f(x), \ldots, \nabla^m f(x)\right) = 0, \qquad x \in \text{dom}(f) \subset \mathbb{R}^n, \tag{18}$$

with some known boundary conditions, where $f : \text{dom}(f) \to \mathbb{R}$ is an unknown function. Note that this differential equation can be linear or non-linear.

Let $N(x;\theta) : \text{dom}(f) \to \mathbb{R}$ denote a neural network with parameters θ. Then we can construct a **trial solution** to our differential equation:

$$\hat{f}(x;\theta) = B(x) + F(x, N(x;\theta)), \tag{19}$$

where $B : \text{dom}(f) \to \mathbb{R}$ satisfies the boundary conditions (and is unchanged by the output of our neural network), and $F \equiv 0$ on the boundary of $\text{dom}(f)$. This construction guarantees that $\hat{f}$ always satisfies the boundary conditions of our differential equation, regardless of the output of $N(x;\theta)$. We will see later that such functions B and F can be constructed in an algorithmic way for a given type of boundary conditions.

Taking a training set of size M, $\{x_1, \ldots, x_M\} \subset \text{dom}(f)$, we have a cost function:

$$J(\theta) = \frac{1}{M} \sum_{j=1}^{M} D\left(x_j, \hat{f}(x_j;\theta), \nabla \hat{f}(x_j;\theta), \nabla^2 \hat{f}(x_j;\theta), \ldots, \nabla^m \hat{f}(x_j;\theta)\right)^2, \tag{20}$$

i.e. we insert our trial solution into equation (18), evaluate this at all of our training points, then square and average these values.

By minimising $J(\theta)$, we can minimise the absolute value of the differential equation at each of our training points x_j, and hence bring our trial solution $\hat{f}$ closer to approximating the true solution f. Notice the difference between this cost function and the one used in equation (14) for basic function approximation. Without having explicit data about the values f takes, we must use the indirect information provided by the differential equation. This can make convergence to a correct solution more difficult, as we will see later (section 5.4.1).

Observe also that to compute equation (20), we require the first m derivatives of our network's output with respect to x at each input x_j. For this reason, in our network architecture we should only use activation functions which are m-times differentiable. Functions like ReLU and Linear (figures 4 and 5) are also generally not suitable, since their second derivatives are identically equal to zero. The sigmoid function (figure 2) or tanh (figure 3) are typically used for the hidden layers, since they are smooth functions with non-zero derivative everywhere. However, we use the Linear activation function for the final layer, so as to not restrict the range of our output (since tanh and σ are bounded).

As mentioned in section 2.3, $N(x;\theta)$ is obtained only by the composition of functions. Hence it is easy to find an analytic expression for its derivative with respect to any input x, as is shown in [6]. However, in practice it is more efficient to make use of automatic differentiation to calculate these derivatives numerically.

Now, with a well-defined and calculable cost function J, we can carry out the optimisation method described in section 2.3 to train our network.

5.2 First-Order Ordinary Differential Equations

We begin with the most basic example, a first-order ordinary differential equation:

$$\frac{df(x)}{dx} = G(x, f(x)), \tag{21}$$

for $x \in [a, b]$, with Dirichlet initial conditions $f(a) = A \in \mathbb{R}$.

For a neural network $N(x; \theta) : [a, b] \to \mathbb{R}$, we have a general trial solution from [6]:

$$\hat{f}(x; \theta) = A + (x - a)N(x; \theta), \tag{22}$$

i.e. setting $B(x) \equiv A$ and $F(x, N(x; \theta)) = (x-a)N(x; \theta)$ in equation (19). Notice that $\hat{f}(a; \theta) = A$ as required. Taking M training data points $\{x_1, \ldots, x_M\} \subset [a, b]$, equation (20) gives cost function:

$$J(\theta) = \frac{1}{M} \sum_{j=1}^{M} \left(\frac{\partial \hat{f}(x_j; \theta)}{\partial x} - G\left(x_j, \hat{f}(x_j; \theta)\right) \right)^2, \tag{23}$$

where it is easy to see that:

$$\frac{\partial \hat{f}(x; \theta)}{\partial x} = N(x; \theta) + (x - a)\frac{\partial N(x; \theta)}{\partial x}. \tag{24}$$

Hence if we use automatic differentiation to calculate $\frac{\partial N(x;\theta)}{\partial x}$, the rest of the evaluation of J is simple.

5.2.1 Example 1

We consider the first example given in [6]. Here:

$$\frac{df(x)}{dx} = x^3 + 2x + x^2 \frac{1 + 3x^2}{1 + x + x^3} - \left(x + \frac{1 + 3x^2}{1 + x + x^3} \right) f(x), \tag{25}$$

for $x \in [0, 2]$, with $f(0) = 1$. By equation (22), we have trial solution $\hat{f}(x; \theta) = 1 + xN(x; \theta)$. This example has analytic equation:

$$f(x) = \frac{e^{-\frac{x^2}{2}}}{1 + x + x^3} + x^2, \tag{26}$$

with which we can compare our trial solution.

We take 20 evenly-spaced points across the interval $[0, 2]$ for our training data. This is done with another subclass of the Dataset class (figure C.4). We create the input data set in line 25:

```
self.dataIn  = torch.linspace(xRange[0], xRange[1], numSamples, requires_grad=True).view(-1,1)
# 'view' method reshapes tensors, in this case into a column vector
```

Note that the tensor has an argument called 'requires_grad' which we set to True. This can be done for any tensor in PyTorch, and means that the tensor's history will be tracked by the autograd package, i.e. a computation graph will be created with this tensor as an input variable, so that we can use automatic differentiation later to calculate gradients with respect to it.

We create a neural network identical to that used for function approximation in section 3.3 figure C.2), i.e. one input, one output, one hidden layer with ten hidden nodes, tanh activation function.

We then define a group of functions to evaluate $\hat{f}$, $\frac{d\hat{f}}{dx}$, the differential equation and f (using equations (22), (24), (25) and (26), shown in figure C.5). We define these functions using many scalar operations; when passing in a tensor as input, PyTorch automatically vectorises these functions and applies the operations component-wise.

Next, we define a function to train our neural network (figure C.6). We have imported the 'grad' function from the package 'torch.autograd'. We use it in line 213 to calculate $\frac{\partial N(x;\theta)}{\partial x}$:

```
# Get derivative of the network output w.r.t. the input values:
dndx = grad(n_out, batch, torch.ones_like(n_out), retain_graph=True)[0]
# torch.ones_like(x) creates a tensor the same shape as x, filled with 1's
```

Its first and second arguments are respectively the output variables, $N(x_1;\theta),\dots,N(x_m;\theta)$, and the input variables, $x_1,\dots,x_m$, with respect to which we calculate the gradients (where m is our batch size). The grad function actually computes the Jacobian-vector product seen in equation (13) at the end of section 2.5, hence the third argument specifies the vector, r, in this product [20]. Note however that this Jacobian will be a diagonal matrix, since all operations applied to our batch of inputs to obtain our outputs are componentwise, i.e. $N(x_j;\theta)$ depends only on x_j for $1 \leq j \leq m$. Thus we choose r to be a vector of ones, such that the output of grad will be:

$$\left(\frac{\partial N(x;\theta)}{\partial x}\bigg|_{x=x_1}, \dots, \frac{\partial N(x;\theta)}{\partial x}\bigg|_{x=x_m} \right)^{\mathbf{T}},$$

which is exactly what we need to evaluate $\frac{\partial \hat{f}}{\partial x}$ on our training data. Finally, in the fourth argument we set 'retain_graph' equal to True. Normally, after a computation graph has been traversed backwards to calculate gradients, autograd will delete the parts of the graph used, in order to free up memory [20]. However, this incomplete computation graph would prevent us from performing backpropagation, hence we set 'retain_graph' equal to True to stop the graph from being deleted.

Lastly, we take our loss function to be squared error loss and use the standard gradient descent algorithm with learning rate 10^{-3}. Once again, we use a DataLoader to load batches of our training data and specify our batch size. We use this example to compare the three types of gradient descent, i.e. to investigate the impact of our choice of batch size.

5.2.2 Effect of Batch Size

We wish to compare the speed and efficacy of each type of gradient descent (i.e. batch, mini-batch and stochastic). To do this, we fix a number of parameter updates, T, and take three identical neural networks (i.e. same structure and same initial parameter values). We train each network using a different type of gradient descent, and for a number of epochs such that the parameters of each network have been updated T times.

For a training set of size M, taking a batch size m which divides M means that all network parameters will be updated M/m times on each epoch of training. Thus to update the parameters T times we must train a network for mT/M epochs.

In our current example, $M = 20$. We take $T = 20,000$. Recall that for batch gradient descent, $m = M = 20$, hence we train for $T \cdot 1 = 20,000$ epochs. Meanwhile for stochastic gradient descent, $m = 1$, and so we train for $T/20 = 1,000$ epochs. Finally, for mini-batch gradient descent we take $m = 5$, and thus train for $T/4 = 5,000$ epochs. The code to run this is shown in figure C.7.

Figure 19 shows the network's output when trained using batch gradient descent to solve the example in 5.2.1 after T parameter updates, compared to the true solution given in equation (26). The outputs of the other two networks are not shown, since they are virtually identical. Meanwhile figures 20, 21 and 22 show each network's final cost value for each epoch of training.

All three networks converge to a good approximation of the solution function. However, it is interesting to compare their cost values over time. In batch gradient descent (figure 20), the cost is monotonically decreasing and tends towards a plateau. Meanwhile, in mini-batch gradient descent (figure 21), although the overall trend is the same, we see the cost decrease 'noisily'. Instead of levelling out completely, it oscillates around a plateau. The case of stochastic gradient descent (figure 22) is a more extreme example of this, with larger oscillations and an even less clear plateau.

This noisiness occurs since, when taking $m < M$, we are using a subset of our training data to approximate the gradients which depend on *all* of our training points. Hence our path through the error landscape becomes noisier (see figure 17) and our cost value can fluctuate up and down (since our m points are randomly selected each time). The smaller m is, the less representative this approximation is of the true gradients; this is why we see more noise in the stochastic case

than in the mini-batch case. When $m = M$ in the batch case, we use the true gradient values, and hence our path is smooth and deterministic.

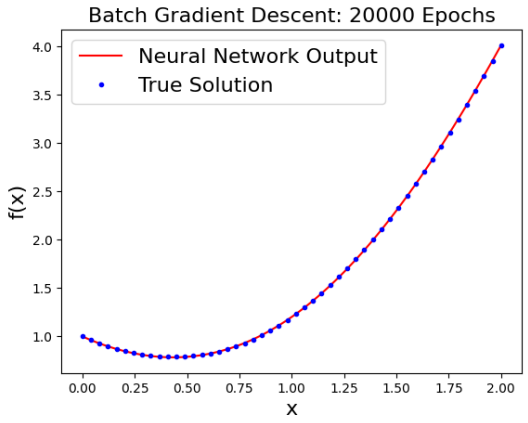

Figure 19: Output of neural network trained to solve equation (25) for $20,000$ parameter updates using **batch** gradient descent, compared to true solution.

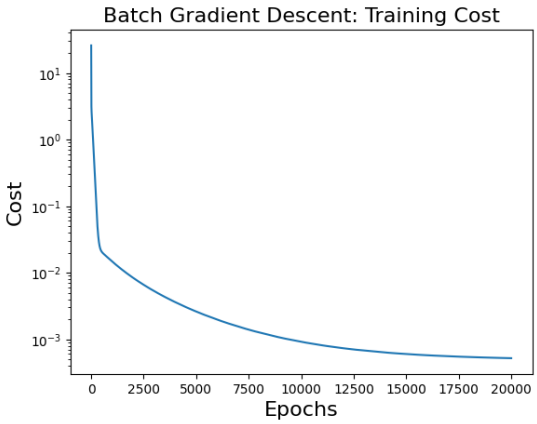

Figure 20: Cost value of each training epoch, for neural network trained using **batch** gradient descent.

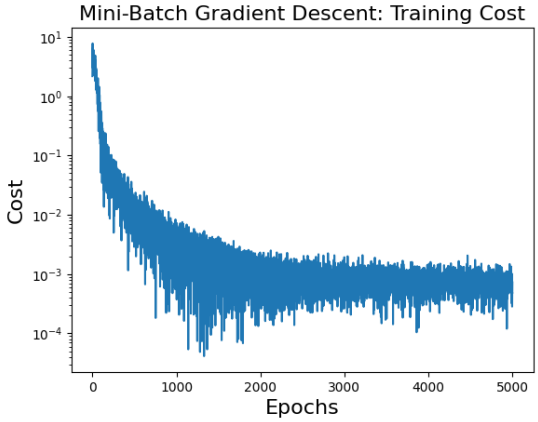

Figure 21: Cost value of the final batch of each training epoch, for neural network trained using **mini-batch** gradient descent.

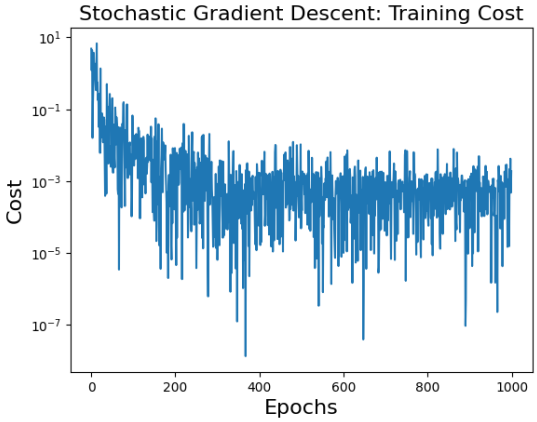

Figure 22: Cost value of the final batch of each training epoch, for neural network trained using **stochastic** gradient descent.

It is also interesting to compare how long each method took to carry out this training, and how the resulting networks compare when tested on 20 other points sampled uniformly from the function's domain. Table 2 shows these results.

	Batch	Mini-batch	Stochastic
Training Time (s)	26.03	20.02	16.90
Test Error	4.26×10^{-4}	5.79×10^{-4}	1.40×10^{-3}

Table 2: Comparison of three types of gradient descent used to train a network to solve equation (25).

As expected, a larger batch size leads to a longer run time, since more computations have to be carried out. However, larger batch sizes give a smaller error. This occurs since every parameter update optimises the network solution for m data points; a larger m means the solution is more likely to generalise better to arbitrary points in the interval.

Note that we have used this example simply to illustrate the differences between the three methods. Since M and our input domain $[0, 2]$ are small, the differences in training time and test error are also small. In practice, it is when M and our space of inputs are large that the results of the three methods vary more. This occurs since for large M the computational cost of batch gradient descent becomes an issue and, conversely, generalising to a large input domain becomes

more difficult if only optimising for a small number, m, of sample points on any given parameter update.

5.2.3 Example 2

We consider another example of the form shown in equation (21), the second example given in [6]:

$$\frac{df(x)}{dx} = e^{-\frac{x}{5}}\cos(x) - \frac{1}{5}f(x), \tag{27}$$

for $x \in [0, 10]$, with $f(0) = 0$. Equation (22) gives trial solution $\hat{f}(x;\theta) = xN(x;\theta)$. In this case, the analytic solution is:

$$f(x) = e^{-\frac{x}{5}}\sin(x). \tag{28}$$

Since this differential equation is of the same type as the previous example, we can use the same code to approximate a solution, changing only the functions which define the trial solution, the differential equation etc. (figure C.5). The appropriate functions for this problem are shown in figure C.8. Since the input range is larger than the previous example, we take more training data points ($M = 50$).

We use this example to compare the different optimisation algorithms described in section 4.

5.2.4 Comparison of Optimisation Algorithms

We have five algorithms that we wish to compare: gradient descent, gradient descent with momentum, RProp, RMSProp and Adam (algorithms 1, 2, 4, 5 and 6 respectively). Since RProp is only suited to batch training, we take our batch size to be $m = M = 50$ for all five algorithms. We take a learning rate of 10^{-3} for all of them (noting that for RProp this is only the *initial* learning rate). For all other hyperparameters, we use the PyTorch default values, since these are intended to be the values most suited to a variety of tasks [20]. As before, we take five networks of identical structure and initial parameter values. The code to carry this out is shown in figure C.9.

Figures 23 to 27 below show the outputs of each network after 20,000 epochs of training, while figure 28 shows their training cost values over time. Meanwhile table 3 shows how long this training took for each network, as well as the networks' error when tested on 50 points uniformly sampled from the interval $[0, 10]$.

The standard gradient descent algorithm (figure 23) has attained the worst solution after 20,000 epochs (with highest training error, test error, and worst graphical approximation of the solution function). Meanwhile, the addition of momentum (figure 24) has made little improvement on this. This can potentially be attributed to a number of factors. Firstly, the implementation of gradient descent with momentum in PyTorch does not include bias correction [20] (see section 4.2.1), and hence can slow down learning during early epochs. Secondly, recall that the benefits of momentum are seen when our error surface has regions of small gradients or noisy gradients. If our gradients are relatively constant, momentum does little to change the original gradient descent algorithm. This may be the case for the early epochs of training in this example.

RProp (figure 25) and RMSProp (figure 26) have provided significantly better approximations. This can be attributed to their parameter-specific learning rates accelerating the convergence process. It is interesting to note the 'bumps' in the cost graph of RProp (green in figure 25): these can most likely be attributed to the final issue described in section 4.3. This issue arises due to parameter updates in RProp ignoring the *magnitude* of the gradients, and so continuing to move in the 'wrong' direction for several epochs before their learning rates have been sufficiently amended.

Adam (figure 27) has clearly provided the best solution after 20,000 epochs. It has attained the smallest test and training errors, as well as a near-perfect graphical approximation of the solution function.

Considering their total training times, gradient descent was the quickest to complete 20,000 epochs, with the addition of momentum only slowing it down slightly. The times taken for the other three optimisation algorithms were quite similar to each other, and roughly ten seconds longer than the two versions of gradient descent. This can be attributed to the larger number of

calculations carried out per parameter update in these three algorithms. Notice, however, that gradient descent (with or without momentum) would take considerably more epochs to achieve a similar cost value to those attained by RProp, RMSProp or Adam, and thus would require much more training time.

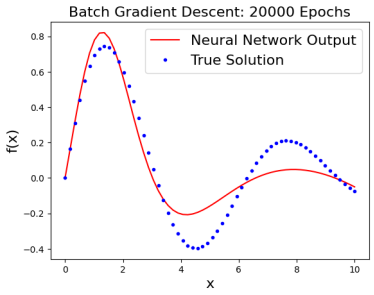

Figure 23: Output of a neural network trained with **gradient descent** to solve equation (27) for 20,000 epochs, compared to the true solution.

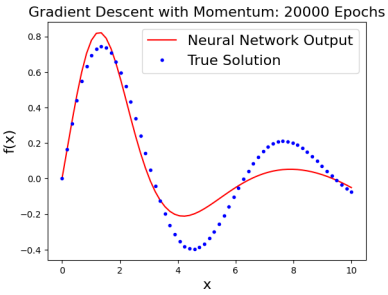

Figure 24: Output of a neural network trained with **gradient descent with momentum** to solve equation (27) for 20,000 epochs, compared to the true solution.

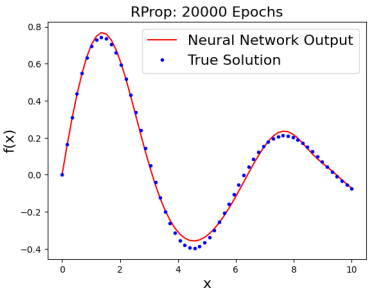

Figure 25: Output of a neural network trained with **RProp** to solve equation (27) for 20,000 epochs, compared to the true solution.

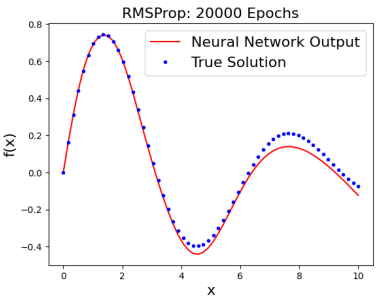

Figure 26: Output of a neural network trained with **RMSProp** to solve equation (27) for 20,000 epochs, compared to the true solution.

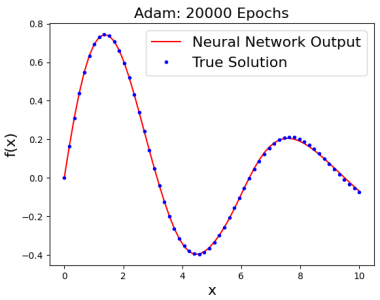

Figure 27: Output of a neural network trained with **Adam** to solve equation (27) for 20,000 epochs, compared to the true solution.

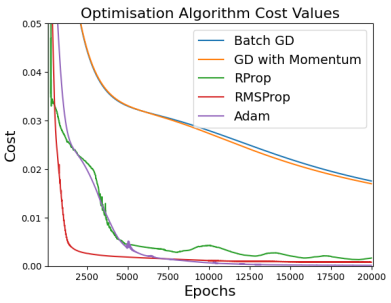

Figure 28: Training cost values for each of the optimisation algorithm. The graph's key is ordered from highest final cost to lowest.

	Gradient Descent	GD with Momentum	RProp	RMSProp	Adam
Training Time (s)	67.43	71.10	77.33	80.10	79.10
Test Error	1.76×10^{-2}	1.70×10^{-2}	1.65×10^{-3}	7.99×10^{-4}	1.58×10^{-4}

Table 3: Comparison of five optimisation algorithms used to train a network to solve equation (27).

Hence we have seen the key benefit of these more complicated optimisation algorithms: their capacity to accelerate convergence.

It is also interesting to observe the outcome of using RProp with mini-batches. We take batch size $m = 10$, and train another network identical to the previous five for $4,000$ epochs (since this will perform $20,000$ parameter updates, by the calculation in section 5.2.2).

As shown in figure 29, we obtain a very poor solution by this method; in fact this solution has a test error of 5.26×10^{-2}. Its cost graph (figure 30) illustrates the erratic parameter updates caused by using RProp with mini-batches described in section 4.3. They have resulted in a 'noisy plateau' around this poor solution, with no indication even of a general downward trend in the cost value.

Due to its superior performance in the above example, and its ability to perform training with any batch size, we will continue to use Adam in future examples.

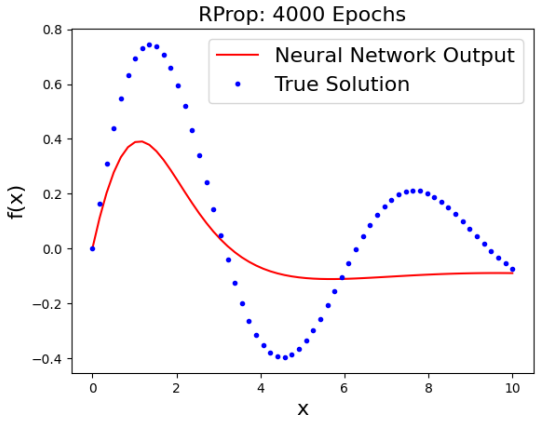

Figure 29: Output of neural network trained to solve equation (27) for 20,000 parameter updates using **RProp with mini-batches**, compared to true solution.

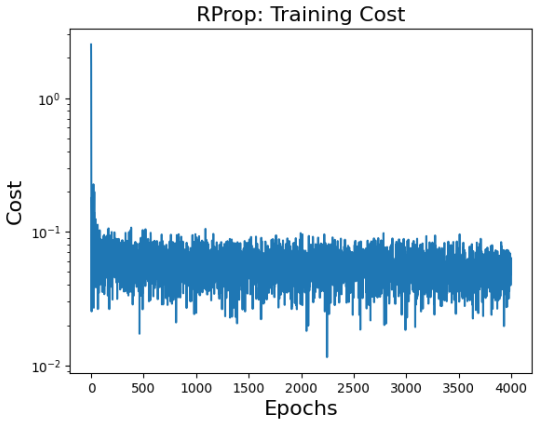

Figure 30: Cost value of the final batch of each training epoch, for neural network trained using **RProp with mini-batches**.

5.3 Second-Order Ordinary Differential Equations

We now consider a second-order ordinary differential equation:

$$\frac{d^2 f(x)}{dx^2} = G\left(x, f(x), \frac{df(x)}{dx}\right), \tag{29}$$

for $x \in [a, b]$. Here we can take Dirichlet boundary conditions $f(a) = A \in \mathbb{R}$, $f(b) = B \in \mathbb{R}$, for which we have trial solution:

$$\hat{f}(x;\theta) = A\frac{(b-x)}{b-a} + B\frac{(x-a)}{b-a} + (x-a)(b-x)N(x;\theta). \tag{30}$$

We can also consider Cauchy boundary conditions $f(a) = A \in \mathbb{R}$, $f'(a) = A' \in \mathbb{R}$. Here we instead have trial solution:

$$\hat{f}(x;\theta) = A + A'(x-a) + (x-a)^2 N(x;\theta). \tag{31}$$

In this second case, we can see that:

$$\frac{\partial \hat{f}(x;\theta)}{\partial x} = A' + (x-a)\left[2N(x;\theta) + (x-a)\frac{\partial N(x;\theta)}{\partial x}\right], \tag{32}$$

$$\frac{\partial^2 \hat{f}(x;\theta)}{dx^2} = 2N(x;\theta) + 4(x-a)\frac{\partial N(x;\theta)}{dx} + (x-a)^2\frac{\partial^2 N(x;\theta)}{dx^2}. \tag{33}$$

The derivatives of equation (30) can be expressed similarly. Taking $\{x_1, \ldots, x_M\} \subset [a, b]$ as our training data, we have cost function:

$$J(\theta) = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{\partial^2 \hat{f}(x_j;\theta)}{\partial x^2} - G\left(x_j, \hat{f}(x_j;\theta), \frac{\partial \hat{f}(x_j;\theta)}{\partial x}\right)\right)^2. \tag{34}$$

We will see shortly that it is possible to calculate second-order derivatives (in fact, derivatives of any order) of $N(x;\theta)$ using automatic differentiation, and so it is easy to calculate J using equations (31), (32) and (33).

5.3.1 Example 3

We take the third example from [6]:

$$\frac{d^2 f(x)}{dx^2} = -\frac{1}{5}\frac{df(x)}{dx} - f(x) - \frac{1}{5}e^{-\frac{x}{5}}\cos(x), \tag{35}$$

for $x \in [0, 10]$ with Cauchy boundary conditions $f(0) = 0$, $f'(0) = 1$. Thus equation (31) gives a trial solution $\hat{f}(x;\theta) = x + x^2 N(x;\theta)$. Here, the exact solution is actually the same as that the previous example, given in equation (28).

We use the same DataSet class and Module class used in previous examples (figures C.4 and C.2 respectively). Functions to compute the trial function, its derivatives and the cost function are shown in figure C.10. Our function to train the network is shown in figure C.11. In line 106, we use the 'grad' function to calculate the derivative of $N(x;\theta)$ with respect to x, as before:

```
# Get first derivative of the network output with respect to the input values:
dndx    =  grad(n_out, batch, torch.ones_like(n_out), retain_graph=True, create_graph=True)[0]
# Get second derivative of the network output with respect to the input values:
d2ndx2  =  grad(dndx, batch, torch.ones_like(dndx), retain_graph=True)[0]
```

Here we also set a fifth argument 'create_graph' to True. This means that, in the process of traversing the computation graph backwards to calculate this first derivative, autograd will create *another* computation graph, for the output of the first graph (i.e. $\frac{\partial N(x;\theta)}{\partial x}$) with respect to its input (i.e. x). This process allows us to calculate the second derivative of $N(x;\theta)$ with respect to x (as we do two lines later), and hence to iteratively compute the derivative of any order.

We use the Adam optimiser, and train with batch size $m = M = 50$. Figure 31 shows the output of the network after 20,000 epochs of training, while figure 32 shows its cost values during training. It has a test cost of 6.43×10^{-5}, while the mean-squared difference between the trial function and the true solution at these test points (i.e., the **solution inaccuracy**) is 2.10×10^{-6}.

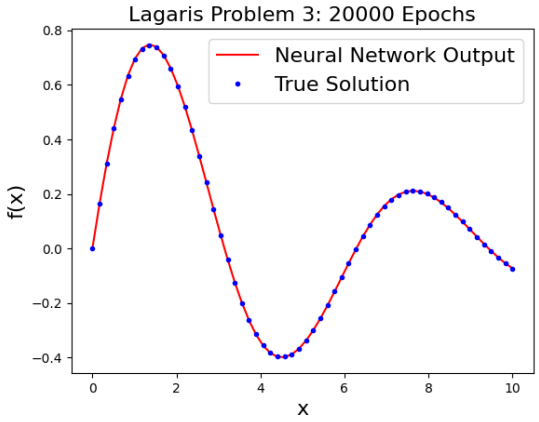

Figure 31: Output of neural network trained to solve equation (35) on $[0, 10]$ for $20,000$ epochs, compared to true solution.

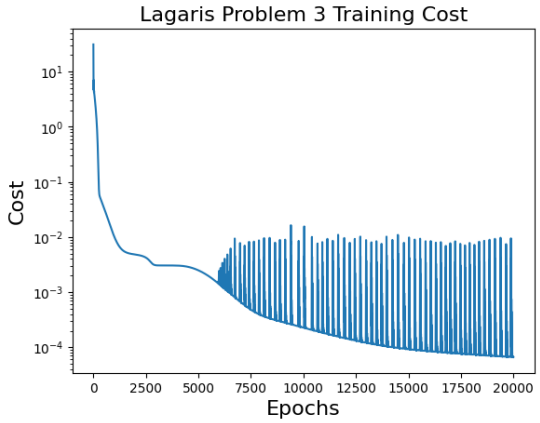

Figure 32: Cost values of network trained to solve equation (35) on $[0, 10]$ for $20,000$ epochs.

The spikes in the cost graph later in training indicate that the learning rate is too large for this region of the error surface. Although the cost is still descending overall, many of our parameter updates are taking us *over* the minimum to a point of higher cost, since our step size is too large. It is often necessary to have a larger learning rate earlier in training to avoid local minima, but this same learning rate can prevent us from getting as close as possible to the minimum. A common solution is an **adaptive learning rate**, which decreases depending simply on the number of the epoch, or according to some other metric. We will explore this in more detail in section 6.3.1.

5.3.2 Extrapolation

For now, it is interesting to consider how well our solution extrapolates beyond the training range $[0, 10]$, given that the true solution is well-defined on all of $\mathbb{R}$. Figures 33 and 34 show the output of our neural network in ranges of width 5 either side of our training interval.

In both cases, the approximation is poor. To the left, we have an error of 18.9 from our cost function, while the solution inaccuracy is 25.3. Meanwhile, to the right these two errors are 0.763 and 0.436 respectively. Clearly, solutions found by this method do not extrapolate well beyond their training domain.

However, the network we have trained on $[0, 10]$ can be further trained on the widened interval $[-5, 15]$. After another 20,000 epochs of training on 100 evenly-space points in $[-5, 15]$, the network's output shown in figure 35. Figure 36 shows the network's training cost for each epoch. Notice the large increase in cost when we begin to train on the larger interval, which decreases with more epochs of training.

This has a final test cost (evaluated on 100 uniformly-sampled points across the interval $[-5, 15]$) of 2.46×10^{-4} and solution inaccuracy 4.97×10^{-5}. Notice that both of these final error values are greater than those which we observed after training on $[0, 10]$. This can be expected, since training a network to approximate a function over an interval of twice the size is a more complicated problem, and hence more epochs would be required to lower the network's cost to the same level.

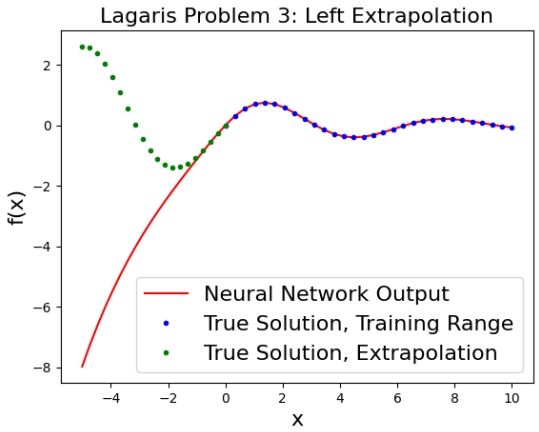

Figure 33: Output of neural network on range $[-5, 10]$, having been trained to solve equation (35) on $[0, 10]$, compared to true solution.

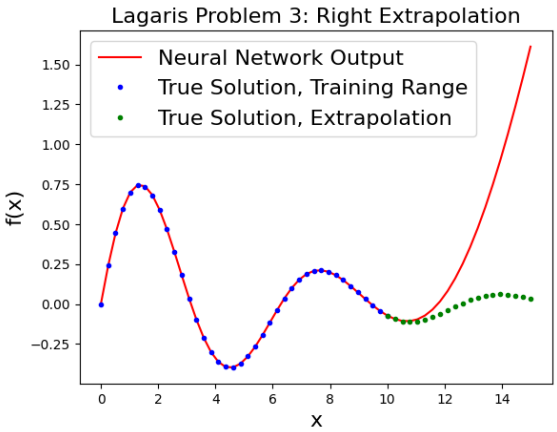

Figure 34: Output of neural network on range $[0, 15]$, having been trained to solve equation (35) on $[0, 10]$, compared to true solution.

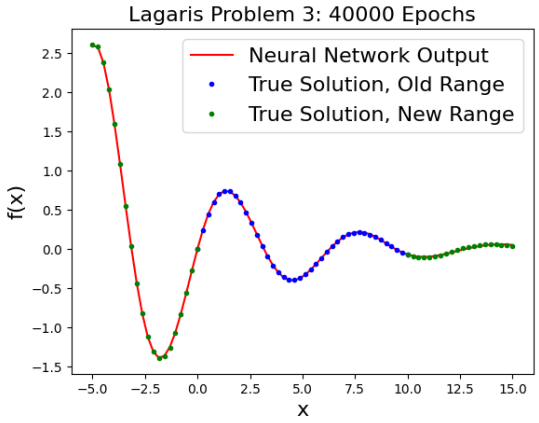

Figure 35: Output of neural network having been trained to solve equation (35) first on $[0, 10]$ and then on $[-5, 15]$, compared to true solution.

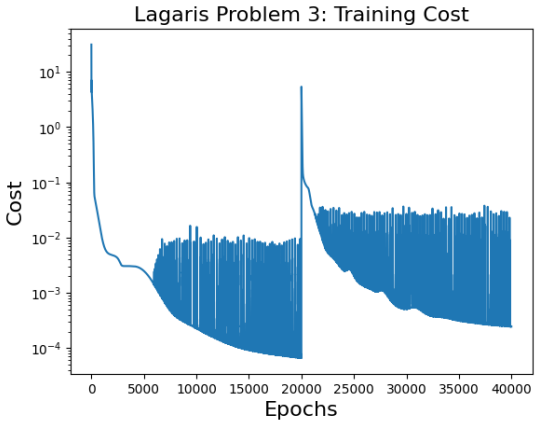

Figure 36: Cost values of network trained to solve equation (35) on $[0, 10]$ for 20,000 epochs and then on $[-5, 15]$ for a further 20,000.

Hence we have seen that we can widen our initial training domain to solve a given differential equation on a larger domain. A natural question to ask is does this produce a *better* or *worse* solution than training on the larger domain from the start?

Figure 37 shows the output of a network initially identical to our previous network, but this time trained to solve equation (35) on 100 evenly-spaced training points in $[-5, 15]$ for 40,000 epochs. Figure 38 shows its cost values during training. Meanwhile table 4 shows a comparison of the performance of this network compared to our previous network.

Training on $[0, 10]$ and then $[-5, 15]$ (network 1) has produced a better solution in a shorter amount of training time and, more importantly, in the same number of parameter updates. The

	Training Time (s)	Test Error	Solution Inaccuracy
Network 1	154.86	2.46×10^{-4}	4.97×10^{-5}
Network 2	190.89	2.34×10^{-3}	2.06×10^{-3}

Table 4: Comparison of two identical networks trained to solve equation (35). Network 1 was trained on [0,10] for 20,000 epochs, and then on [-5,15] for a further 20,000. Network 2 was trained only on [-5,15] for 40,000 epochs.

reduced training time is a result of the fact that for the first 20,000 epochs network 1 has only 50 training points, while network 2 always has 100. This means that network 1 carries out fewer computations for the first 20,000 epochs, and hence these epochs take less time. The code to carry out this training for both networks is shown in figure C.12.

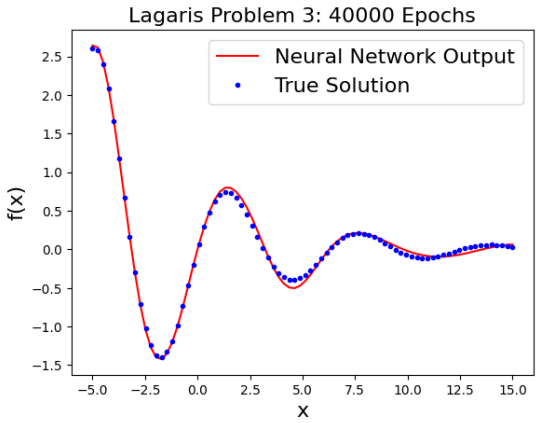

Figure 37: Output of neural network trained to solve equation (35) on $[-5, 15]$ for 40,000 epochs, compared to true solution.

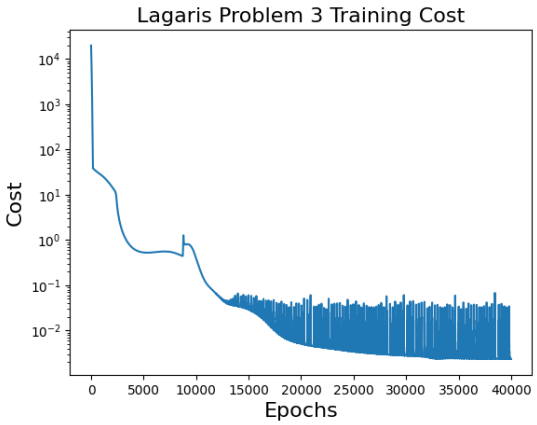

Figure 38: Cost values of network trained to solve equation (35) on $[-5, 15]$ for 40,000 epochs.

We will explore more in the next example why the method by which we have trained network 1 (known as **curriculum learning**) has produced a better solution.

5.4 Systems of Ordinary Differential Equations

We can also consider a system of K first-order differential equations:

$$\frac{df_i(x)}{dx} = G_i\left(x, f_1(x), \ldots, f_K(x)\right), \tag{36}$$

for $x \in [a, b]$, with Dirichlet boundary conditions $f_i(a) = A_i \in \mathbb{R}$ for $i = 1, \ldots, K$.

We can take a neural network $N(x; \theta) : [a, b] \to \mathbb{R}^K$, and create trial solutions for each $i = 1, \ldots, K$ [6]:

$$\hat{f}_i(x; \theta) = A_i + (x - a)N_i(x; \theta), \tag{37}$$

where $N_i(x; \theta)$ denotes the i^{th} output of $N(x; \theta)$. Choosing M training points $x_1, \ldots, x_M \in [a, b]$, our cost function then becomes the sum of K cost functions defined in equation (23), i.e.:

$$J(\theta) = \frac{1}{M} \sum_{i=1}^{K} \sum_{j=1}^{M} \left(\frac{\partial \hat{f}_i(x_j; \theta)}{\partial x} - G_i\left(x_j, \hat{f}_1(x_j; \theta), \ldots, \hat{f}_K(x_j; \theta)\right) \right)^2, \tag{38}$$

where the derivative of each $\hat{f}_i$ with respect to x is analogous to equation (24).

Note that in [6], the general solution they propose uses K neural networks $N_i(x; \theta) : [a, b] \to \mathbb{R}$, $i = 1, \ldots, K$, instead of one network $N(x; \theta) : [a, b] \to \mathbb{R}^K$. When implementing the example below, both methods were found to give equivalent results (as long as the total number of parameters remained constant), and so we choose to use just one network, as this makes the code more concise.

5.4.1 Example 4

To demonstrate this, we consider the fourth example given in [6]. Here we have a system of coupled ODEs:

$$f_1'(x) = \cos(x) + f_1(x)^2 + f_2(x) - 1 - x^2 - \sin^2(x), \tag{39}$$

$$f_2'(x) = 2x - (1 + x^2)\sin(x) + f_1(x)f_2(x), \tag{40}$$

where $x \in [0, 3]$, $f_1(0) = 0$ and $f_2(0) = 1$. Taking a neural network $N(x;\theta) : [0, 3] \to \mathbb{R}^2$, $x \mapsto (N_1(x;\theta), N_2(x;\theta))$, our trial solutions are:

$$\hat{f}_1(x;\theta) = xN_1(x;\theta), \qquad \hat{f}_2(x;\theta) = 1 + xN_2(x;\theta). \tag{41}$$

In this case, the exact solutions are $f_1(x) = \sin(x)$, $f_2(x) = 1 + x^2$.

We thus take a neural network identical to those used previously (figure C.2), but with two outputs. Since this is a more complicated problem than previous examples, we increase the number of hidden nodes to 16. Our Dataset object is identical to previous examples. Figure C.13 shows the code used to train this neural network.

Note that the network's output is of shape $(m,2)$ this time (where m is the batch size), i.e. a tensor with one column whose entries are $N_1(x_j;\theta)$, and a second column whose entries are $N_2(x_j;\theta)$ for $j = 1, \ldots, m$. To evaluate the trial solution and the two differential equations, we must separate these two column vectors in some way. This must be done carefully, however, in order to preserve the computation graphs needed to calculate their derivatives.

Tensors can be sliced much in the same way as Python lists or NumPy arrays, but note that slicing actually creates a *new* tensor entirely, whose values are the same as the original tensor. This does *not* retain the gradient history of the original tensor, i.e. the new tensor has a new computation graph unrelated to the previous one. To separate the tensor's columns and retain their computation graphs, we must use the PyTorch 'split' function:

```
# Separate two columns of output (one for f_1, one for f_2)
# Using torch.split retains tensor history for autograd
n1_out, n2_out = torch.split(n_out, split_size_or_sections = 1, dim = 1)
```

Passing the network output in as its first argument, the third argument determines which axis we want to separate the tensor along (1 in our case will split the tensor into columns), and the second argument determines into how many sections each column is split (we set this to 1 as well, so that the i^{th} column contains all values $N_i(x_j;\theta)$, $1 \leq j \leq m$, for $i = 1, 2$).

5.4.2 Curriculum Learning

The interesting feature of this example is that, using the standard network architecture and training method we have used in our previous examples, the network failed to converge consistently to the correct solution. In fact, it would converge to several different solutions (depending on the initial weight values) one of which was the correct solution, and the rest of which were incorrect (for example, figure 39 below). The final training cost values were comparable, however, whether the correct solution was attained or not (e.g. in figure 40, the final cost is 2.58×10^{-5}). Varying the hyperparameters we have discussed up until this point (i.e. network structure, activation functions, weight initialisation method, learning rate, batch size, optimisation algorithm) produced similar results.

Although it is difficult to obtain concrete information regarding the shape of this high-dimensional error surface, we can potentially assume that the surface contains one or more local minima that are difficult to avoid using gradient-based optimisation.

What guaranteed convergence to the correct solution in this example was **curriculum learning**. This is inspired by the idea of a classroom teaching curriculum, in which the concepts to be taught are ordered from simplest to most complicated, and then taught in this order. To apply this to neural networks, curriculum learning involves ranking a network's training data from 'easiest' to 'hardest', then training a network on data samples of increasing difficulty. Although this gives an intuitive understanding, there is also analytic evidence that curriculum learning actually *alters* the error landscape, making it steeper while retaining the same global minimum [26].

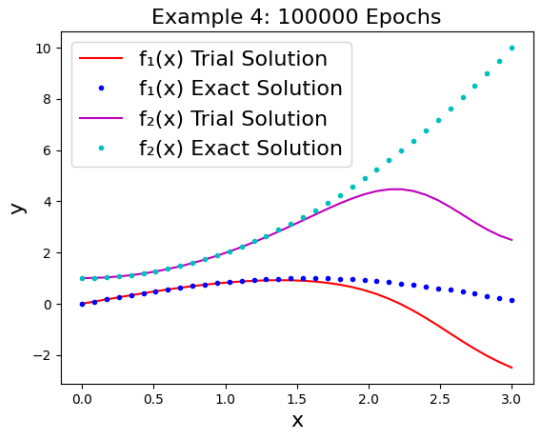

Figure 39: Output of neural network trained to solve equations (39) and (40) on $[0, 3]$ for 100,000 epochs, compared to true solution.

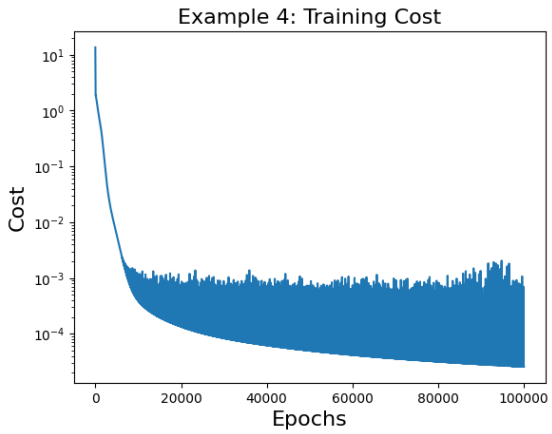

Figure 40: Cost values of network trained to solve equations (39) and (40) on $[0, 3]$ for 100,000 epochs.

As mentioned in the previous example, curriculum learning for our application can be implemented by dividing our training interval into sub-intervals, and training the network on these individual sub-intervals. We argue that points closer to $a = 0$, where our boundary conditions are imposed, can be considered 'easier' training points, since at $x = 0$ our error function is always equal to zero. The error function is continuous by construction, and hence points close to 0 will have a small error. Thus we can argue that optimising our network at these points will be easier, since the error is already closer to being minimised. Notice also in figure 39, the solution's inaccuracy increases monotonically as x increases.

Figures 41 and 42 below show the output and training costs of our network trained on the intervals $[0, 1]$, $[0, 2]$ and $[0, 3]$ successively for $12,000$ epochs each. Notice, as in the last example, the large spikes in cost value when we increase our interval's width (and thus introduce the network to unseen data). In far fewer epochs, we have converged to the correct solution, and achieved a final cost value of 1.48×10^{-5}.

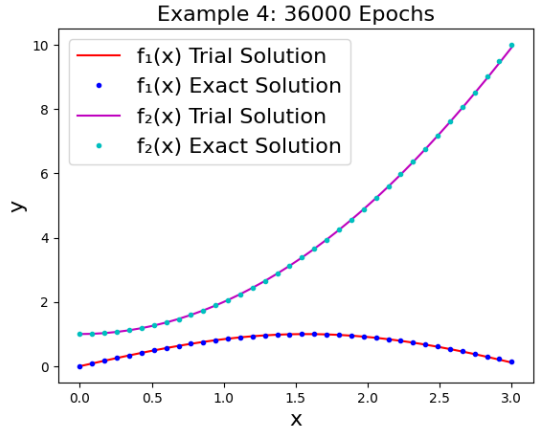

Figure 41: Output of neural network trained with curriculum learning to solve equations (39) and (40) on $[0, 3]$ for 36,000 epochs, compared to true solution.

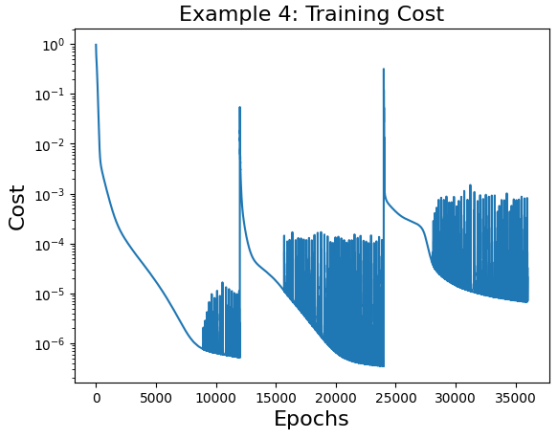

Figure 42: Cost values of network trained with curriculum learning to solve equations (39) and (40) on $[0, 3]$ for 36,000 epochs

This example highlights the caveat we made about the Universal Approximation Theorem in section 3.1. Although it is guaranteed that we *can* find a neural network to approximate the solutions of these two functions, it is left to us to find the *optimal* network architecture and training methodology to achieve convergence.

This example might present a drawback of the Lagaris method: without knowing our true solutions a priori, how could we know that the solutions obtained in figure 39 were not correct?

Given that traditional numerical methods for solving differential equations do not require the same experimentation, why should the Lagaris method be preferred?

Regarding the first question, since the network was converging to several different solutions depending on the initial parameter values, it was clear that some must be incorrect and our network was not sufficient, at least in this example. The second question remains a valid issue, and we will address it further in section 6.6.

Returning to curriculum learning, is important to note that the literature is divided on its merits and implementation. Although it has been shown to improve solutions in some cases [26], other authors have demonstrated otherwise. In [27], for example, they show that in some use-cases it offers little to no improvement, while in others the same benefits can be attained simply by dynamically altering the training set size (i.e. curriculum learning without ranking data from easiest to hardest).

To investigate this in our example, consider table 5 below. It compares various possible curricula for our network, all trained for a total of $36,000$ epochs. The cost and solution inaccuracy are tested on one set of 30 points uniformly sampled across $[0, 3]$.

In each curriculum, the network is trained for an equal number of epochs on each sub-interval. Some train on sub-intervals of increasing size, starting from 0 (rows 1-3). Others train on non-intersecting intervals whose union is the full interval, beginning with an interval which starts at 0 (rows 4-5). Others do the same as this, while finally training on the full interval [0,3] (rows 6-7). The remaining rows repeat these three methods, but starting from 3 rather than 0. The code to carry out training on each of these curricula is shown in figure C.14.

Row	Curriculum	Solution	Test Error	Solution Inaccuracy
1	[0,3]	Incorrect	8.42×10^{-5}	4.97
2	[0,1.5], [0,3]	Correct	7.00×10^{-3}	3.14×10^{-4}
3	[0,1], [0,2], [0,3]	Correct	1.48×10^{-5}	2.63×10^{-4}
4	[0,1.5], [1.5,3]	Correct	6.24×10^{-3}	1.85×10^{-3}
5	[0,1], [1,2], [2,3]	Correct	5.30×10^{-3}	1.62×10^{-3}
6	[0,1.5], [1.5,3], [0,3]	Correct	2.21×10^{-5}	1.84×10^{-4}
7	[0,1], [1,2], [2,3], [0,3]	Correct	7.55×10^{-6}	1.69×10^{-4}
8	[1.5,3], [0,3]	Incorrect	1.93×10^{-4}	5.80
9	[2,3], [1,3], [0,3]	Incorrect	2.58×10^{-4}	6.35
10	[2.25,3], [1.5,3], [0.75,3], [0,3]	Incorrect	7.86×10^{-4}	7.75
11	[1.5,3], [0,1.5], [0,3]	Correct	6.55×10^{-4}	4.39×10^{-4}
12	[2,3], [1,2], [0,1], [0,3]	Correct	1.82×10^{-5}	7.85×10^{-4}
13	[1.5,3], [0,1.5]	Correct	7.21×10^{-3}	1.62×10^{-3}
14	[2,3], [1,2], [0,1]	Correct	1.38×10^{-1}	6.78×10^{-3}

Table 5: Comparison of identical networks trained for $36,000$ epochs to solve equations (39) and (40) with different curricula.

Rows 2-7 show that training close to zero first leads to convergence to the correct solution, with better solutions observed when we end by training on the whole input range $[0, 3]$ (rows 2, 3, 6, and 7) and train on more sub-intervals (rows 3 and 7).

Interestingly, comparing rows 8-10 with rows 11-14 show us that we can achieve convergence to the correct solution even if we begin by training close to 3, but only if *at some point* we train the network *only* on points close to 0. We also see an improved performance here when we end by training on the whole input range $[0, 3]$ (rows 11 and 12).

We can conclude that, in this case, training on our 'easier' data samples in isolation can speed up convergence to a correct solution. Although doing so later in training will still cause us to converge to the correct solution, a better solution can be obtained by doing so at the beginning. Hence this example resembles more closely those considered in [26] rather than in [27]. Finally, we can conclude that ending with training on the full input interval gives a better overall solution,

since we avoid overfitting our model to a particular sub-interval.

5.5 Partial Differential Equations

The Lagaris method can also be applied to problems in higher dimensions, i.e. to partial differential equations whose solutions are functions in n variables. We demonstrate with two examples in two dimensions, but note that it is not difficult to generalise to higher dimensions.

We consider the two-dimensional Poisson equation, a fundamental PDE used to model the process of diffusion and arising in many contexts (e.g. fluid dynamics, heat transfer) [28]:

$$\Delta f(x,y) = \frac{\partial^2}{\partial x^2} f(x,y) + \frac{\partial^2}{\partial y^2} f(x,y) = G(x,y), \tag{42}$$

for $x \in [a,b]$, $y \in [c,d]$. We examine first a case with Dirichlet boundary conditions $f(a,y) = A(y)$, $f(b,y) = B(y)$, $f(x,c) = C(x)$ and $f(x,d) = D(x)$ for some functions $A, B : [c,d] \to \mathbb{R}$, $C, D : [a,b] \to \mathbb{R}$. Taking a neural network $N(x,y;\theta) : \mathbb{R}^2 \to \mathbb{R}$, we have a trial solution:

$$\hat{f}(x,y;\theta) = E(x,y) + (x-a)(b-x)(y-c)(d-y)N(x,y;\theta), \tag{43}$$

where $E(x,y)$ is constructed like so:

$$\begin{aligned} E(x,y) = \frac{b-x}{b-a}A(y) + \frac{x-a}{b-a}B(y) + \frac{d-y}{d-c}\left\{C(x) - \left[\frac{b-x}{b-a}C(a) + \frac{x-a}{b-a}C(b)\right]\right\} \\ + \frac{y-c}{d-c}\left\{D(x) - \left[\frac{b-x}{b-a}D(a) + \frac{x-a}{b-a}D(b)\right]\right\}, \end{aligned} \tag{44}$$

noting that for $f(x,y)$ to be well-defined we must have $A(c) = C(a)$, $A(d) = D(a)$, $B(c) = C(b)$ and $B(d) = D(b)$, and hence our construction of $E(x,y)$ satisfies these boundary conditions.

We may also consider the Poisson equation with mixed boundary conditions, i.e. Dirichlet on some parts of the boundary, and Neumann on others. For example, we could have $f(a,y) = A(y)$, $f(b,y) = B(y)$, $f(x,c) = C(x)$ and $\frac{\partial}{\partial y} f(x,d) = \tilde{D}(x)$ for some functions $A, B : [c,d] \to \mathbb{R}$, $C, \tilde{D} : [a,b] \to \mathbb{R}$.

In this scenario we can construct a trial solution:

$$\hat{f}(x,y;\theta) = F(x,y) + (x-a)(b-x)(y-c)\left[N(x,y;\theta) - N(x,d;\theta) - (d-c)\frac{\partial}{\partial y}N(x,d;\theta)\right], \tag{45}$$

where $F(x,y)$ is given by:

$$\begin{aligned} F(x,y) = \frac{b-x}{b-a}A(y) + \frac{x-a}{b-a}B(y) + C(x) - \left[\frac{b-x}{b-a}C(a) + \frac{x-a}{b-a}C(b)\right] \\ + \frac{(y-c)^2}{2(d-c)}\left\{\tilde{D}(x) - \left[\frac{b-x}{b-a}\tilde{D}(a) + \frac{x-a}{b-a}\tilde{D}(b)\right]\right\}. \end{aligned} \tag{46}$$

As in the previous case, it is trivial to see that $F(x,y)$ (and thus $\hat{f}(x,y;\theta)$) satisfies the first three boundary conditions (noting again that $A(c) = C(a)$, $B(c) = C(b)$). For the final condition, we have:

$$\begin{aligned} \frac{\partial}{\partial y}\hat{f}(x,y;\theta) = \frac{b-x}{b-a}A'(y) + \frac{x-a}{b-a}B'(y) + \frac{y-c}{d-c}\left\{\tilde{D}(x) - \left[\frac{b-x}{b-a}\tilde{D}(a) + \frac{x-a}{b-a}\tilde{D}(b)\right]\right\} \\ + (x-a)(b-x)\left[N(x,y;\theta) - N(x,d;\theta) - (d-c)\frac{\partial}{\partial y}N(x,d;\theta)\right. \\ \left. + (y-c)\frac{\partial}{\partial y}N(x,y;\theta)\right], \end{aligned}$$

which satisfies the fourth boundary condition when $y = d$, noting again that for $f(x,y)$ to be well defined we must have $A'(d) = \tilde{D}(a)$ and $B'(d) = \tilde{D}(b)$.

For both kinds of boundary conditions, choosing M training points $(x_1, y_1), \ldots, (x_M, y_M) \in [a, b] \times [c, d]$, we have cost function:

$$J(\theta) = \sum_{j=1}^{M} \left(\frac{\partial^2}{\partial x^2} \hat{f}(x_j, y_j; \theta) + \frac{\partial^2}{\partial y^2} f(x_j, y_j; \theta) - G(x_j, y_j) \right)^2, \tag{47}$$

where expressions for the (partial) derivatives of $\hat{f}$ can be derived analogously to the one above.

5.5.1 Example 5

We consider the fifth example given in [6]. Here, we have the Poisson equation:

$$\frac{\partial^2}{\partial x^2} f(x, y) + \frac{\partial^2}{\partial y^2} f(x, y) = e^{-x}(x - 2 + y^3 + 6y), \tag{48}$$

for $(x, y) \in [0, 1] \times [0, 1]$, with Dirichlet boundary conditions $f(0, y) = y^3$, $f(1, y) = (1 + y^3)e^{-1}$, $f(x, 0) = xe^{-x}$ and $f(x, 1) = e^{-x}(x + 1)$. Equation (43) gives us trial solution:

$$\hat{f}(x, y; \theta) = E(x, y) + x(1 - x)y(1 - y)N(x, y; \theta), \tag{49}$$

where $E(x, y)$ is given by equation (44):

$$\begin{aligned} E(x, y) = (1 - x)y^3 + x(1 + y^3)e^{-1} + (1 - y)x(e^{-x} - e^{-1}) \\ + y\left[e^{-x}(x + 1) - \left(1 - x + 2e^{-1}x\right)\right]. \end{aligned} \tag{50}$$

This has exact solution $f(x, y) = e^{-x}(x + y^3)$.

We create a neural network $N(x, y; \theta) : [0, 1] \times [0, 1] \to \mathbb{R}$, with one hidden layer and the tanh activation function (figure C.15). Next, since our input data is no longer one-dimensional, we create a new Dataset object (figure C.16). Taking 10 evenly-spaced x- and y-values across the interval $[0, 1]$, we can use the PyTorch 'meshgrid' function to take the Cartesian product of these points. This gives us a lattice of 100 points spread across the domain $[0, 1] \times [0, 1]$ as training points. We must then format this into a tensor of shape $(100, 2)$ in order to pass it into our network as input.

Explicit expressions for the partial derivatives of $\hat{f}(x, y; \theta)$ are easy to derive, hence we omit this code. Our function to train the network is shown in figure C.17.

To calculate the first- and second-order partial derivatives, we use the 'grad' function to calculate the derivative with respect to the whole tensor 'batch', of shape (m,2) (where m is the batch size), i.e. with respect to both the x- and y-values simultaneously:

```
dn  = grad(n_out, batch, torch.ones_like(n_out), retain_graph=True, create_graph=True)[0]
dn2 = grad(dn, batch, torch.ones_like(dn), retain_graph=True, create_graph=True)[0]
```

Doing this is important to ensure that the gradient history is properly retained. Using the 'split' function, for example, to separate 'batch' into a tensor for x and a tensor for y, and then calculating the gradient with respect to these tensors, will not work. This is because the tensors x and y belong to a new computation graph originating at 'batch'. They do not appear in the computation graph for the neural network's output, only 'batch' does. Hence PyTorch will throw an error.

It would be natural to consider passing the separate tensors x and y as inputs to our network, and so removing the need to split any tensors at all. However, this is not possible in PyTorch: since our network takes two inputs, the input tensor *must* be of shape (m,2). An alternative would be to create separate tensors x and y in our Dataset object, then *concatenate* these into shape (m,2) and pass them through our network. Then, calculating gradients with respect to x or y individually is possible, since the 'cat' function retains gradient history, and so our computation graph starts at x and y. We avoid this method in this example, since we require derivative with respect to the full 'batch' anyway, but in a later example (section 6.3) this will not be the case.

This distinction is important: using the 'split' function to separate our *outputs* only extends the existing computation graph, and so allows correct gradient calculation with respect to our initial inputs. Whereas splitting our *inputs* creates tensors which share no history with our outputs, and hence prohibits these gradient calculations.

5.5.2 Effect of Learning Rate

We use this example to explore the effect the learning rate, α, has on training our network. Up until this point we have generally chosen the learning rate for our networks to be 10^{-3}. For each example, this figure was reached by a trial and error process similar to the one shown below.

We first ascertain the correct order of magnitude for our learning rate. To do this, we take $\alpha = 10^{-i}$ for $i = 1, 2, \ldots, 10$, and train ten networks with identical initial parameter values, for 10,000 epochs (code in figure C.18). Figures 43 and 44 below show the effects of these learning rates on the networks' test costs and solution inaccuracies respectively (when tested on 100 points uniformly sampled from the input domain $[0, 1] \times [0, 1]$). Figure 45 below shows the output of the neural network trained with learning rate 10^{-3}, compared to the exact solution. The plots of the other networks are not shown, since they are virtually identical.

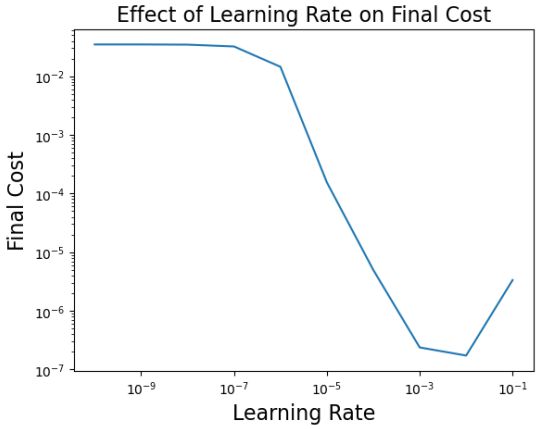

Figure 43: Test cost of neural networks trained with learning rates between 10^{-10} and 10^{-1} to solve equation (48) on $[0, 1] \times [0, 1]$ for 10,000 epochs.

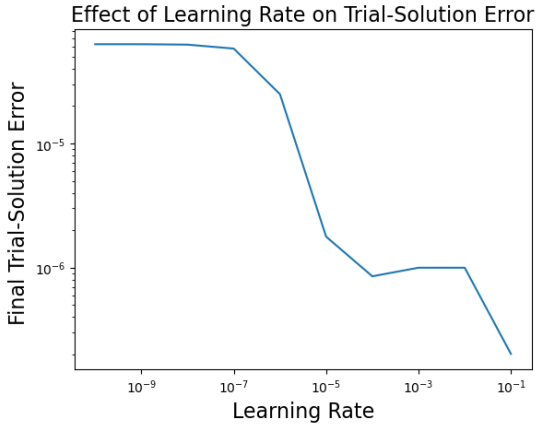

Figure 44: Solution inaccuracy of neural networks trained with learning rates between 10^{-10} and 10^{-1} to solve equation (48) on $[0, 1] \times [0, 1]$ for 10,000 epochs.

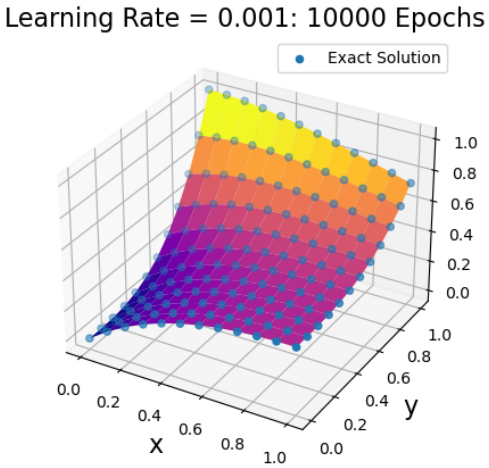

Figure 45: Output of neural network with learning rate 10^{-3} trained to solve equation (48) on $[0, 1] \times [0, 1]$ for 10,000 epochs, compared to the exact solution.

For learning rates any smaller than 10^{-6}, both errors remain high: these learning rates are too small to provide meaningful improvement of our solution within a small number of epochs. Between 10^{-5} and 10^{-2}, increasing the learning rate leads to a reduction in the test cost value. Meanwhile a learning rate of 10^{-1} seems to provide a worse cost value than 10^{-2} or 10^{-3}, for example. The learning rate's effect on solution inaccuracy is mostly correlated with its effect on test cost, with a slight deviation around 10^{-2}. We will examine these two observations in more detail shortly.

Figure 46 shows the training cost of each epoch for the networks with learning rates 10^{-4} and 10^{-3}, while figure 47 shows the same for learning rates 10^{-2} and 10^{-1}.

Although the descent for 10^{-4} is smooth, it achieves the third best final cost value of the four learning rates. 10^{-3} attains the second best, with a descent that is entirely smooth aside from some small spikes in the later epochs, similar to those we have seen in previous examples.

The largest two learning rates have a much noisier descent. For $\alpha = 10^{-2}$, we see a 'noisy plateau' commencing shortly before the 4000^{th} epoch, and hence there is little improvement on its cost after this point. Despite this, it achieves the best final cost value of the four. For 10^{-1}, the cost varies chaotically throughout training. Its best value is actually attained before the 2000^{th} epoch, but the learning rate is large enough that it moves away from this region. It even attains cost values close to 1 at certain points, and gives no indication of convergence.

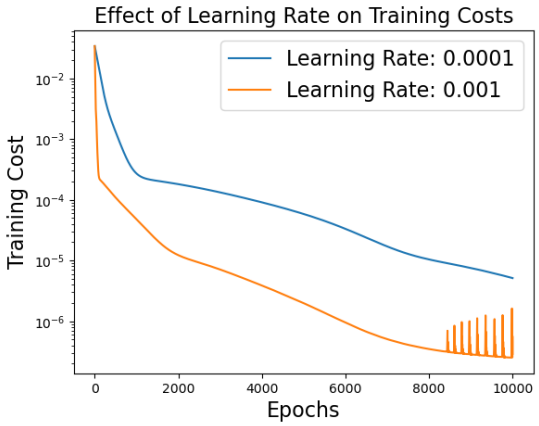

Figure 46: Cost values throughout training of networks trained with learning rates 10^{-4} and 10^{-3} to solve equation (48) on $[0,1] \times [0,1]$ for 10,000 epochs.

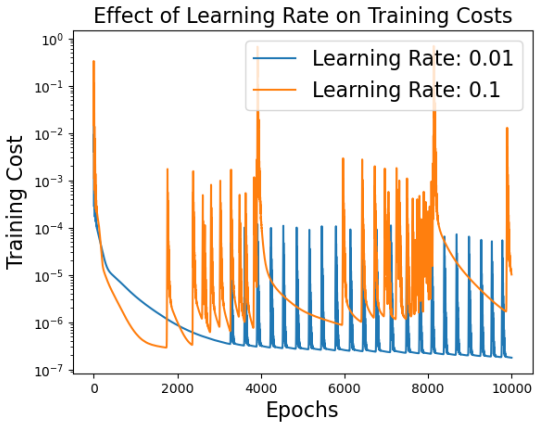

Figure 47: Cost values throughout training of networks trained with learning rates 10^{-2} and 10^{-1} to solve equation (48) on $[0,1] \times [0,1]$ for 10,000 epochs.

Clearly, we have identified an ideal region for our learning rate, between 10^{-3} and 10^{-2}. Smaller than this region, our training is too slow, while for $\alpha > 10^{-2}$, the large learning rate causes erratic movements across the error surface and inhibits convergence.

Figures 48 and 49 show the test cost and solution inaccuracy for ten more (initially) identical networks, this time trained with learning rates between 10^{-3} and 10^{-2}.

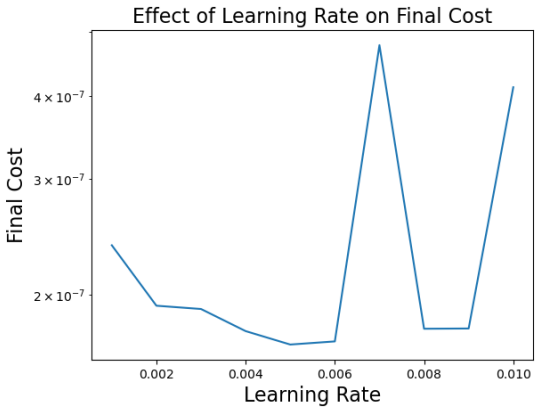

Figure 48: Test cost of neural networks trained with learning rates between 10^{-3} and 10^{-2} to solve equation (48) on $[0,1] \times [0,1]$ for 10,000 epochs.

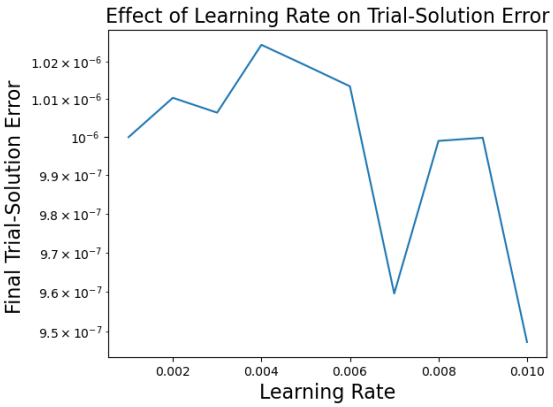

Figure 49: Solution inaccuracy of neural networks trained with learning rates between 10^{-3} and 10^{-2} to solve equation (48) on $[0,1] \times [0,1]$ for 10,000 epochs.

We can see some fluctuation in the final cost values seen: learning rates between 0.004 and 0.006 seem to perform better. Note that the solution inaccuracy does not seem to be correlated with this. As we have seen previously, this is a difficulty which can arise when using differential equations to approximate solution functions numerically: it is possible for the cost value to be small while the solution inaccuracy remains high. Note however the scales of the y-axes of the two

graphs: although the solution inaccuracy does not seem to be correlated with the test cost, the fluctuations in solution inaccuracy are much smaller, and none of the values observed are large.

In general, the ideal learning rate for a problem is very dependent on the problem itself, and requires a trial-and-error process to be ascertained. We will see later (section 6.3.1) that an **adaptive** learning rate is also often necessary.

5.5.3 Example 6

We now consider the final example given in [6], in which we have *non-linear* variant of the Poisson equation:

$$\frac{\partial^2}{\partial x^2}f(x,y)+\frac{\partial^2}{\partial y^2}f(x,y)=\sin(\pi x)(2-\pi^2y^2+2y^3\sin(\pi x))-f(x,y)\frac{\partial}{\partial y}f(x,y), \tag{51}$$

for $(x,y)\in[0,1]\times[0,1]$, with mixed boundary conditions $f(0,y)=f(1,y)=f(x,0)=0$ and $\frac{\partial}{\partial y}f(x,1)=2\sin(\pi x)$. Here equations (45) and (46) give us trial solution:

$$\hat{f}(x,y;\theta)=y^2\sin(\pi x)+x(1-x)y\left[N(x,y;\theta)-N(x,1;\theta)-\frac{\partial}{\partial y}N(x,1;\theta)\right]. \tag{52}$$

In this case the analytic solution is $f(x,y)=y^2\sin(\pi x)$.

We train a neural network identical to that of the previous example (figure C.15). The calculation of the first and second partial derivatives of our trial solution are not difficult, but lengthy (formulae are shown in figures C.19, C.20 and C.21).

Calculating these requires many different mixed partial derivatives of our network's output, both at our training points (x,y) and at points $(x,1)$. This means their calculation using PyTorch's 'grad' function requires more care. Recall that automatic differentiation only ever gives the *value* of the derivative at a given point, and never produces an analytic expression for the derivative. Thus this point must always be specified.

To calculate $\frac{\partial}{\partial y}N(x,1;\theta)$, for example, we should pass the input $(x,1)$ through our network, and calculate the 'grad' of $N(x,1;\theta)$ with respect to $(x,1)$. This will give us a tensor containing both $\frac{\partial}{\partial x}N(x,1;\theta)$ and $\frac{\partial}{\partial y}N(x,1;\theta)$, which we can then 'split'. This is done as shown below (lines 281-282 of figure C.22):

```
# Get all required derivatives of n(x,1):
grad_n_outX1 = grad(n_outX1, x1, torch.ones_like(n_outX1), retain_graph=True, create_graph=True)[0]
n_outX1_x, n_outX1_y = torch.split(grad_n_outX1 , 1, dim = 1 )     # n_x |(y=1) , n_y |(y=1)
```

It is then also possible to calculate mixed partial derivatives. For example, to obtain $\frac{\partial}{\partial y}\frac{\partial}{\partial x}N(x,1;\theta)$, we can calculate the 'grad' of $\frac{\partial}{\partial x}N(x,1;\theta)$ with respect to $(x,1)$, and 'split' the resulting tensor into $\frac{\partial^2}{\partial x^2}N(x,1;\theta)$ and $\frac{\partial}{\partial y}\frac{\partial}{\partial x}N(x,1;\theta)$. This is done like so (lines 284-285 of figure C.22):

```
dn2dx_x1 = grad(n_outX1_x, x1, torch.ones_like(n_outX1_x), retain_graph = True, create_graph = True)[0]
n_outX1_xx , n_outX1_xy = torch.split(dn2dx_x1, 1, dim = 1)     # n_xx |(y=1), n_xy |(y=1)
```

5.5.4 Comparison of Sampling Methods for Training Data

We use this example to explore the effect of our method of sampling training data. Until this point, we have taken evenly-spaced points across our training interval in one-dimensional problems, and the Cartesian product of evenly-spaced points (i.e. a lattice/mesh) in two-dimensional scenarios. However, it is interesting to consider if randomly selected points might provide a better solution. In fact, some authors have shown that so-called **meshfree** implementations of the Lagaris method, in which new training points are randomly selected every epoch according to some probability distribution, perform better for PDEs of higher dimension. This occurs since the number of data points in a lattice is an order of magnitude greater for every extra dimension, and thus a lattice becomes computationally infeasible [29].

We compare training on a fixed lattice of $M=100$ points (see figure 50 below) with two other sampling methods. In the first method, every epoch we sample 100 points uniformly across

the interval $[0, 1]$ for x and repeat this for y to create 100 uniformly sampled (x, y) coordinates (figure 51). In the second, every epoch we sample 100 points for x from the Normal distribution $\mathcal{N}(0.5, 0.11)$, such that the probability of any generated point lying outside the region $[0, 1]$ is negligible. As before, we repeat this for y to create 100 normally sampled (x, y) coordinates (figure 52). This choice of Normal distribution leads to a higher concentration of points in the centre of the square $[0, 1] \times [0, 1]$. A priori, knowing nothing of the solution function, this is the region where we could assume the cost to be highest, since we know it will be identically zero on the boundary of the square. Hence it is interesting to consider how the approximation will compare if we focus our training on this region. The code to create such data sets is shown in figures C.23 and C.24.

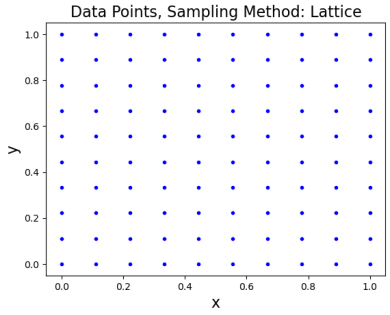

Figure 50: Lattice of training points across the square $[0, 1] \times [0, 1]$.

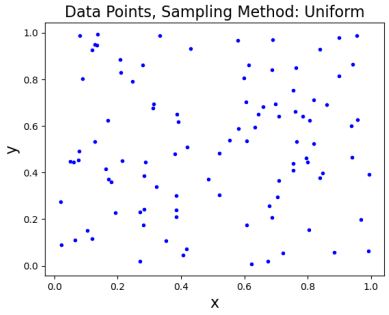

Figure 51: Training points uniformly sampled across the square $[0, 1] \times [0, 1]$.

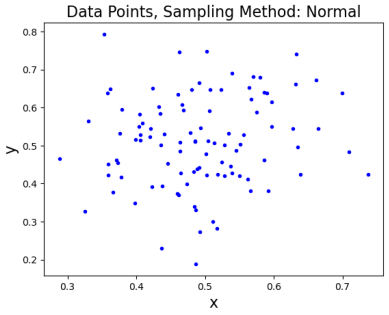

Figure 52: Training points where $x-$ and $y-$values are taken from the distribution $\mathcal{N}(0.5, 0.11)$.

We train each of these networks for 10,000 epochs and compare their outputs (code in figure C.25). Figure 53 below shows the output of the neural network trained on a lattice; we do not show the outputs of the other networks since the surfaces they produce are virtually identical. Figure 54 shows the networks' cost values throughout training, while table 6 compares their training time, as well as test cost and solution inaccuracy when tested on 100 points sampled uniformly from the square $[0, 1] \times [0, 1]$.

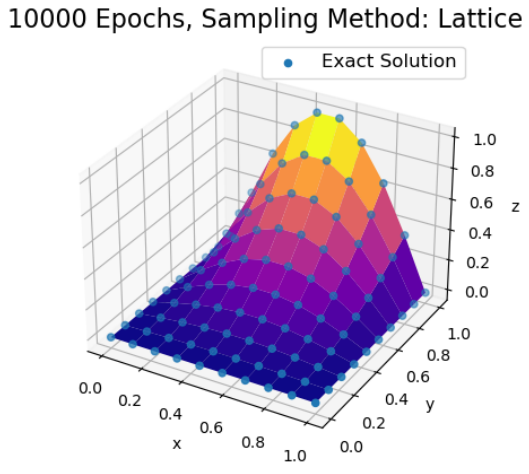

Figure 53: Output of neural network trained on a lattice of training points across the square $[0, 1] \times [0, 1]$ to solve equation (51) for 10,000 epochs, compared to exact solution.

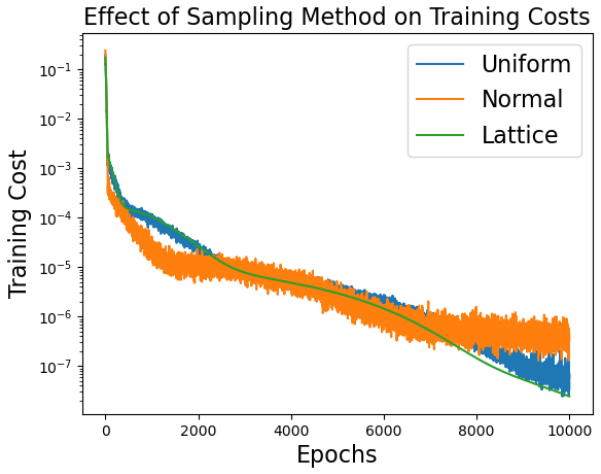

Figure 54: Training cost values of networks trained to solve equation (51) for 10,000 epochs with 3 different methods of sampling training points.

All three methods have provided an adequate approximation of the solution function. We notice that the cost graph's descent for the network trained on a lattice is smooth, while there is a small amount of noise in the descent of the uniform and Normal sampling methods. This can be attributed to the random selection of new training points on each epoch (similar to mini-batch gradient descent).

	Lattice	Uniform	Normal
Training Time (s)	53.35	60.87	63.80
Test Error	2.07×10^{-8}	4.08×10^{-8}	9.24×10^{-6}
Solution Inaccuracy	2.94×10^{-12}	2.13×10^{-11}	5.09×10^{-9}

Table 6: Comparison of three types of data sampling methods used when training a neural network to solve equation (51).

We also notice that sampling new data points every epoch in the uniform and Normal networks has increased training time by approximately 10 seconds. Most importantly, we notice that both the test error and solution inaccuracy for the network trained on normally-distributed points are significantly higher. We can conclude that the training points were not distributed over a large enough region of our domain to generalise as well as the other sampling methods.

Meanwhile, the network trained on a lattice has performed slightly better than the network train on uniform samples. There are a number of reasons why this might be the case. Firstly, since our solution is smooth and changes gradually (i.e. with small gradient), it varies little between neighbouring lattice points. Hence our solution trained on these lattice points will interpolate well. For surfaces with more local variation, uniform sampling might perform better, since these small perturbations can be detected.

Furthermore, due to the random nature of the sampling, it may be that some samples represent the whole training domain less well than the lattice does. Hence parameter updates based on these samples do not necessarily lead to as significant of a decrease in global cost. This is represented in the noise in the cost graph, and may be what has slowed the convergence of this network.

We note however that the network's output performs only slightly worse than the network trained on the lattice. As mentioned in [29], in higher dimensions uniform sampling can speed up training by requiring fewer training points than the lattice, and so in these scenarios it is preferred.

6 Solution Bundles

So far we have seen the Lagaris method in its most basic form, applied to a variety of examples. However, since the method's first appearance in the 1990s, it has been developed and expanded upon by many different researchers. We consider an example from 2020, in which the notion of **solution bundles** is proposed [30].

6.1 Description of the Method

We take a general (system of) first-order differential equation(s):

$$D\left(t, \mathbf{x}, \frac{d\mathbf{x}}{dt}\right) = 0, \tag{53}$$

where $t \in [t_0, t_f]$ represents time, $\mathbf{x} = (x_1(t), \ldots, x_n(t)) \in \mathbb{R}^n$, $x_i : [t_0, t_f] \to \mathbb{R}$ for $i = 1, \ldots, n$, and $\frac{d\mathbf{x}}{dt} = (x_1'(t), \ldots, x_n'(t))$. Let $\mathbf{x_0}$ denote initial conditions of $\mathbf{x}$ at $t = t_0$, and assume that the ODE is such that these initial conditions uniquely determine the solution $\mathbf{x}$. Let $X_0 \subset \mathbb{R}^n$ denote a subset of possible initial conditions for $\mathbf{x_0}$. Then we can define a **solution bundle** to equation (53) as $\mathbf{x}(t, \mathbf{x_0})$, i.e. a *collection* of functions which satisfy the differential equation for all $t \in [t_0, t_f]$, each one corresponding to a given set of initial conditions $\mathbf{x_0} \in X_0$.

Then we can train a neural network with parameters θ, $N(t, \mathbf{x_0}; \theta) : \mathbb{R}^{n+1} \to \mathbb{R}^n$, to approximate this solution bundle. To do this, we create a bundle of trial solutions:

$$\hat{\mathbf{x}}(t, \mathbf{x_0}; \theta) = \mathbf{x_0} + B(t)N(t, \mathbf{x_0}; \theta), \tag{54}$$

where $B(t) : [t_0, t_f] \to \mathbb{R}$ is such that $B(t_0) = 0$, to guarantee that $\hat{\mathbf{x}}(t_0, \mathbf{x_0}; \theta) = \mathbf{x_0}$. Choosing M training points $\{(t_1, \mathbf{x_0}_1), \ldots, (t_M, \mathbf{x_0}_M)\} \subset [t_0, t_f] \times X_0$, we can train the network by minimising

the cost function:

$$J(\theta) = \frac{1}{M}\sum_{j=1}^{M} W(t_j)\cdot D\left(t_j, \hat{\mathbf{x}}\left(t_j, \mathbf{x_0}_j; \theta\right), \frac{\partial \hat{\mathbf{x}}\left(t_j, \mathbf{x_0}_j; \theta\right)}{\partial t}\right)^2, \tag{55}$$

where $W(t) : [t_0, t_f] \to \mathbb{R}^+$ is some monotonically decreasing **weighting function** designed such that error for earlier values of t has more influence than error for later values (e.g. $W(t) = e^{-\lambda(t-t_0)}$ for some $\lambda > 0$). The theoretical justification for this weighting function is given in [30]: they show that high *local* error for early values of t can cause exponential growth in the *global* error of the solution bundle. Hence giving the network a predisposition to reduce the early local error can actually lower the upper bound for the global error across the interval $[t_0, t_f]$.

In essence, this method extends the Lagaris method by taking the initial conditions as inputs to our network and solution function, hence solving the differential equation for a *range* of initial conditions, rather than one fixed set of them. Note that this increases the complexity of the problem greatly and so, as we will see in the example shortly, a much larger network with more hidden layers is required. Since this increases computation time significantly, it is more practical to train our network on a GPU.

To add to this, our input space now has potentially much higher dimension than previous examples. As discussed in the last section, training on a fixed mesh of M training points will be computationally slow if M is large, and will cause poor generalisation to arbitrary values of t and $\mathbf{x_0}$ if M is small. Instead, as in [29], it is suggested that for each parameter update, M training points are sampled (uniform) randomly from $[t_0, t_f] \times X_0$. This then removes the idea of an epoch, since the size of our training data set is technically infinite, and each training batch is unique.

6.2 Example: Planar Circular Restricted Three-Body Problem

We describe an example given in [30] to demonstrate this method, in which a special case of the **three-body problem** is modelled. The general three-body problem relates to finding solutions to the differential equations governing the motion of three masses interacting with each other under the laws of Newtonian gravity (for example, the Earth, the moon and an asteroid). It is a problem for which no closed solution exists, however it is crucial to modelling many scenarios in astrophysics, and so numerical solutions are frequently employed [31].

We study a simplified version of this problem, in which we neglect the mass of the third body (i.e. the asteroid), such that it has no effect on the motion of the other two. We can then consider the motion of this asteroid "in the co-rotating frame of the first two bodies in circular orbits around their barycenter" [30], i.e. the frame with respect to which the Earth and the moon are *stationary*, and so only the asteroid appears to move. Furthermore, we restrict this to planar motion, such that the Earth and moon have fixed positions $(0, 0)$ and $(1, 0)$ respectively, and the asteroid moves only in this xy-plane.

Let m_1, m_2 be the masses of the Earth and moon respectively, and define $\mu = m_2/(m_1 + m_2)$. Then $\tilde{m}_1 = 1 - \mu$ and $\tilde{m}_2 = \mu$ are the 'non-dimensionalised' masses of the Earth and moon respectively. Let $\mathbf{x}(t) = (x(t), y(t))^{\mathbf{T}}$ and $\mathbf{u}(t) = (u(t), v(t))^{\mathbf{T}}$ denote the position and velocity vectors of the asteroid. Then the motion of the asteroid is given by the (non-dimensionalised) equations [30]:

$$\begin{aligned} \frac{dx}{dt} &= u, & \frac{du}{dt} &= x - \mu + 2v - \left[\frac{\mu(x-1)}{((x-1)^2+y^2)^{3/2}} + \frac{(1-\mu)x}{(x^2+y^2)^{3/2}}\right], \\ \frac{dy}{dt} &= v, & \frac{dv}{dt} &= y - \mu + 2u - \left[\frac{\mu y}{((x-1)^2+y^2)^{3/2}} + \frac{(1-\mu)y}{(x^2+y^2)^{3/2}}\right]. \end{aligned} \tag{56}$$

As suggested in [30], we take the ratio of masses to be $\mu = 0.01$. We wish to solve this for initial values $\mathbf{x_0} = (x_0, y_0, u_0, v_0) \in X_0 = [1.05, 1.052] \times [0.099, 0.101] \times [-0.5, -0.4] \times [-0.3, -0, 2]$

across time range $t \in [0, 3]$. Note that none of these ranges are particularly large but, as we will see shortly, even this will increase the complexity of our problem significantly.

To create a solution bundle, we take a neural network $N(t, \mathbf{x_0}; \theta) : [t_0, t_f] \times X_0 \to \mathbb{R}^4$. As in previous examples, we construct a trial solution from these outputs which satisfies the initial conditions. We could take, for example, the trial solution $\hat{\mathbf{x}}_\mathbf{0}(t, \mathbf{x_0}; \theta) = \mathbf{x_0} + (t - t_0)N(t, \mathbf{x_0}; \theta) = \mathbf{x_0} + tN(t, \mathbf{x_0}; \theta)$, as in the original Lagaris method. However, in [30], they propose an alternative which they found to perform better in practice: $\hat{\mathbf{x}}_\mathbf{0}(t, \mathbf{x_0}; \theta) = \mathbf{x_0} + (1 - e^{(t_0 - t)})N(t, \mathbf{x_0}; \theta) = \mathbf{x_0} + (1 - e^{-t})N(t, \mathbf{x_0}; \theta)$. The reasoning behind this is that $1 - e^{(t_0 - t)}$ tends to 1 as t increases, hence its effect diminishes. Meanwhile $t - t_0$ is unbounded, and hence predisposes our trial solution to grow with time. Thus for each input $(t, \mathbf{x_0}) \in [0, 3] \times X_0$, we have trial solution:

$$\hat{\mathbf{x}}_\mathbf{0}(t, \mathbf{x_0}; \theta) = \mathbf{x_0} + (1 - e^{-t})N(t, \mathbf{x_0}; \theta) \tag{57}$$

whose derivative with respect to time is simply:

$$\frac{\partial \hat{\mathbf{x}}_\mathbf{0}(t, \mathbf{x_0}; \theta)}{\partial t} = (1 - e^{-t})\frac{\partial N(t, \mathbf{x_0}; \theta)}{\partial t} + e^{-t}N(t, \mathbf{x_0}; \theta), \tag{58}$$

where we calculate the partial derivative of each element of $N(t, \mathbf{x_0}; \theta)$ using automatic differentiation as usual.

6.3 Implementation

On each parameter update, we create a new data set by uniformly sampling $M = 10,000$ inputs $(t_j, \mathbf{x_0}_j) = (t_j, x_{0_j}, y_{0_j}, u_{0_j}, v_{0_j})$ from $[-0.01, 3] \times X_0$ (figure C.26). Note that we train on values of t slightly below 0, since in [30] it was observed that this gives better solution accuracy for small positive values of t. Our initial conditions are still specified for $t = 0$, however.

We pass this data set through our network and evaluate the trial solutions and their derivatives at these points (code to evaluate these is shown in figures C.27 and C.28). We then substitute these values into the four differential equations (equations (56)), subtract the right-hand sides from the left-hand sides, before squaring and summing these four outputs. This gives us the D^2 term in our cost function (equation (55)) for all $j \in \{1, \dots, M\}$. As suggested in [30], we multiply these values by the weighting function $W(t_j) = e^{-2t_j}$. Finally, we average all M of these and obtain our cost value $J(\theta)$. The code to train this network is shown in figure C.29.

Note that our DataSet object loads the 5 inputs x_0, y_0, u_0, v_0 and t as separate tensors this time. We then concatenate them into a tensor of shape $(M, 5)$, pass this through our network, before splitting the four outputs. Finally, we calculate the 'grad' of each output with respect to t. The 'cat' and 'split' functions in PyTorch retain the gradient history, so our computation graph begins at our five inputs, and ends at our four outputs, as required.

In previous examples, we have split only the output, and calculated the partial derivatives of each *individual* output with respect to the *full* input tensor (in this case of shape $(M, 5)$). We have done this since (almost) all the partial derivatives have been required in previous cases, so it is more efficient to perform fewer backwards passes. Here, of the 20 possible partial derivatives, we require only four (all outputs with respect to t). Hence it is computationally more efficient to calculate only these.

To learn more quickly, we require a larger neural network: we take a neural network of 8 hidden layers, each made up of 128 nodes. We continue to use tanh as our activation function. Multiple hidden layers can be created using a ModuleList object in PyTorch, which acts like a normal Python list, and can be used to store our Linear objects (i.e. our layers). Since we have a deeper network now, and are using tanh, we use Xavier initialisation (see section 2.4.3) to reduce the chance of exploding or vanishing gradients. The code to build this network structure is shown in figure C.30.

In studying this example we also see how easy it is to implement training on a GPU in PyTorch (figure C.31). We verify if a GPU is available with the PyTorch function 'cuda.is_available'. Our GPU is then accessible as the output of the PyTorch 'device' function, with argument 'cuda'. We

store this as a variable called 'device'. All tensors and Module objects (i.e. networks) then have a 'to' method, which accepts as argument a CPU or GPU (cuda) object, and specifies where the tensor or Module is stored. Moving all Modules and tensors to the GPU is all that is required: the distribution of operations across the GPU's processors is handled automatically by PyTorch (although manual optimisation is possible, if required) [20].

Notice that the key difference in this example which makes it ideal for implementation on a GPU is the large batch size. For each step in our training algorithm, we require several identical operations to be performed on all $M = 10,000$ data samples. These can be processed in parallel, and hence training can be carried out more quickly. Interestingly, for a smaller batch size (as in the next example), training on a GPU is actually *slower*. This occurs since time is required for work to be distributed across the GPU's processors; if the batch size is small enough, this time will outweigh the time needed for a CPU to process the whole batch, and thus render the training less efficient [20].

In this example we do not have an analytic solution with which to compare our network's approximation. Instead, we compare the network's output to the solution given by a popular finite-difference method for solving differential equations: the **fourth-order Runge-Kutta method**. A reminder of this method is given in appendix B, and code to apply this to equations (53) is shown in figures C.32 and C.33.

6.3.1 Learning Rate Decay

As we have seen previously, one fixed learning rate is generally not suitable to obtain an optimal solution. Even though Adam, our preferred optimisation algorithm, has ways of altering the step size for each individual parameter (see section 4.5), it still uses a fixed learning rate α for all parameters. We have seen that a trial-and-error process can be used to ascertain preferable values of α. We have also seen that a larger initial learning rate can speed up training and avoid convergence to a local minimum, however this larger learning rate can later cause 'spikes' in our cost graph that indicate oscillation around a minimum and hinder convergence to it.

A natural solution to this dilemma is to decrease the learning rate later in training, otherwise known as **learning rate decay**. In this way we retain the benefits of a larger initial learning rate, while avoiding the later oscillations and thus accelerating our later training. Although this gives an intuition of why this method improves training, it has also been shown that decreasing the learning rate over time has wider benefits in more complex problems. The larger α early in training prevents the network overfitting to noisy training data, while the smaller learning rate later increases the network's ability to learn complicated patterns within the data [32].

The question that then arises is *how* should we decrease the learning rate? There are a number of basic ways to do this [20]. For example, let α_t denote the learning rate on epoch t (or batch t in our current example), where we choose beforehand an initial value α_0. Then on epoch t we could update α multiplicatively:

$$\alpha_t = \beta \cdot \alpha_{t-1}, \tag{59}$$

for some fixed $\beta \in [0, 1)$. Another option would be do decrease α in 'steps', i.e. multiply α by β every T epochs:

$$\alpha_t = \alpha_0 \cdot \beta^{\left[\frac{t}{T}\right]}, \tag{60}$$

where $\left[\frac{t}{T}\right]$ indicates the integer part of t/T. Alternatively, we could have α decay exponentially:

$$\alpha_t = \alpha_0 \cdot e^{-\gamma t}, \tag{61}$$

for some fixed $\gamma > 0$. These can all be easily implemented in PyTorch using objects known as **schedulers**.

We choose to implement a more complicated form of learning rate decay known as **reducing learning rate on plateau** [20]. Here, we reduce the learning rate when we observe a (potentially noisy) plateau in the cost graph, i.e. when we have not seen a significant decrease in cost for a certain number of epochs. More specifically, we fix first a few hyperparameters:

- $\beta \in [0, 1)$: the multiplicative factor by which we reduce the learning rate,
- $p \in \mathbb{N}$: our **patience**, i.e. the number of epochs in which we observe no improvement in cost value before decreasing α,
- $\delta \in [0, 1)$: our **threshold** by which a new cost value must improve upon the current best value to be considered a significant improvement.

Let $n \in \mathbb{N}$ denote the number of epochs in which we have seen no improvement in cost. Then on epoch t we compare the cost, J_t, with $(1 - \delta)J_{best}$, where J_{best} is our lowest observed cost value. If $J_t < (1 - \delta)J_{best}$, then $J_{best} \leftarrow J_t$ and $n \leftarrow 0$. Otherwise, we have seen no improvement in cost: $n \leftarrow n + 1$, and we compare n to our patience p. If $n > p$, then $\alpha \leftarrow \beta\alpha$ and $n \leftarrow 0$. If $n \leq p$, we continue with the current learning rate. Note that:

$$J_t < (1 - \delta)J_{best} \iff \delta < \frac{J_{best} - J_t}{J_{best}}, \tag{62}$$

i.e. we consider J_t to be significantly less than J_{best} if their difference (relative to the size of J_{best}) is greater than δ. Hence the larger δ is, the greater an improvement we require.

Although this method of learning rate decay is more complicated, its adaptive nature makes it preferable to the basic methods mentioned previously. This is because the learning rate is only reduced when our cost is not decreasing; while the current learning rate is performing well, we do not alter it. As always, ideal choices of hyperparameters remain to be found and, without prior knowledge, are typically determined for a given example via trial and error.

In PyTorch, default values for these hyperparameters are $\beta = 0.1$, $p = 1$ and $\delta = 10^{-4}$. However, in [30], they observed that values $\beta = 0.5$, $p = 200,000$ and $\delta = 0.5$ gave better performance for the three-body example. Note also that it is possible to specify a minimum value for the learning rate to be reduced to. When implementing this example, we set this minimum value to be 10^{-6}, since for α less than this value, the rate of decrease of cost is minimal.

6.4 Results

We train our network described in the previous sections to create a solution bundle to equations (56). The code to implement this is shown in figure C.31. We train on $3,000,000$ batches of $10,000$ uniformly-sampled data samples; note that this training has taken over 48 hours, even on a GPU. Our network has outputs shown in figure 55, for a set of 10 uniformly-sampled initial conditions, and for $1,000$ values of t evenly-spaced across the interval $[0, 3]$. Figure 56 shows the network's cost values throughout training. Testing on another random batch of $10,000$ inputs, this network has test cost of 3.66×10^{-8}, and a solution inaccuracy of 1.38×10^{-4} (compared to the Runge-Kutta solution).

The network has provided a near-perfect approximation of the solution for early values of t, while for t close to 3 we see an increase in inaccuracy. This can be attributed to the weighting factor e^{-2t} we included in our cost function: earlier values of t contribute more to the cost, and hence our optimisation algorithm prioritises a better approximation for these values of t.

We notice also that our cost graph has the form of a sequence of steps, each one resembling a noisy plateau. This is the effect of our learning rate scheduler: between the start of training and approximately batch $2,400,000$, our learning rate has been reduced (halved) ten times. Starting at 10^{-3}, it has now decreased to our specified minimum: 10^{-6}. On each reduction we see an immediate drop in cost value as, with a reduced step-size, we are able to descend further down into the error landscape. With such a small learning rate, the final $600,000$ batches of training show little decrease in cost.

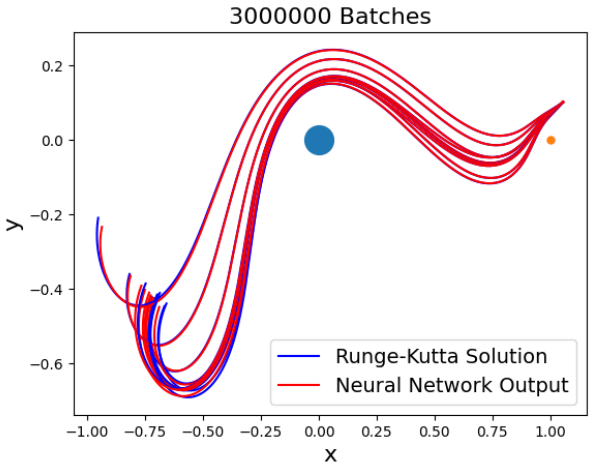

Figure 55: Output of neural network trained to solve the planar circular restricted three-body problem on a bundle of initial conditions, compared to the 4^{th}-order Runge-Kutta solution. The Earth and the moon are shown in blue and orange.

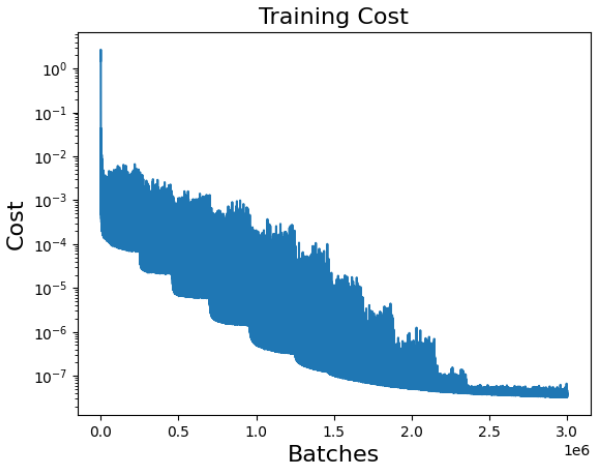

Figure 56: Training cost values of network trained to solve the planar circular restricted three-body problem on a bundle of initial conditions, for 3,000,000 batches of 10,000 inputs.

Note the improvement this has had on our final cost: without reducing the learning rate, it is possible that we may never have exited the first noisy plateau with a cost value of around 10^{-4}. Figure 57 shows how the network's output evolves throughout training. The weighting factor in the cost function causes the network's solution to converge faster for earlier values of t.

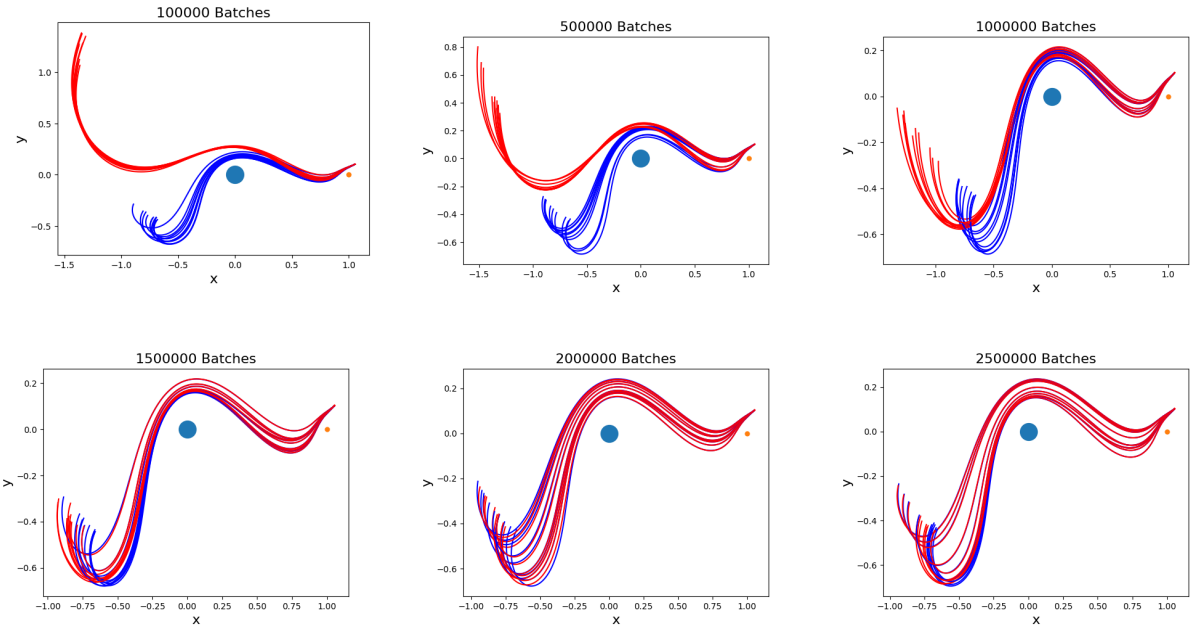

Figure 57: Progress throughout training of neural network solution bundle (red) to the planar circular restricted three-body problem, compared to the 4^{th}-order Runge-Kutta solution (blue). Due to the weighting factor in the cost function, the network converges to the correct solution faster for earlier values of t.

6.5 Curriculum Learning

It is interesting to consider if curriculum learning on our time interval $[-0.01, 3]$ can be applied to this example, particularly given the proof in [30] that reducing the local error for small values of t can lower the global error of the solution bundle. For all four networks considered in this section, we remove the weighting function e^{-2t} from our cost function, to see how curriculum learning performs in place of this.

Taking a network with identical initial parameter values to those in the previous section, we trained it using the curriculum which gave the best results in the example in section 5.4.2: $[-0.01, 1], [1, 2], [2, 3], [-0.01, 3]$. For distinction later, we will refer to this as **separate curriculum learning**. Again, we take $3,000,000$ batches in total; hence we train on $750,000$ batches on each sub-interval (figure C.34). The network's output after training is shown in figure 58 below, while its cost during training is shown in figure 59. This network has a test cost of 7.67×10^{-4} and a solution inaccuracy of 0.228.

Clearly, curriculum learning has provided a much worse approximation of our solution. In the cost graph we can see that, when training on sub-intervals of $[-0.01, 3]$, the network's cost continually decreases (despite the initial spike when beginning to train on a new sub-interval). However, when we return to train on the full interval $[-0.01, 3]$, the cost remains high, around 10^{-3}.

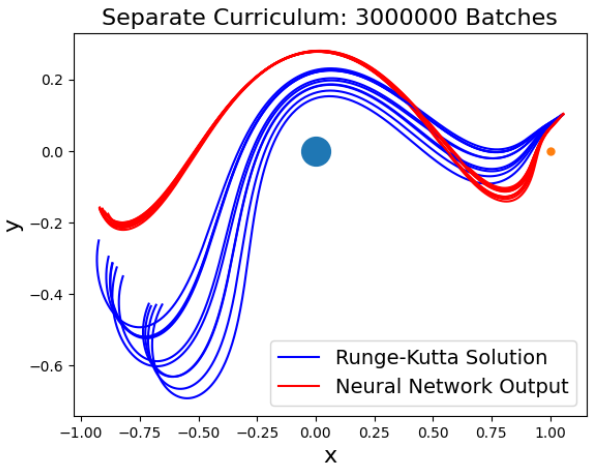

Figure 58: Output of neural network trained using **separate curriculum learning** to solve equations (56) on a bundle of initial conditions, compared to the 4^{th}-order Runge-Kutta solution.

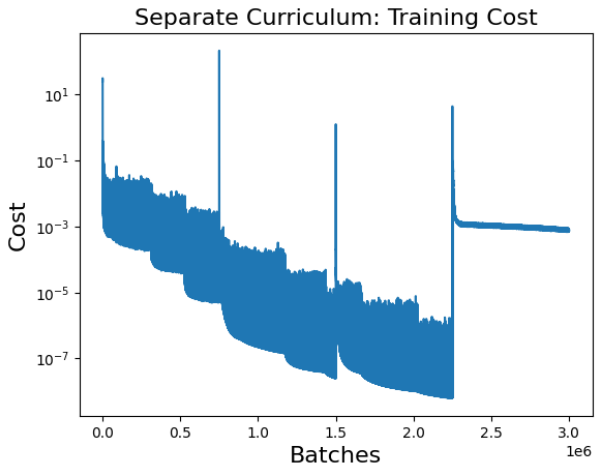

Figure 59: Training cost values of network trained using **separate curriculum learning** to solve equations (56) on a bundle of initial conditions, for 3,000,000 batches of 10,000 inputs.

To train effectively on our sub-intervals, the learning rate decay is required (note the downward steps similar to figure 56). However, when we return to train on the full interval, our learning rate is now too small to significantly change the network's inaccurate output. We can conclude that the differential equation is of sufficient complexity that learning done on the separate intervals does not generalise to the full interval $[-0.01, 3]$. Note this in contrast to the example in section 5.4.2, in which curriculum learning was actually *required* to attain convergence to the correct solution.

We consider another form of curriculum learning discussed in other examples in [30], which we will call **continuous curriculum learning**. Here, let $N = 3,000,000$ denote the total number of training batches, and let $1 \leq n \leq 3,000,000$ denote the number of the current batch. Then we can train our network on a *continuously widening* interval $[-0.01, t_n]$, where t_n depends on the current batch number n. We consider three possible ways of defining t_n:

$$t_n = \min\left\{3,\ \frac{1}{2}\exp\left(\frac{3\log(6)(n-1)}{2.5 \cdot N}\right)\right\}, \tag{63}$$

(where log denotes the natural logarithm) which we will call our **exponential curriculum**,

$$t_n = \min\left\{3,\ \frac{1}{2} + \frac{3(n-1)}{N}\right\}, \tag{64}$$

which we will call our **linear curriculum**, and

$$t_n = \min\left\{3,\ \frac{1}{2} + \frac{2.5\log\left(1 + \frac{3(n-1)}{N}\right)}{\log(3.5)}\right\}, \tag{65}$$

which we will call our **logarithmic curriculum**. These are constructed such that at $n = 1$, $t_n = 0.5$, and for $n = 2,500,001$ (and thus for all $n \geq 2,500,001$), $t_n = 3$. Hence for the final $500,000$ training batches, we train on the full interval $[-0.01, 3]$. Between 1 and $2,500,001$, these three formulae for t_n differ only in their *rate of increase* of t from 0.5 to 3, respectively at an exponential rate, a linear rate and a logarithmic rate.

Figures 60, 61 and 62 show the networks' outputs after training, while figures 63, 64 and 65 show their cost values during training (implemented in figure C.35). Table 7 shows a comparison of the test costs and solution inaccuracy (tested on another randomly generated batch of size $10,000$), alongside those of the previous two methods we have used.

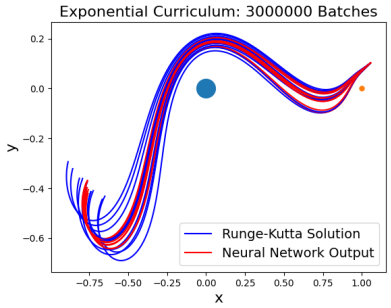

Figure 60: Output of neural network trained on an **exponential curriculum** to solve equations (56) on a bundle of initial conditions, compared to the 4^{th}-order Runge-Kutta solution.

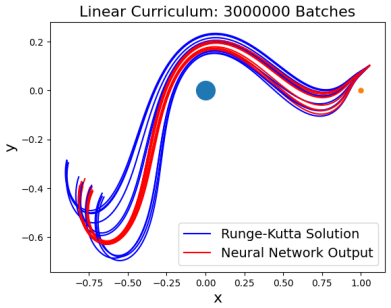

Figure 61: Output of neural network trained on a **linear curriculum** to solve equations (56) on a bundle of initial conditions, compared to the 4^{th}-order Runge-Kutta solution.

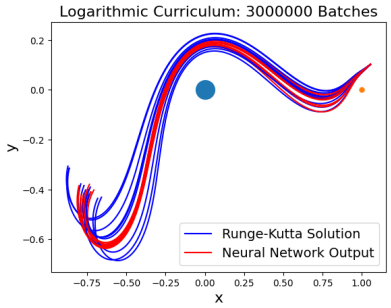

Figure 62: Output of neural network trained on a **logarithmic curriculum** to solve equations (56) on a bundle of initial conditions, compared to the 4^{th}-order Runge-Kutta solution.

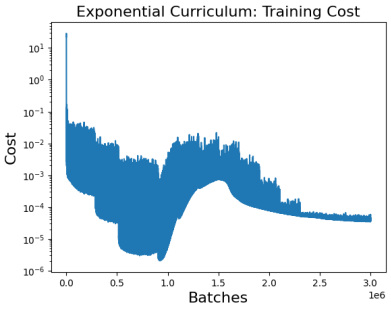

Figure 63: Training cost values of network trained on an **exponential curriculum** to solve equations (56) on a bundle of initial conditions, for 3,000,000 batches.

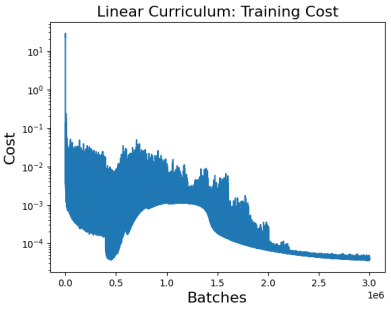

Figure 64: Training cost values of network trained on a **linear curriculum** to solve equations (56) on a bundle of initial conditions, for 3,000,000 batches.

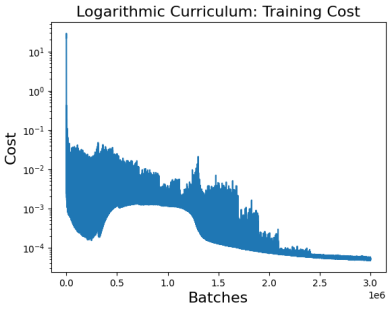

Figure 65: Training cost values of network trained on a **logarthmic curriculum** to solve equations (56) on a bundle of initial conditions, for 3,000,000 batches.

	Weighted	Separate	Exponential	Linear	Logarithmic
Test Cost	3.66×10^{-8}	7.67×10^{-4}	3.76×10^{-5}	4.02×10^{-5}	5.21×10^{-5}
Solution Inaccuracy	1.38×10^{-4}	0.228	2.70×10^{-3}	5.26×10^{-3}	8.22×10^{-3}

Table 7: Comparison of the test costs and solution accuracies of 5 identical networks trained to solve equations (56) on a bundle of initial conditions. The first has a cost function weighted in terms of time t, whereas the others are trained using different versions of curriculum learning.

The test costs of the three different curricula are relatively similar, with the exponential curriculum providing a slightly better solution than the other two, and the logarithmic curriculum attaining the worst. While all three have provided a better solution than the separate curricula method, their approximations are worse than our first network which used a weighted cost function and trained on the full time interval.

In particular, while providing a reasonable approximation for earlier values of t, these three networks' outputs tend to cluster together for later values of t; they do not 'fan out' as the Runge-Kutta solution indicates they should do. Identifying this fanning out clearly requires more training on later values of t, potentially earlier on in training when the learning rate is larger.

The cost graphs of these three networks also share a particular pattern. In all three there is a certain point at which the cost stops decreasing, and actually begins to increase. This occurs roughly when $t_n = 1$, at which point the asteroid's trajectory passes above the Earth and begins to curve downwards. This behaviour must differ from the existing pattern the network has learned, and hence we see an increase in cost. Eventually, after t_n has increased further and the learning rate has been reduced, the cost again begins to decrease.

The value $t_n = 1$ is reached latest for the exponential curriculum, and earliest for the logarithmic curriculum, due to these respectively having the smallest and largest gradient of t_n with respect to n for early n. Hence the increase in cost after $t_n = 1$ is most notable for the exponential curriculum, which had previously attained a cost almost as small as 10^{-6}. Despite this, the final cost value achieved by the exponential curriculum is still the smallest. In fact, we can see that for these three networks, those which trained longer on earlier values of t eventually produced a better solution.

However, we should reiterate that the differences between these three are relatively insignificant when compared to the result of our first network (trained on the full time interval with a weighted cost function). This has attained by far the best solution, and hence we can conclude that curriculum learning (at least in the ways in which we have employed it) is not suitable for this example.

6.6 Comparison to Finite-Difference Methods

It is interesting to compare the solution bundles method with traditional numerical methods for solving differential equations. In [30], they compare the Euler method, the Runge-Kutta method, and the solution bundles method in terms of both work (i.e. number of floating-point operations) and memory required to solve a differential equation for a variety of initial conditions.

They found that the neural network method was the cheapest in terms of memory, while ranked second in terms of computational expense, beaten by the Runge-Kutta method. It is noted, however, that evaluations made by the neural network are able to be carried out in parallel: for a given set of initial conditions, a function from the network's solution bundle can be evaluated for any value of t, and thus for many values of t simultaneously in parallel. This is not possible in the Runge-Kutta method. Given a set of initial conditions, to evaluate the solution at a given time t the solution must be calculated iteratively from t_0 to t in fixed time-steps. Hence not only are later values of t more expensive to compute, we also cannot take advantage of parallelisation to speed up computation time.

This highlights an advantage of both the original Lagaris method and its extension to solution bundles. Once our network is sufficiently trained, it can provide a solution to the differential equation for any point in the input domain with an equal amount of computation. It provides a closed-form expression of a smooth function, with analytic expressions of its derivatives available. This is never the case in finite-difference methods: here we only ever attain numerical estimates for the solution function's values at given points.

The solution bundles method also addresses a problem we have previously encountered with the original Lagaris method (see example in section 5.4.2): in solving a given differential equation (with one fixed set of boundary conditions) with a neural network, our network may converge to a solution which is in fact incorrect, and likely a local minimum of the cost function. We could compare our solution to one found by a traditional numerical method (e.g. Runge-Kutta), but this makes the Lagaris method redundant: it would have been computationally simpler just to use the traditional method originally.

However, if instead we create a solution bundle for the differential equation over a range of boundary conditions, we can occasionally generate some sample solutions using the Runge-Kutta

method for comparison during training. This way, we can avoid the problem of convergence to an incorrect solution, while taking full advantage of the benefits of the neural network method mentioned above.

7 Parameter Inference

Beyond the Lagaris method and its extensions, there are other ways in which neural networks can be applied in the field of differential equations. Another method, for example, proposed in 2017 [33], involves **data-driven discovery** of partial differential equations, aided by neural networks.

7.1 Description of the Method

Suppose we have a differential equation:

$$D\left(x, u(x), \nabla u(x), \nabla^2 u(x), \ldots, \nabla^m u(x);\ \lambda, \ldots, \lambda_p\right) = 0, \qquad x \in \mathrm{dom}(u) \subset \mathbb{R}^n, \tag{66}$$

where $\lambda_1, \ldots, \lambda_p \in \mathbb{R}$ are unknown parameters of the differential equation, and $u(x) : \mathrm{dom}(u) \to \mathbb{R}$ is the unknown solution function. Suppose also that we have M sample points, $x_1, \ldots, x_M \in \mathrm{dom}(u)$, where the value of u is known, $u_1, \ldots, u_M \in \mathbb{R}$ respectively.

Then we can train a neural network to approximate the function u based on these sample points (analogously to the example of $\sin(x)$ in section 3.3) and simultaneously determine the unknown parameters $\lambda_1, \ldots, \lambda_p$.

To do this, let $N(x; \theta) : \mathrm{dom}(u) \to \mathbb{R}$ be a neural network with weights and biases θ. Then we train this network to approximate u and find the correct values of $\lambda_1, \ldots, \lambda_p$ by minimising the cost function:

$$J(\theta, \lambda_1, \ldots, \lambda_p) = J_u(\theta) + J_D(\theta, \lambda_1, \ldots, \lambda_p), \tag{67}$$

where:

$$J_u = \frac{1}{M} \sum_{j=1}^{M} \left(N(x_j; \theta) - u_j\right)^2, \tag{68}$$

$$J_D = \frac{1}{M} \sum_{j=1}^{M} D\left(x_j, N(x_j; \theta), \nabla N(x_j; \theta), \nabla^2 N(x_j; \theta), \ldots, \nabla^m N(x_j; \theta);\ \lambda_1, \ldots, \lambda_p\right)^2. \tag{69}$$

Note that $\lambda_1, \ldots, \lambda_p$ will be trained by the same *optimisation algorithm* as the parameters θ, even though they do not affect the neural network's output.

Observe also how this differs from the Lagaris method. Here, we have explicit data about the solution function, but are lacking data about the differential equation itself. Hence approximating the solution function is (in theory) a relatively simple task, whereas determining the parameters $\lambda_1, \ldots, \lambda_p$ may not be. In the Lagaris method we had the opposite scenario: all the information about the differential equation was available to us, but this was the only metric by which we could understand the solution function. As we have seen, this means approximating the solution could be a more difficult task.

7.2 Example: Burger's Equation

We give an example from [33], in which they consider a generalised form of Burger's equation:

$$\frac{\partial u}{\partial t} + \lambda u \frac{\partial u}{\partial x} - \nu \frac{\partial^2 u}{\partial t^2} = 0, \tag{70}$$

where $u(x, t) : [-1, 1] \times [0, 1] \to \mathbb{R}$, and $\lambda \in \mathbb{R}$, $\nu > 0$. Historically, Burger's equation was first used to model the speed, $u(x, t)$ of a fluid at spatial point x at time t, where $\nu > 0$ represents the viscosity of the fluid, and λ is generally equal to 1 [34]. However, it has since been applied in mathematical modelling in various disparate fields, including non-linear acoustics and traffic flow

[33]. Although an analytic solution exists, when the viscosity term is small it can be difficult to evaluate for small values of t, since certain terms in the solution tend to explode [34]. Similarly in this scenario, it can be difficult to resolve the equation using traditional numerical methods [33].

We use this example to demonstrate the data-driven discovery method. We use $\lambda = 1$, $\nu = 0.01/\pi \approx 0.003183$, initial conditions $u(x, 0) = -\sin(\pi x)$ and boundary conditions $u(\pm 1, t) = 0$. We then take $M = 2,000$ randomly-generated points $(x_j, t_j) \in [-1, 1] \times [0, 1]$, $j = 1, \ldots, M$, and their corresponding u-values $u(x_j, t_j)$, provided in the data sets found in [33].

We create a neural network with two inputs, x and t, and one output, our approximation $N(x, t; \theta)$ of $u(x, t)$. Our cost function is $J(\theta, \lambda, \nu) = J_u(\theta) + J_D(\theta, \lambda, \nu)$, where:

$$J_u(\theta) = \frac{1}{M} \sum_{j=1}^{M} (N(x_j, t_j; \theta) - u_j)^2, \tag{71}$$

$$J_D(\theta, \lambda, \nu) = \frac{1}{M} \sum_{j=1}^{M} \left(\frac{\partial N(x_j, t_j; \theta)}{\partial t} + \lambda N(x_j, t_j; \theta) \frac{\partial N(x_j, t_j; \theta)}{\partial x} - \nu \frac{\partial^2 N(x_j, t_j; \theta)}{\partial t^2} \right)^2, \tag{72}$$

by equations (68) and (69).

7.3 Implementation

We take a neural network with 8 hidden layers, each consisting of 32 nodes and use the tanh activation function. Next we load and format our training data samples $(x_j, t_j, u(x_j, t_j))$ from a MATLAB binary file. There are $25,600$ of these samples; we sample $2,000$ randomly based on their index, and use the rest for testing our solution after training. Our Dataset object (figure C.36) will load the inputs (x_j, t_j) in a tensor of shape $(m, 2)$ (where m is the batch size) and also the true values $u(x_j, t_j)$ in a separate tensor of shape $(m, 1)$.

Our trainable parameters λ and ν can be added to our Module (neural network) object as Parameter objects (see figure C.37). This will create two tensors of shape (1) with 'requires_grad' equal to True, which will be updated by our optimisation algorithm (Adam) alongside all the network's weights and biases. As done in [33], since we know the viscosity ν to be positive, we can instead train to find a $\tilde{\nu}$ where $\nu = \exp(\tilde{\nu})$. Note that we must also specify the initial values of λ and ν (i.e. $\tilde{\nu}$). In [33], they chose $\lambda = 0$ and $\nu = e^{-6} \approx 0.002479$. In our implementation, we initialised λ and $\tilde{\nu}$ with random values between 0 and 1, and found the choice of initial value had virtually no impact on convergence to the correct values.

As in the previous example, we also use a learning rate scheduler to reduce our learning rate upon plateau. We found slightly different hyperparameters to be optimal in this example, however. Specifically, we chose patience $p = 500$ and threshold $\delta = 10^{-4}$, while keeping the multiplicative factor $\beta = 0.5$.

Training can then be carried out as described in the previous section, with cost function given by equations (71) and (72), and where the partial derivatives of $N(x, t; \theta)$ are calculated using automatic differentiation. The code to train our network is shown in figure C.38.

After training this network for $100,000$ epochs (figure C.39), its output when plotted at all $25,600$ data points (x_j, t_j) is shown in figure 66, while the exact solutions $u(x_j, t_j)$ are shown in figure 67. The inaccuracy of our network's output at these points is 1.52×10^{-5}. Figure 68 then shows the total cost values $J(\theta, \lambda, \nu)$ throughout training, while figure 69 shows how λ and ν have changed throughout training. The cost value (tested on all $25,600$ data points) is 8.96×10^{-4}, while our final parameter values are $\lambda = 0.997$ and $\nu = 0.003025$ (relative errors of 0.307% and 4.972% respectively).

Our network has produced an accurate approximation of the true solution and has also ascertained the correct parameter values. Notice in figure 69 that the network converges very quickly to the correct regions for values of both λ and ν. It is also visible in figure 68 that our learning rate has been reduced three times during training.

Although this has provided an accurate solution, we consider also a few modifications to this method for comparison. The first modification is to *first* train the network to approximate $u(x, t)$,

and *then* ascertain the values λ and ν. That is, remove the Parameter objects from our neural network, and train it only with cost function $J_u(\theta)$ from equation (71) (shown in figure C.40). Once a sufficiently low cost value is reached, create two tensors of shape (1) with 'requires_grad' equal to True, separate from the neural network, and pass these to an optimiser as parameters to be updated:

```
lmbda     = torch.tensor(torch.rand(1), requires_grad = True)
nu        = torch.tensor(torch.rand(1), requires_grad = True)
optimiser = torch.optim.Adam([{"params" : lmbda, 'lr' : 1e-3}, {"params" : nu, 'lr' : 1e-3}])
```

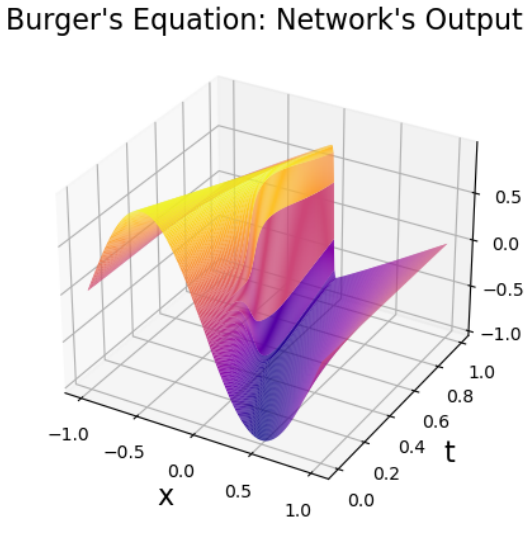

Figure 66: Output of a network trained to approximate the solution to Burger's equation for 100,000 epochs.

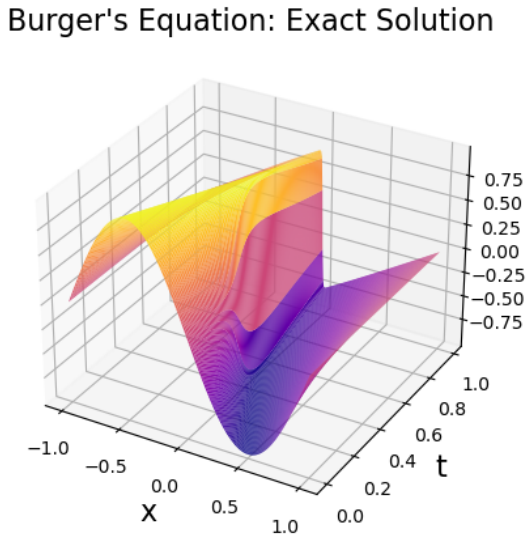

Figure 67: Exact solution to Burger's equation, plotted for 25,600 data sample points.

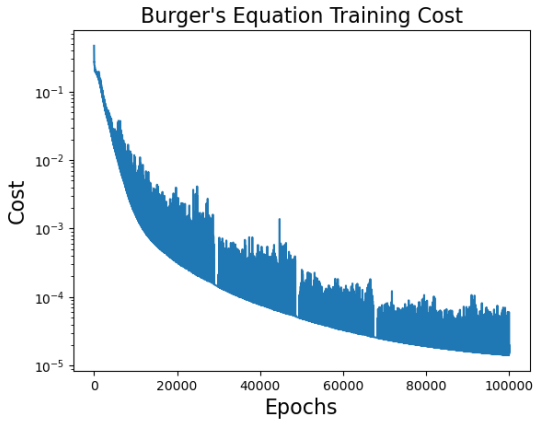

Figure 68: Training cost values of a network trained to approximate the solution to Burger's equation for 100,000 epochs.

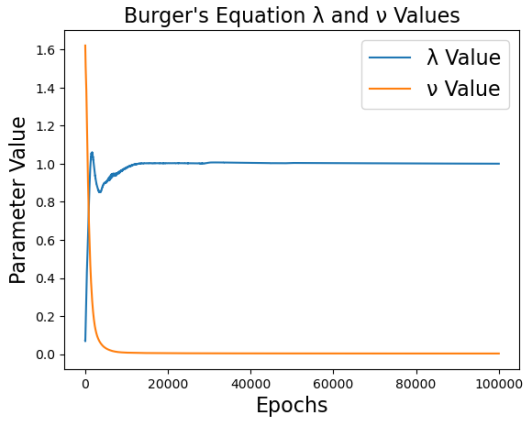

Figure 69: Neural network's approximation of Burger's equation parameter values λ and ν throughout training.

We now train with this optimiser to minimise the cost function $J_D(\lambda, \nu)$, i.e. equation (72), but where our network's parameters θ are *fixed*, and will not be updated by our optimisation algorithm (shown in figure C.41). $J_D(\lambda, \nu)$ is now an error surface with only two coordinates, i.e. a surface in three dimensions much like figure 6, and hence could potentially be a simpler optimisation problem.

We are interested in this method since if we had a perfect approximation of the function $u(x, t)$, then the only values of λ, ν which would satisfy our differential equation are the correct ones. Hence a better approximation of $u(x, t)$ would lead to more accurate values for λ, ν. It may be harder to find these when minimising J_u and J_D simultaneously since the values of θ (over 7,500 parameters) have much more impact on minimising $J = J_u + J_D$ than λ and ν do.

The second alteration is to use a different activation function in the place of tanh. Observing the approximation of $u(x,t)$ in figure 66, we see that for $t = 0$, $u(x,0)$ is sinusoidal. Then, as t increases, this sin curve deforms into a 'saw-tooth' wave, i.e. the gradient at $x = 0$ becomes steeper and steeper, to the point of almost having a discontinuity at $t = 1$. It is this shape that makes Burger's equation difficult to solve for small values of ν.

Recall when we compared activation functions in section 3.3 that ReLU, not being a smooth function, provided an approximation of $\sin(x)$ which was not smooth. We could then potentially consider ReLU as a better candidate to approximate this 'discontinuity' at $x = 0$. However, recall that the second derivative of ReLU is identically zero. This means that when we come to evaluate J_D, the second-order partial derivative of our network's output will always be zero.

Hence we seek an activation function which mimics this lack of smoothness, but whose second derivative is not zero. A suitable choice could be an activation function known as **Swish**, defined as $\text{Swish}(x) = x \cdot \sigma(\mu x) = x/(1 + e^{-\mu x})$, where $\mu > 0$ and $\sigma(x)$ denotes the sigmoid function. Swish is a relatively recent activation function, first implemented in 2017, and has been shown to outperform ReLU in many instances [35].

If $\mu = 1$ is fixed, then $\text{Swish}(x) = x \cdot \sigma(x)$ is known as the SiLU function, i.e. the **Sigmoid Linear Unit**. Typically, if $\mu \neq 1$, then μ is actually set as a *trainable* parameter of the network, and so will be updated in the same manner as the weights and biases. Figure 70 shows $\text{Swish}(x)$ close to 0 for $\mu = 0.1$, $\mu = 1$ and $\mu = 10$, while figure 71 shows their derivatives.

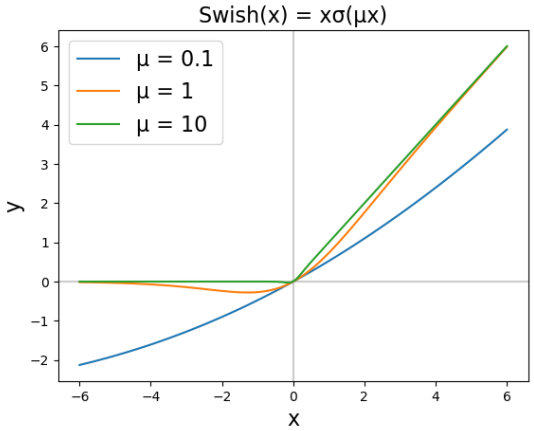

Figure 70: Activation function $\text{Swish}(x) = x \cdot \sigma(\mu x) = x/(1 + e^{-\mu x})$ for $\mu = 0.1$, $\mu = 1$ and $\mu = 10$.

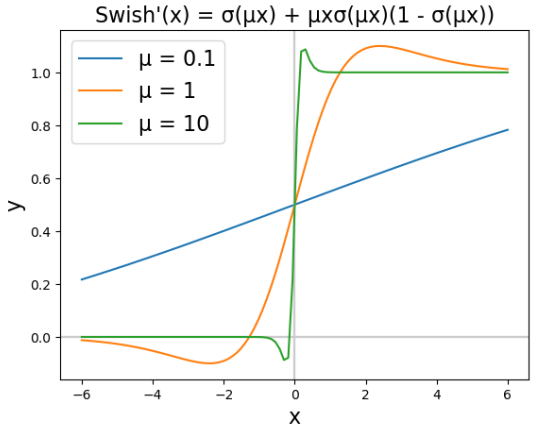

Figure 71: Derivative of $\text{Swish}(x)$, $\text{Swish}'(x) = \sigma(\mu x) + \mu \cdot x \cdot \sigma(\mu x) \cdot (1 - \sigma(\mu x))$ for $\mu = 0.1$, $\mu = 1$ and $\mu = 10$.

As $x \to -\infty$, $\text{Swish}(x) \to 0$ and $\text{Swish}'(x) \to 0$ for all values of μ. Meanwhile as x increases, $\text{Swish}(x)$ behaves more and more like $\text{Linear}(x) = x$. It is this behaviour which makes Swish a smooth imitation of ReLU. Notice this resemblance is stronger the larger μ is.

μ is typically set to 1, although sometimes set to be trainable. Interestingly, μ is often seen to converge to (or close to) 1 when it is set as a trainable parameter [35]. For clarity, we will use the term SiLU when $\mu = 1$ fixed, and Swish when μ is a trainable parameter. The Kaiming weight initialisation method, designed for ReLU, has also been shown to perform well when using Swish [35].

A third modification is to use 64-bit floats (i.e. doubles) instead of 32-bit ones (i.e. singles). 32-bit floats are the default for any objects (tensors or Modules) in PyTorch, since the precision of doubles is rarely required and increases computation time [20].

However, our sample training data $(x_j, t_j, u(x_j, t_j))$ are all stored as 64-bit floats. Since PyTorch does not typecast automatically, we must choose whether to convert our training data to singles for reduced computation (as is done in [33]) or store all our network parameters as doubles for increased precision. Since all values involved in our computations (i.e. x_j, t_j, $u(x_j, t_j)$ and θ) are small, the increased precision of 64-bit floats may be able to provide an improved solution.

7.4 Comparison of Various Methods

We then have 12 different methods for training our network: 2 data-types (32- or 64-bit), 2 variations on our cost function (minimising J_u and J_D simultaneously or separately) and 3 activation functions (tanh, SiLU and Swish). Fixing one randomly sampled data set of $2,000$ points, we train 12 neural networks for $100,000$ epochs, each with a different methodology. Networks using tanh all had the same initial weight- and bias-values, initialised according to Xavier initialisation and converted to 32-bit or 64-bit floats appropriately. The same was true for networks using SiLU or Swish, using Kaiming initialisation this time. The same initial values (randomly generated between 0 and 1) for λ and $\tilde{\nu}$ were used for all networks: approximately $\lambda = 0.0741$, $\tilde{\nu} = 0.483$.

The tables below compare the outputs of each network in terms of test solution accuracy and cost (J_u and J_D respectively, evaluated on all $25,600$ sample points), training time, and percentage error for λ and ν given by:

$$100 \cdot |\lambda - 1|, \qquad 100 \cdot \left| \frac{\nu - 0.01/\pi}{0.01/\pi} \right|.$$

Tables 8 and 9 show these results for networks using 32-bit floats, while tables 10 and 11 show them for networks using 64-bit floats.

Simultaneous Training	tanh	SiLU	Swish
λ Percentage Error	0.307	0.178	0.088
ν Percentage Error	4.972	0.334	0.367
Solution Inaccuracy $J_u(\theta)$	1.52×10^{-5}	1.65×10^{-5}	1.54×10^{-5}
Test Cost $J_D(\theta, \lambda, \nu)$	8.81×10^{-4}	7.73×10^{-5}	5.77×10^{-3}
Training Time (h)	3.71	3.43	3.75

Table 8: Comparison of 3 activation functions used to solve Burger's equation and approximate its parameters, using **32-bit floats** and trained minimising both cost functions **simultaneously**.

Separate Training	tanh	SiLU	Swish
λ Percentage Error	3.740	4.307	5.042
ν Percentage Error	7.091	4.022	2.243
Solution Inaccuracy $J_u(\theta)$	1.75×10^{-6}	1.41×10^{-6}	6.02×10^{-6}
Test Cost $J_D(\theta, \lambda, \nu)$	5.87×10^{-2}	7.42×10^{-2}	4.25×10^{-2}
Training Time (h)	2.43	2.46	2.49

Table 9: Comparison of 3 activation functions used to solve Burger's equation and approximate its parameters, using **32-bit floats** and trained minimising each cost function **separately**.

Simultaneous Training	tanh	SiLU	Swish
λ Percentage Error	0.315	0.017	0.004
ν Percentage Error	2.049	1.105	0.686
Solution Inaccuracy $J_u(\theta)$	1.08×10^{-5}	1.03×10^{-5}	1.06×10^{-5}
Test Cost $J_D(\theta, \lambda, \nu)$	4.02×10^{-3}	7.65×10^{-3}	7.73×10^{-3}
Training Time (h)	4.03	4.58	5.01

Table 10: Comparison of 3 activation functions used to solve Burger's equation and approximate its parameters, using **64-bit floats** and trained minimising both cost functions **simultaneously**.

Comparing our activation functions, trends are quite difficult to ascertain: for any fixed data-type and training method, none of the three functions outperforms the other two in reducing all four errors. Swish generally has an increased training time due to its extra trainable parameters,

Separate Training	tanh	SiLU	Swish
λ Percentage Error	1.286	0.788	1.500
ν Percentage Error	0.141	2.645	0.577
Solution Inaccuracy $J_u(\theta)$	1.11×10^{-6}	5.43×10^{-7}	2.08×10^{-6}
Test Cost $J_D(\theta, \lambda, \nu)$	1.25×10^{-2}	1.94×10^{-2}	2.67×10^{-2}
Training Time (h)	2.69	2.91	2.81

Table 11: Comparison of 3 activation functions used to solve Burger's equation and approximate its parameters, using **64-bit floats** and trained minimising each cost function **separately**.

but this is not consistently the case. Interestingly, the μ-values in the Swish function were never seen to deviate far from their initial values of $\mu = 1$; the observed value furthest from 1 was 2.04.

Their similar performance is reminiscent of the power the Universal Approximation Theorem, in that all that is required of our activation function is to be continuous and non-polynomial. Recall when comparing activation functions in section 3.3, inferior performance in other activation functions was due to smaller gradients, linearity or a lack of smoothness. Similarly, for the Lagaris method, we only required non-zero derivatives in order to satisfy the differential equation. None of our three activation functions here suffer from these issues, and hence have converged quickly to good approximations.

We can see that the use of doubles instead of singles consistently adds computation time to our training. Although it generally leads to a reduction in error for λ and ν, this is not consistently the case. The same can be said for a reduction in solution accuracy and test cost. Interestingly, for networks trained separately rather than simultaneously, using doubles provides a more significant reduction in error and a smaller increase in computation time. We will return to this observation shortly.

We see that training for both cost functions simultaneously generally produces smaller errors in λ and ν. The reason for this is that, referring to figure 69, these simultaneous networks identify approximately correct values for λ and ν within very few ($\approx 10,000$) epochs. Afterwards, although their values of λ and ν continue to become more accurate, we can conclude that the differential equation cost function $J_D(\theta, \lambda, \nu)$ is actually helping to improve our approximation of $u(x, t)$ (not just λ and ν). This is because the network is also able to learn information about the *derivatives* of $u(x, t)$ at our training points, not just the *value* of $u(x, t)$. Hence the final solution better satisfies the differential equation, and thus produces more accurate λ- and ν-values.

This is further evidenced by examining $J_D(\theta, \lambda, \nu)$ and $J_u(\theta)$ for networks trained separately on each cost function. These networks consistently provide a smaller solution inaccuracy $J_u(\theta)$ (since they are trained *only* to reduce this) but also a larger cost $J_D(\theta, \lambda, \nu)$. This is because they are not trained to identify the derivatives of $u(x, t)$. Given our finite number of training points (x_j, t_j, u_j), there are infinitely many functions for which $u(x_j, t_j) = u_j$ for all j, but only one which satisfies the differential equation, i.e. only one for which all *derivatives* also agree with our desired solution function.

Despite this advantage of simultaneous training, observe that its computation times are always longer. This is because evaluating $J_D(\theta, \lambda, \nu)$ is computationally expensive: it requires two backward passes of automatic differentiation to evaluate the partial derivatives of u. This additional computation every epoch uses significantly more time and memory, and hence we see slower training.

Meanwhile, when training separately, after sufficient training on $J_u(\theta)$, we can evaluate these partial derivatives only once, and repeatedly use these values to minimise $J_D(\theta, \lambda, \nu)$ in terms of λ and ν. This training for λ and ν is then incredibly quick: they converge to their final values within $35,000$ parameter updates, which actually takes less than ten seconds. Due to this method's reduced computation, using 64-bit floats adds less additional training time: observe that separate training with 64-bit floats is actually faster than simultaneous training with 32-bit floats. Hence we see that minimising the two cost functions separately and simultaneously both have their merits and drawbacks.

8 Conclusion

We have seen how neural networks (even in their simplest forms) are powerful tools for function approximation, and provide new methods to tackle centuries-old problems. Further to this, we have observed how modern machine learning libraries such as PyTorch simplify the implementation of neural networks, and hence make them an even more potent instrument.

However, we have also experienced their limitations. We have seen that many problems can arise, both of analytic and computational nature. The amount of choices available when using a neural network, from their basic structure to their training methodology, can make it difficult to achieve optimal results first time. Lessons learned in previous examples are not necessarily applicable to future scenarios; experimentation is always required to attain the best outcomes.

It should be noted that there are many aspects of neural networks which we have not explored in this paper. For example, we have considered only one type of neural network: fully-connected feedforward. Although other network types (e.g. convolutional, recurrent, modular) are typically used for problems of a different nature [2], it would be interesting to explore their capabilities when applied to differential equations. Furthermore, we have not fully explored the effect of varying the width and depth of a feedforward neural network. We have also focused only on gradient descent and its variants as optimisation algorithms, whereas there exist many other methods to minimise a given cost function. For example, second-order algorithms use the second derivatives of the cost function to improve convergence [2]. It would be interesting to see how these other methods compare when applied to differential equations.

Furthermore, when comparing the performance of different network features and training methods, we have measured only the cost / accuracy of our solutions, as well as their training time. For a deeper understanding of their performance, it would be useful to also explore the number of floating-point operations required, and the amount of memory used.

Finally, we conclude by noting how many features of this paper come from developments in very recent years. The methods discussed in sections 6 and 7, the Swish activation function, the Adam optimisation algorithm and even the Python library PyTorch itself would have all been unavailable to us just ten years ago. It is clear that neural networks and their application to differential equations are ever-expanding fields in which there is still much more to be learned.

Appendices

A Xavier Weight Initialisation

Since for our application to differential equations we typically use tanh, we describe a standard method known as **Xavier Initialisation** [10], which is designed for tanh.

We consider a network as in figure 1, except with $L-1$ hidden layers and where the width of the i^{th} layer is n_i, for $0 \leq i \leq L$. We also use the notation described below figure 1, letting $g = \tanh$ and defining $a^{[i]} = W^{[i]} z^{[i-1]} + b^{[i]}$. Let J denote the network's cost function.

Suppose, for each layer i of our network, the weights $W^{[i]}$ are taken independently and identically from a random distribution (**i.i.d.**) with mean zero. Then the variance of this distribution, $\text{Var}\left[W^{[i]}\right]$, determines the magnitude of the weights of the i^{th} layer. Our goal is to find an optimal value for $\text{Var}\left[W^{[i]}\right]$ for each i such that magnitude of all our gradients $\frac{\partial J}{\partial w}$ is the same, i.e. that $\text{Var}\left[\frac{\partial J}{\partial w}\right]$ is constant, for all weights w in our network.

Note that $W^{[i]}$ is of dimension $n_i \times n_{i-1}$. Let $1 \leq j \leq n_i$, $1 \leq k \leq n_{i-1}$. Then:

$$\begin{aligned} \frac{\partial J}{\partial W^{[i]}_{jk}} &= \frac{\partial J}{\partial a^{[i]}_j} \cdot \frac{\partial a^{[i]}_j}{\partial W^{[i]}_{jk}} \qquad \left(\text{since } a^{[i]}_j = \sum_{l=1}^{n_{i-1}} W^{[i]}_{jl} z^{[i-1]}_l + b^{[i]}_j\right). \\ &= \frac{\partial J}{\partial a^{[i]}_j} \cdot z^{[i-1]}_k \end{aligned}$$

Thus to understand $\text{Var}\left[\frac{\partial J}{\partial W^{[i]}}\right]$, we can examine $\text{Var}\left[\frac{\partial J}{\partial a^{[i]}}\right]$ and $\text{Var}\left[z^{[i-1]}\right]$. We require a few other assumptions:

1. The inputs, x, are i.i.d. and normalised (with mean zero).
2. The gradients $\frac{\partial J}{\partial a^{[L]}}$ are i.i.d. with mean zero.
3. $W^{[i]}$ and $z^{[i-1]}$ are mutually independent for all i.

Assumption 1 is not necessarily always true: for example, if our network is an image classifier then x is the pixel data of a given image. Neighbouring pixels are often more likely to be similar colours, and hence are not independent. Note for our applications to differential equations this assumption does hold however. Although assumption 2 is also not always true, it is needed to show why Xavier initialisation works in theory. Furthermore, the efficacy of Xavier initialisation has been verified in practice [10].

Meanwhile, assumption 3 is clearly true since $z^{[i-1]}$ does not depend on $W^{[i]}$ (on initialisation) and $W^{[i]}$ are initialised randomly.

We initialise all biases b to zero since, as mentioned previously, they have little impact on the resulting gradients. Also, with weights $W^{[i]}$ having mean zero and small enough variance, setting biases to zero means that $a^{[i]} = W^{[i]} z^{[i-1]} + b^{[i]}$ is close to zero. This results in $z^{[i]} = g(a^{[i]}) = \tanh\left(a^{[i]}\right) \approx a^{[i]}$ since $\tanh'(x) \approx 1$ for x close to zero (see figure 3). This (combined with the independence in assumption 3) also means that for all i, $\text{E}\left[a^{[i]}\right] = \text{E}\left[z^{[i]}\right] = \text{E}\left[x\right] = 0$.

Then we have:

$$\begin{aligned}\frac{\partial J}{\partial a_j^{[i]}} &= \sum_{l=1}^{n_{i+1}} \frac{\partial J}{\partial a_l^{[i+1]}} \cdot \frac{\partial a_l^{[i+1]}}{\partial a_j^{[i]}} \\ &= \sum_{l=1}^{n_{i+1}} \frac{\partial J}{\partial a_l^{[i+1]}} \cdot \frac{\partial}{\partial a_j^{[i]}} \left(\sum_{m=1}^{n_i} W_{lm}^{[i+1]} g(a_m^{[i]}) \right) \\ &= \sum_{l=1}^{n_{i+1}} \frac{\partial J}{\partial a_l^{[i+1]}} \cdot g'\left(a_j^{[i]}\right) W_{lj}^{[i+1]} \\ &= \sum_{l=1}^{n_{i+1}} \frac{\partial J}{\partial a_l^{[i+1]}} \cdot W_{lj}^{[i+1]}, \end{aligned} \tag{73}$$

since $g'\left(a_j^{[i]}\right) \approx 1$. Note that this also shows mutual independence of $\frac{\partial J}{\partial a^{[i]}}$ and $W^{[i]}$, as well as that of $\frac{\partial J}{\partial a^{[i]}}$ and $z^{[i-1]}$, since $\frac{\partial J}{\partial a^{[i]}}$ does not depend on $W^{[i]}$ or $z^{[i-1]}$ (and it is clear that $W^{[i]}$ and $z^{[i-1]}$ do not depend on $\frac{\partial J}{\partial a^{[i]}}$).

We can iteratively reapply the same steps to the gradients of higher layers to see that $\frac{\partial J}{\partial a_j^{[i]}}$ depends only on weights of higher layers and $\frac{\partial J}{\partial a^{[L]}}$. The independence from assumption 3 then allows us to conclude that for all i, $\mathrm{E}\left[\frac{\partial J}{\partial a^{[i]}}\right] = \mathrm{E}\left[\frac{\partial J}{\partial a^{[L]}}\right] = 0$ (by assumption 2).

To calculate the variance we require two results for independent random variables X, Y:

$$\mathrm{Var}[X+Y] = \mathrm{Var}[X] + \mathrm{Var}[Y], \tag{74}$$

$$\mathrm{Var}[XY] = \mathrm{E}[X]^2\mathrm{Var}[Y] + \mathrm{Var}[X]\mathrm{E}[Y]^2 + \mathrm{Var}[X]\mathrm{Var}[Y]. \tag{75}$$

These equations, along with the consequences of assumptions 2 and 3, give:

$$\begin{aligned}\mathrm{Var}\left[\frac{\partial J}{\partial a^{[i]}}\right] &= \mathrm{Var}\left[\frac{\partial J}{\partial a_j^{[i]}}\right] \\ &= \mathrm{Var}\left[\sum_{l=1}^{n_{i+1}} \frac{\partial J}{\partial a_l^{[i+1]}} \cdot W_{lj}^{[i+1]}\right] \\ &= \sum_{l=1}^{n_{i+1}} \mathrm{Var}\left[\frac{\partial J}{\partial a_l^{[i+1]}} \cdot W_{lj}^{[i+1]}\right] \\ &= \sum_{l=1}^{n_{i+1}} \mathrm{E}\left[\frac{\partial J}{\partial a_l^{[i+1]}}\right]^2 \mathrm{Var}\left[W_{lj}^{[i+1]}\right] + \mathrm{Var}\left[\frac{\partial J}{\partial a_l^{[i+1]}}\right] \mathrm{E}\left[W_{lj}^{[i+1]}\right]^2 \\ &\qquad + \mathrm{Var}\left[W_{lj}^{[i+1]}\right] \mathrm{Var}\left[\frac{\partial J}{\partial a_l^{[i+1]}}\right] \\ &= \sum_{l=1}^{n_{i+1}} \mathrm{Var}\left[W_{lj}^{[i+1]}\right] \mathrm{Var}\left[\frac{\partial J}{\partial a_l^{[i+1]}}\right] \quad \text{(since } W_{lj}^{[i+1]} \text{ and } \frac{\partial J}{\partial a_l^{[i+1]}} \text{ have mean zero)} \\ &= \sum_{l=1}^{n_{i+1}} \mathrm{Var}\left[W^{[i+1]}\right] \mathrm{Var}\left[\frac{\partial J}{\partial a^{[i+1]}}\right] \\ &= n_{i+1} \cdot \mathrm{Var}\left[W^{[i+1]}\right] \cdot \mathrm{Var}\left[\frac{\partial J}{\partial a^{[i+1]}}\right] \\ &= \dots \\ &= n_{i+1} \cdot \mathrm{Var}\left[W^{[i+1]}\right] \cdot n_{i+2} \cdot \mathrm{Var}\left[W^{[i+2]}\right] \cdot \dots \cdot n_L \cdot \mathrm{Var}\left[W^{[L]}\right] \cdot \mathrm{Var}\left[\frac{\partial J}{\partial a^{[L]}}\right] \\ &= \mathrm{Var}\left[\frac{\partial J}{\partial a^{[L]}}\right] \prod_{l=i+1}^{L} n_l \mathrm{Var}\left[W^{[l]}\right]. \end{aligned} \tag{76}$$

By steps analogous to those above, using assumptions 1 and 3 (shown in [36]), one can show that:

$$\text{Var}\left[z^{[i-1]}\right] = \text{Var}\left[z_k^{[i-1]}\right] = \text{Var}\left[x\right] \prod_{l=1}^{i-1} n_{l-1} \text{Var}\left[W^{[l]}\right]. \tag{77}$$

Then finally we have (by equation (75), the independence of $z_k^{[i-1]}$ and $\frac{\partial J}{\partial a_l^{[i+1]}}$, and the fact that both have mean zero):

$$\begin{aligned}
\text{Var}\left[\frac{\partial J}{\partial W^{[i]}}\right] &= \text{Var}\left[\frac{\partial J}{\partial W_{jk}^{[i]}}\right] \\
&= \text{Var}\left[\frac{\partial J}{\partial a_j^{[i]}} \cdot z_k^{[i-1]}\right] \\
&= \text{Var}\left[\frac{\partial J}{\partial a_j^{[i]}}\right] \text{Var}\left[z_k^{[i-1]}\right] \\
&= \text{Var}\left[\frac{\partial J}{\partial a^{[i]}}\right] \text{Var}\left[z^{[i-1]}\right] \\
&= \text{Var}\left[x\right] \cdot \text{Var}\left[\frac{\partial J}{\partial a^{[L]}}\right] \cdot \prod_{l=1}^{i-1} n_{l-1} \text{Var}\left[W^{[l]}\right] \cdot \prod_{l=i+1}^{L} n_l \text{Var}\left[W^{[l]}\right],
\end{aligned} \tag{78}$$

by combining equations (76) and (77).

To keep $\text{Var}\left[\frac{\partial J}{\partial W^{[i]}}\right]$ as constant as possible, we would like to keep $\text{Var}\left[\frac{\partial J}{\partial a^{[i]}}\right]$ and $\text{Var}\left[z^{[i-1]}\right]$ as constant as possible. This first condition would be achieved by setting $\text{Var}\left[W^{[l]}\right] = \frac{1}{n_l}$ for all l, while the second would require $\text{Var}\left[W^{[l]}\right] = \frac{1}{n_{l-1}}$ for all l. A sufficient compromise is thus found in setting:

$$\text{Var}\left[W^{[l]}\right] = \frac{2}{n_l + n_{l-1}} \quad \forall\, 1 \leq l \leq L. \tag{79}$$

Thus we can take, for example, $W^{[i]} \sim \mathcal{N}\left(0, \frac{2}{n_l + n_{l-1}}\right)$. Another common distribution to sample weights from is the Uniform distribution. For $X \sim U(-x, x)$, we have $\text{Var}[X] = \frac{1}{12}(2x)^2$. Combining this with equation (79) gives us:

$$W^{[i]} \sim U\left(-\sqrt{\frac{6}{n_l + n_{l-1}}}, \sqrt{\frac{6}{n_l + n_{l-1}}}\right). \tag{80}$$

Hence we have found an initialisation method to prevent exploding and vanishing gradients for the activation function tanh. Note that we required the approximate linearity of tanh close to zero, which is not a property common to many other (non-linear) activation functions.

B Finite-Difference Methods

We give a brief summary of the two finite-difference methods for solving ordinary differential equations mentioned in this paper. We give their details for the first-order case, but note that it is possible to generalise to ODEs of higher order, and also to PDEs.

The first method is attributed to **Euler** [5]. Suppose we have a differential equation:

$$f'(t) = D(t, f(t)),$$

for some unknown function $f : [t_0, t_f] \to \mathbb{R}$, where we have initial condition $f(t_0) = y_0$. Then, fixing a small step-size h, we can consider the Taylor expansion of f around t_0:

$$\begin{aligned} f(t_0 + h) &= f(t_0) + hf'(t_0) + \frac{1}{2}h^2 f''(t_0) + \frac{1}{3!}h^3 f'''(t_0) + \dots \\ &= f(t_0) + hf'(t_0) + O(h^2) \\ &\approx y_0 + h \cdot D(t_0, f(t_0)). \end{aligned}$$

Thus we can use the initial conditions and the differential equation to approximate f close to t_0. This can be repeatedly iteratively at points $t_0 + 2h = (t_0 + h) + h$, $t_0 + 3h = (t_0 + 2h) + h$, ... and so on, until we have approximated the values of f at a sufficient number of points in our interval $[t_0, t_f]$. Note the smaller h is, the smaller our error term $O(h^2)$. However, a smaller h will also require more iterations to cover the whole interval $[t_0, t_f]$, and hence more computation.

There are many developments and variations of this original method, one of the more popular of which is known as the **fourth-order Runge Kutta** method [5]. Its essential difference is that we use more evaluations of $D(t, f(t))$ in the interval $[t_0, t_0 + h]$ to obtain a better approximation of the gradient between $f(t_0)$ and $f(t_0 + h)$, and hence a better approximation of $f(t_0 + h)$. Specifically, we calculate:

$$\begin{aligned} k_1 &= D\left(t_0, f(t_0)\right), \\ k_2 &= D\left(t_0 + \frac{h}{2}, f(t_0) + h\frac{k_1}{2}\right), \\ k_3 &= D\left(t_0 + \frac{h}{2}, f(t_0) + h\frac{k_2}{2}\right), \\ k_4 &= D\left(t_0 + h, f(t_0) + hk_3\right), \end{aligned}$$

i.e. k_1 is the gradient of f at t_0, k_2 and k_3 are two different estimates of the gradient at $t_0 + h/2$, and k_4 is an estimate of the gradient at $t_0 + h$. We then average them in this way to approximate $f(t_0 + h)$:

$$f(t_0 + h) \approx \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

As before, we continue this process iteratively to approximate f across the whole interval.

Note that this requires more computation than the Euler method to obtain an approximation of $f(t_0 + h)$, however this comes with a reduced error. In fact, the error from this method can be shown to be $O(h^4)$ (from which it takes the name 'fourth-order'), instead of $O(h^2)$ as in the Euler method.

It is possible to compare the two methods using a larger step-size in the Runge-Kutta method (and hence fewer time-steps) such that the number of floating point operations are the same. It can then be shown that in this scenario, the Runge-Kutta method still provides a better approximation due to its smaller error bound, in some examples [30].

We should note at this point the key differences between these finite-difference methods and the Lagaris method we use throughout the paper (see section 5.1). Most fundamentally, the Euler and Runge-Kutta methods provide a numerical approximation of the value of the solution function f for given values of t. The Lagaris method, on the other hand, provides an approximation *function*, in the form of a neural network output. This approximation function comes with a closed-form expression, and expressions for its derivatives can be explicitly calculated.

Furthermore, the finite-difference methods are dependent on time-step iteration: to evaluate $f(t)$, we must evaluate f at t_0, $t_0 + h$, $t_0 + 2h$, ... etc., until we reach t. Using a neural network, however, we can evaluate the solution function at any t value within our training domain with equal computational cost. However, the additional computational expense of training our neural network should also be noted.

C Code Samples

Here we include extracts of the code used implement the examples discussed. All files are available in [8].

```
5   class DataSet(torch.utils.data.Dataset):
6       """
7       An object which generates the x values for the input node
8       and the corresponding true values y=f(x) of the output node.
9       """
10      def __init__(self, fcn, xRange, numSamples):
11          """
12          Arguments:
13          fcn (function) -- function to be approximated
14          xRange (list of length 2) -- lower and upper limits for input values x
15          numSamples (int) -- number of training data samples
16
17          Returns:
18          DataSet object with two attributes:
19              dataIn (PyTorch tensor of shape (numSamples,1)) -- 'numSamples'
20                  evenly-spaced data points from xRange[0] to xRange[1]
21              dataOut (PyTorch tensor of shape (numSamples,1)) -- corresponding
22                  values of function at these points
23          """
24          self.dataIn  = torch.linspace(xRange[0], xRange[1], numSamples).view(-1,1)
25          self.dataOut = fcn(self.dataIn).view(-1,1)
26          # 'view' method reshapes tensors, in this case into column vectors
27
28      def __len__(self):
29          """
30          Arguments:
31          None
32
33          Returns:
34          len(self.dataIn) (int) -- number of training data points
35          """
36          return len(self.dataIn)
37
38      def __getitem__(self, idx):
39          """
40          Used by DataLoader object to retrieve training data points
41
42          Arguments:
43          idx (int) -- index of data point required
44
45          Returns:
46          (x,f(x)) (tuple of 1x1 tensors) -- (x,f(x)) pair at index idx
47          """
48          return (self.dataIn[idx], self.dataOut[idx])
```

Figure C.1: Creating a data set as a subclass of the Dataset class to approximate $\sin(x)$.

```
class Fitter(torch.nn.Module):
    """
    The neural network object, with 1 node in the input layer,
    1 node in the output layer, and 1 hidden layer with 'numHiddenNodes' nodes.
    """
    def __init__(self, numHiddenNodes):
        """
        Arguments:
        numHiddenNodes (int) -- number of nodes in hidden layer

        Returns:
        Fitter object (neural network) with two attributes:
        fc1 (fully connected layer) -- linear transformation of hidden layer
        fc2 (fully connected layer) -- linear transformation of outer layer
        """
        super(Fitter, self).__init__()
        self.fc1 = torch.nn.Linear(in_features = 1, out_features = numHiddenNodes)
        self.fc2 = torch.nn.Linear(in_features = numHiddenNodes, out_features = 1)

    def forward(self, x):
        """
        Function which connects inputs to outputs in the neural network.

        Arguments:
        x (PyTorch tensor shape (batchSize,1)) -- input of neural network

        Returns:
        y (PyTorch tensor shape (batchSize,1)) -- output of neural network
        """
        # tanh activation function used on hidden layer
        h = torch.tanh(self.fc1(x))
        # Linear activation function used on outer layer
        y = self.fc2(h)
        return y
```

Figure C.2: Creating a basic neural network with one hidden layer as a subclass of the Module class.

```
86  def train(network, loader, lossFn, optimiser, numEpochs):
87      """
88      A function to train a network.
89
90      Arguments:
91      network (Module) -- the neural network
92      loader (DataLoader) -- generates batches from the training dataset
93      lossFn (Loss Function) -- network's loss function
94      optimiser (Optimiser) -- carries out parameter optimisation
95      numEpochs (int) -- number of training epochs
96
97      Returns:
98      costList (list of length 'numEpochs') -- cost values of all epochs
99      """
100     costList = []
101     network.train(True) # set module in training mode
102     for epoch in range(numEpochs):
103         for batch in loader:
104             x, y = batch[0], batch[1] # x and y=f(x) values
105             yOut = network.forward(x) # network output, prediction of y
106             cost  = lossFn(yOut, y) # calculate cost value
107             cost.backward() # back propagation, calculate gradients
108             optimiser.step() # perform parameter optimisation
109             optimiser.zero_grad() # reset gradients to zero
110         costList.append(cost.item()) # store cost of each epoch
111     network.train(False) # set module out of training mode
112     return costList
113
114 fcn         = torch.sin
115 numEpochs   = 10000
116
117 network     = Fitter(numHiddenNodes=16)
118 trainSet    = DataSet(fcn, xRange = [-3,3], numSamples=30)
119 loader      = torch.utils.data.DataLoader(dataset=trainSet, batch_size=30)
120 lossFn      = torch.nn.MSELoss() # mean-squared error loss
121 optimiser   = torch.optim.SGD(network.parameters(), lr=1e-2)
122 # gradient descent algorithm (batch_size in loader determines
123 # type (i.e. batch, mini-batch, stochastic))
124
125 costList    = train(network, loader, lossFn, optimiser, numEpochs)
```

Figure C.3: Training a network to approximate the function $\sin(x)$.

```python
class DataSet(torch.utils.data.Dataset):
    """
    An object which generates the x values for the input node
    """
    def __init__(self, numSamples, xRange):
        """
        Arguments:
        xRange (list of length 2) -- lower and upper limits for input values x
        numSamples (int) -- number of training data samples

        Returns:
        DataSet object with one attributes:
            dataIn (PyTorch tensor of shape (numSamples,1)) -- 'numSamples'
                evenly-spaced data points from xRange[0] to xRange[1]
        """
        self.dataIn  = torch.linspace(xRange[0], xRange[1], numSamples, requires_grad=True).view(-1,1)
        # 'view' method reshapes tensors, in this case into a column vector

    def __len__(self):
        """
        Arguments:
        None

        Returns:
        len(self.dataIn) (int) -- number of training data points
        """
        return len(self.dataIn)

    def __getitem__(self, idx):
        """
        Used by DataLoader object to retrieve training data points

        Arguments:
        idx (int) -- index of data point required

        Returns:
        x (tensor shape (1,1)) -- data point at index 'idx'
        """
        return self.dataIn[idx]
```

Figure C.4: Creating a data set of evenly-spaced points across an interval.

```python
def trialFunc1(x, n_out):
    """
    Trial solution to Lagaris problem 1: f(x) = 1 + xN(x)
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        n_out (tensor of shape (batchSize,1)) -- output of neural network
    Returns:
        1 + x * n_out (tensor of shape (batchSize,1)) -- trial solution to differential equation
    """
    return 1 + x * n_out

def dTrialFunc1(x, n_out, dndx):
    """
    Derivative of trial solution to Lagaris problem 1: f'(x) = N(x) + xN'(x)
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        n_out (tensor of shape (batchSize,1)) -- output of neural network
        dndx (tensor of shape (batchSize,1)) -- derivative of n_out w.r.t. x
    Returns:
        n_out + x * dndx (tensor of shape (batchSize,1)) -- derivative of trial function w.r.t. x
    """
    return n_out + x * dndx

def diffEq1(x, f_trial, df_trial):
    """
    Returns D(x) of differential equation D(x) = 0 from Lagaris problem 1
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        f_trial (tensor of shape (batchSize,1)) -- trial solution at x
        df_trial (tensor of shape (batchSize,1)) -- derivative of trial solution at x
    Returns:
        LHS - RHS (tensor of shape (batchSize,1)) -- differential equation evaluated at x"""
    RHS = x**3 + 2*x + (x**2 * ((1+3*x**2) / (1 + x + x**3)))
    LHS = df_trial + ((x + (1+3*(x**2)) / (1 + x + x**3) ) * f_trial)
    return LHS - RHS

def solution1(x):
    """
    Analytic solution to Lagaris problem 1
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
    Returns:
        y (tensor of shape (batchSize,1)) -- analytic solution of differential equation at x"""
    y =  (torch.exp(-(x**2)/2) / (1 + x + x**3))+ x**2
    return y
```

Figure C.5: Defining functions needed to solve example 1 from the Lagaris paper (i.e. the trial solution, the trial solution's derivative, the differential equation applied to the trial solution, and the exact solution for comparison).

```
def train(network, loader, lossFn, optimiser, numEpochs):
    """
    A function to train a neural network to solve a
    first-order ODE with Dirichlet boundary conditions.

    Arguments:
    network (Module) -- the neural network
    loader (DataLoader) -- generates batches from the training dataset
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    numEpochs (int) -- number of training epochs

    Returns:
    costList (list of length 'numEpochs') -- cost values of all epochs
    """
    cost_list=[]
    network.train(True) # set module in training mode
    for epoch in range(numEpochs):
        for batch in loader:
            n_out = network.forward(batch) # network output

            # Get derivative of the network output w.r.t. the input values:
            dndx = grad(n_out, batch, torch.ones_like(n_out), retain_graph=True)[0]
            # torch.ones_like(x) creates a tensor the same shape as x, filled with 1's

            # Get value of trial solution f(x)
            f_trial = trialFunc(batch, n_out)
            # Get df / dx
            df_trial = dTrialFunc(batch, n_out, dndx)
            # Get LHS of differential equation D(x) = 0
            diff_eq = diffEq(batch, f_trial, df_trial)

            cost = lossFn(diff_eq, torch.zeros_like(diff_eq)) # calculate cost
            # torch.zeros_like(x) creates a tensor the same shape as x, filled with 0's
            cost.backward() # perform backpropagation
            optimiser.step() # perform parameter optimisation
            optimiser.zero_grad() # reset gradients to zero

        cost_list.append(cost.detach().numpy())# store cost of each epoch
    network.train(False) # set module out of training mode
    return cost_list
```

Figure C.6: Training a network to solve example 1 from the Lagaris paper.

```
try: # load saved network if possible
    checkpoint = torch.load('problem1.pth')
    # checkpoint = torch.load('problem2.pth')
    network    = checkpoint['network']
except: # create new network
    network    = Fitter(numHiddenNodes=10)
    checkpoint = {'network': network}
    torch.save(checkpoint, 'problem1.pth')
    # torch.save(checkpoint, 'problem2.pth')
xRange       = [0, 2]
numSamples   = 20
batchSize    = 20
paramUpdates = 20000
train_set    = DataSet(numSamples, xRange)
train_loader = torch.utils.data.DataLoader(dataset=train_set, batch_size=batchSize, shuffle=True)
lossFn       = torch.nn.MSELoss()
optimiser    = torch.optim.SGD(network.parameters(), lr=1e-3)

solution = solution1
trialFunc = trialFunc1
dTrialFunc = dTrialFunc1
diffEq = diffEq1

# solution = solution2
# trialFunc = trialFunc2
# dTrialFunc = dTrialFunc2
# diffEq = diffEq2

costList = []
epoch = 0
numEpochs = 1000
totalEpochs = int((batchSize/numSamples) * paramUpdates)
start = time.time()
while epoch < totalEpochs:
    costList.extend(train(network, train_loader, lossFn, optimiser, numEpochs))
    epoch += numEpochs
end = time.time()
```

Figure C.7: Code to carry out 20,000 parameter updates and measure the time taken when training to solve example 1 from the Lagaris paper. By creating a model and saving its initial state, we guarantee that the three networks we wish to compare have the same initial parameter values. Note that uncommenting the commented lines will train to solve example 2 instead.

```
114  def trialFunc2(x, n_out):
115      """
116      Trial solution to Lagaris problem 2: f(x) = x * N(x)
117      Arguments:
118          x (tensor of shape (batchSize,1)) -- input of neural network
119          n_out (tensor of shape (batchSize,1)) -- output of neural network
120      Returns:
121          x * n_out (tensor of shape (batchSize,1)) -- trial solution to differential equation
122      """
123      return x * n_out
124
125  def dTrialFunc2(x, n_out, dndx):
126      """
127      Derivative of trial solution to Lagaris problem 2: f'(x) = N(x) + x * N'(x)
128      Arguments:
129          x (tensor of shape (batchSize,1)) -- input of neural network
130          n_out (tensor of shape (batchSize,1)) -- output of neural network
131          dndx (tensor of shape (batchSize,1)) -- derivative of n_out w.r.t. x
132      Returns:
133          n_out + x * dndx (tensor of shape (batchSize,1)) -- derivative of trial function w.r.t. x
134      """
135      return n_out + x * dndx
136
137  def diffEq2(x, f_trial, df_trial):
138      """
139      Returns D(x) of differential equation D(x) = 0 from Lagaris problem 2
140      Arguments:
141          x (tensor of shape (batchSize,1)) -- input of neural network
142          f_trial (tensor of shape (batchSize,1)) -- trial solution at x
143          df_trial (tensor of shape (batchSize,1)) -- derivative of trial solution at x
144      Returns:
145          LHS - RHS (tensor of shape (batchSize,1)) -- differential equation evaluated at x"""
146      RHS = df_trial + (1/5)*f_trial
147      LHS = torch.exp(-x/5) * torch.cos(x)
148      return LHS - RHS
149
150  def solution2(x):
151      """
152      Analytic solution to Lagaris problem 1
153      Arguments:
154          x (tensor of shape (batchSize,1)) -- input of neural network
155      Returns:
156          y (tensor of shape (batchSize,1)) -- analytic solution of differential equation at x"""
157      y = torch.exp(-x/5) * torch.sin(x)
158      return y
```

Figure C.8: Defining functions needed to solve example 2 from the Lagaris paper (i.e. the trial solution, the trial solution's derivative, the differential equation applied to the trial solution, and the exact solution for comparison).

```
209  try: # load saved network (initial state) and dictionary containing cost lists, if possible
210      checkpoint = torch.load('problem2.pth')
211      network    = checkpoint['network']
212      costsDict  = torch.load('problem2Costs.pth')
213  except: # create new network and new dictionary to store cost lists
214      network    = Fitter(numHiddenNodes=10)
215      checkpoint = {'network': network}
216      torch.save(checkpoint, 'problem2.pth')
217      costsDict  = {}
218  xRange       = [0, 10]
219  numSamples   = 50
220  batchSize    = 50
221  train_set    = DataSet(numSamples, xRange)
222  train_loader = torch.utils.data.DataLoader(dataset=train_set, batch_size=batchSize, shuffle=True)
223  lossFn       = torch.nn.MSELoss()
224
225  # algorithm    = "Batch Gradient Descent"
226  # optimiser    = torch.optim.SGD(network.parameters(), lr=1e-3)
227
228  # algorithm    = "Gradient Descent with Momentum"
229  # optimiser    = torch.optim.SGD(network.parameters(), lr=1e-3, momentum = 0.9, dampening = 0.9)
230
231  algorithm    = "RProp"
232  optimiser    = torch.optim.Rprop(network.parameters(), lr=1e-3)
233
234  # algorithm    = "RMSProp"
235  # optimiser    = torch.optim.RMSprop(network.parameters(), lr=1e-3)
236
237  # algorithm    = "Adam"
238  # optimiser    = torch.optim.Adam(network.parameters(), lr=1e-3)
239
240  costList    = []
241  epoch       = 0
242  numEpochs   = 1000
243  totalEpochs = 20000
244
245  start = time.time()
246  while epoch < totalEpochs:
247      costList.extend(train(network, train_loader, lossFn, optimiser, numEpochs))
248      epoch += numEpochs
249  end = time.time()
250
251  costsDict[algorithm] = costList # store cost list for each algorithm in a dictionary
252  torch.save(costsDict, 'problem2Costs.pth') # save dictionary
```

Figure C.9: Code to carry out 20,000 epochs and measure the time taken when training to solve example 2 from the Lagaris paper. By creating a model and saving its initial state, we guarantee that the five networks we wish to compare have the same initial parameter values. We can test each optimisation algorithm by uncommenting the appropriate lines of code.

Note that to implement gradient descent with momentum in line 229 above, we also specify a value for 'dampening'. This is because PyTorch implements a more general version of momentum, in which we fix two parameters $\beta, \gamma \in [0, 1)$ known respectively as the momentum and dampening constants. Then the first line of algorithm 2 is $m_\theta \leftarrow \beta m_\theta + (1 - \gamma)\frac{\partial J}{\partial \theta}$. In line 229 we set $\beta = \gamma = 0.9$, and recover our version given in algorithm 2.

```python
def trialFunc(x, n_out):
    """
    Trial solution to Lagaris problem 3: f(x) = x + x^2 * N(x)
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        n_out (tensor of shape (batchSize,1)) -- output of neural network
    Returns:
        x + (x**2 * n_out) (tensor of shape (batchSize,1)) -- trial solution to differential equation"""
    return x + (x**2 * n_out)

def dTrialFunc(x, n_out, dndx):
    """
    First derivative of trial solution to Lagaris problem 3: f'(x) = 1 + 2xN(x) + x^2 * N'(x)
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        n_out (tensor of shape (batchSize,1)) -- output of neural network
        dndx (tensor of shape (batchSize,1)) -- derivative of n_out w.r.t. x
    Returns:
        1 + (2*x*n_out) + (x**2 * dndx) (tensor of shape (batchSize,1)) -- 1st derivative of trial function w.r.t. x)"""
    return 1 + (2*x*n_out) + (x**2 * dndx)

def d2TrialFunc(x,n_out,dndx,d2ndx2):
    """
    Second derivative of trial solution to Lagaris problem 3: f''(x) = 2N(x) + (4x * N'(x)) + x^2 N''(x)
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        n_out (tensor of shape (batchSize,1)) -- output of neural network
        dndx (tensor of shape (batchSize,1)) -- 1st derivative of n_out w.r.t. x
        d2ndx2 (tensor of shape (batchSize,1)) -- 2nd derivative of n_out w.r.t. x
    Returns:
        2*n_out + (4*x*dndx) + (x**2 * d2ndx2) (tensor of shape (batchSize,1)) -- 2nd derivative of trial function w.r.t. x)"""
    return 2*n_out + (4*x*dndx) + (x**2 * d2ndx2)

def diffEq(x, f_trial, df_trial, d2f_trial):
    """
    Returns D(x) of differential equation D(x) = 0 from Lagaris problem 3
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
        f_trial (tensor of shape (batchSize,1)) -- trial solution at x
        df_trial (tensor of shape (batchSize,1)) -- 1st derivative of trial solution at x
        d2f_trial (tensor of shape (batchSize,1)) -- 2nd derivative of trial solution at x
    Returns:
        LHS - RHS (tensor of shape (batchSize,1)) -- differential equation evaluated at x"""
    LHS = d2f_trial + (1/5)*df_trial + f_trial
    RHS = -(1/5) * torch.exp(-x/5) * torch.cos(x)
    return LHS - RHS

def solution(x):
    """
    Analytic solution to Lagaris problem 3
    Arguments:
        x (tensor of shape (batchSize,1)) -- input of neural network
    Returns:
        torch.exp(-x/5) * torch.sin(x) (tensor of shape (batchSize,1)) -- analytic solution of differential equation at x"""
    return torch.exp(-x/5) * torch.sin(x)
```

Figure C.10: Defining functions needed to solve example 3 from the Lagaris paper (i.e. the trial solution, the trial solution's first and second derivatives, the differential equation applied to the trial solution, and the exact solution for comparison).

```python
def train(network, loader, lossFn, optimiser, numEpochs):
    """
    A function to train a neural network to solve a
    second-order ODE with Cauchy boundary conditions.

    Arguments:
    network (Module) -- the neural network
    loader (DataLoader) -- generates batches from the training dataset
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    numEpochs (int) -- number of training epochs

    Returns:
    cost_list (list of length 'numEpochs') -- cost values of all epochs
    """
    cost_list=[]
    network.train(True)
    for epoch in range(numEpochs):
        for batch in loader:
            n_out = network(batch)

            # Get first derivative of the network output with respect to the input values:
            dndx = grad(n_out, batch, torch.ones_like(n_out), retain_graph=True, create_graph=True)[0]
            # Get second derivative of the network output with respect to the input values:
            d2ndx2 =grad(dndx, batch, torch.ones_like(dndx), retain_graph=True)[0]

            # Get value of trial solution f(x)
            f_trial = trialFunc(batch, n_out)
            # Get f'(x)
            df_trial = dTrialFunc(batch, n_out, dndx)
            # Get f''(x)
            d2f_trial = d2TrialFunc(batch,n_out,dndx,d2ndx2)
            # Get LHS of differential equation D(x) = 0
            diff_eq = diffEq(batch, f_trial, df_trial, d2f_trial)

            cost = lossFn(diff_eq, torch.zeros_like(diff_eq)) # calculate cost
            # torch.zeros_like(x) creates a tensor the same shape as x, filled with 0's
            cost.backward() # perform backpropagation
            optimiser.step() # perform parameter optimisation
            optimiser.zero_grad() # reset gradients to zero

        cost_list.append(cost.detach().numpy())# store cost of each epoch
    network.train(False)
    return cost_list
```

Figure C.11: Training a network to solve example 3 from the Lagaris paper.

```
try: # load saved network and cost list, if possible
    checkpoint = torch.load('problem3InitialNetwork.pth')
    network    = checkpoint['network']
    costList   = checkpoint['costList']
except: # create new network and cost list
    network     = Fitter(numHiddenNodes=10)
    costList    = []
    checkpoint  = {'network': network,
                   'costList': costList}
    torch.save(checkpoint, 'problem3InitialNetwork.pth')

networkName  = 'Network1'
# networkName  = 'Network2'

xRange       = [0, 10]
numSamples   = 50
batchSize    = 50
trainData    = DataSet(numSamples, xRange)
trainLoader  = torch.utils.data.DataLoader(dataset=trainData, batch_size=batchSize, shuffle=True)

xRangeWide       = [-5, 15]
numSamplesWide   = 100
batchSizeWide    = 100
trainDataWide    = DataSet(numSamplesWide, xRangeWide)
trainLoaderWide  = torch.utils.data.DataLoader(dataset=trainDataWide, batch_size=batchSizeWide, shuffle=True)

lossFn      = torch.nn.MSELoss()
optimiser   = torch.optim.Adam(network.parameters(), lr=1e-3)
epoch       = 0
numEpochs   = 1000
totalEpochs = 40000

start = time.time()
while epoch < totalEpochs:
    if networkName == 'network1' and epoch < 20000:
        costList.extend(train(network, trainLoader, lossFn, optimiser, numEpochs))
    else:
        costList.extend(train(network, trainLoaderWide, lossFn, optimiser, numEpochs))
    epoch += numEpochs
end = time.time()

checkpoint  = {'network': network,
               'costList': costList,}
torch.save(checkpoint, 'problem3' + networkName + '.pth')
```

Figure C.12: Code to carry out 40,000 epochs of training to solve example 3 from the Lagaris paper. By creating a model and saving its initial state, we guarantee that the two networks we wish to compare have the same initial parameter values. Network 1 is trained on [0,10] for 20,000 epochs, and then on [-5,15] for a further 20,000. Network 2 was trained only on [-5,15] for 40,000 epochs.

```
def train(network, loader, lossFn, optimiser, numEpochs):
    """
    A function to train a neural network to solve a system of two first-order ODEs with Dirichlet boundary conditions.

    Arguments:
    network (Module) -- the neural network
    loader (DataLoader) -- generates batches from the training dataset
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    numEpochs (int) -- number of training epochs

    Returns:
    cost_list (list of length 'numEpochs') -- cost values of all epochs """
    cost_list=[]
    network.train(True) # set module in training mode
    for epoch in range(numEpochs):
        for batch in loader:
            n_out = network.forward(batch)

            # Separate two columns of output (one for f1, one for f2)
            # Using torch.split retains tensor history for autograd
            n1_out, n2_out = torch.split(n_out, split_size_or_sections=1, dim=1)

            # Get the derivative of both networks' outputs with respect to the input values.
            dn1dx = grad(n1_out, batch, torch.ones_like(n1_out), retain_graph=True, create_graph=True)[0]
            dn2dx = grad(n2_out, batch, torch.ones_like(n2_out), retain_graph=True, create_graph=True)[0]
            # Get value of trial solutions f1(x), f2(x)
            f1_trial = f1Trial(batch, n1_out)
            f2_trial = f2Trial(batch, n2_out)
            # Get f1'(x) and f2'(x)
            df1_trial = df1Trial(batch, n1_out, dn1dx)
            df2_trial = df2Trial(batch, n2_out, dn2dx)
            # Get LHS of differential equations D1(x) = 0, D2(x) = 0
            D1 = diffEq1(batch, f1_trial, f2_trial, df1_trial)
            D2 = diffEq2(batch, f1_trial, f2_trial, df2_trial)

            # Calculate and store cost
            cost1 = lossFn(D1, torch.zeros_like(D1))
            cost2 = lossFn(D2, torch.zeros_like(D2))
            cost = cost1 + cost2

            cost.backward() # perform backpropagation
            optimiser.step() # perform parameter optimisation
            optimiser.zero_grad() # reset gradients to zero

        cost_list.append(cost.detach().numpy()) # store cost of each epoch
    network.train(False) # set module out of training mode
    return cost_list
```

Figure C.13: Training a network to solve example 4 from the Lagaris paper.

```
try: # load saved network if possible
    checkpoint = torch.load('problem4InitialNetwork.pth')
    network    = checkpoint['network']
except: # create new network
    network    = DESolver(numHiddenNodes=16)
    checkpoint = {'network': network}
    torch.save(checkpoint, 'problem4InitialNetwork.pth')
lossFn      = torch.nn.MSELoss()
optimiser   = torch.optim.Adam(network.parameters(), lr = 1e-3)
totalXRange      = [0,3]
numTotalSamples  = 30

###### TRAINING ON SUBINTERVALS OF INCREASING SIZE, STARTING FROM 0
ranges = [[0,3]]  # fails
# ranges = [[0,1.5], [0,3]]  # succeeds
# ranges = [[0,1],[0,2],[0,3]]  # succeeds
###### TRAINING ON NON-INTERSECTING SUBINTERVALS, STARTING FROM 0
# ranges = [[0,1.5], [1.5,3]] # succeeds but suboptimal solution
# ranges = [[0,1],[1,2],[2,3]] # succeeds but suboptimal solution
###### TRAINING ON NON-INTERSECTING SUBINTERVALS, THEN FULL INTERVAL, STARTING FROM 0
# ranges = [[0,1.5], [1.5,3], [0,3]] # succeeds
# ranges = [[0,1],[1,2],[2,3],[0,3]] # succeeds
###### TRAINING ON SUBINTERVALS OF INCREASING SIZE, STARTING FROM 3
# ranges = [[1.5,3], [0,3]]  # fails
# ranges = [[2,3],[1,3],[0,3]]  # fails
# ranges = [[2.25,3],[1.5,3],[0.75,3],[0,3]] # fails
###### TRAINING ON NON-INTERSECTING SUBINTERVALS, THEN FULL INTERVAL, STARTING FROM 3
# ranges = [[1.5,3], [0,1.5], [0,3]] # succeeds
# ranges = [[2,3],[1,2],[0,1],[0,3]] # succeeds
###### TRAINING ON NON-INTERSECTING SUBINTERVALS, STARTING FROM 3
# ranges = [[1.5,3], [0,1.5]] # succeeds
# ranges = [[2,3],[1,2],[0,1]] # succeeds but suboptimal solution

costList = []
epoch = 0
numEpochs = 1000
totalEpochs = 36000
epochsPerSubRange = int(totalEpochs / len(ranges))
for subRange in ranges:
    epochCounter = 0
    numSamples = int(10 * (subRange[1] - subRange[0]))
    trainData    = DataSet(numSamples, subRange)
    trainLoader = torch.utils.data.DataLoader(dataset=trainData, batch_size=int(numSamples), shuffle=True)

    while epochCounter < epochsPerSubRange:
        costList.extend(train(network, trainLoader, lossFn, optimiser, numEpochs))
        epoch += numEpochs
        epochCounter += numEpochs
```

Figure C.14: Code to carry out 36,000 epochs of training to solve example 4 from the Lagaris paper. By creating a model and saving its initial state, we guarantee that the 14 networks we wish to compare have the same initial parameter values. Each network is trained on a curriculum of sub-intervals, where the total 36,000 epochs are divided equally between these sub-intervals.

```
class PDESolver(torch.nn.Module):
    """
    The neural network object, with 2 nodes in the input layer,
    1 node in the output layer, and 1 hidden layer with 'numHiddenNodes' nodes.
    """
    def __init__(self, numHiddenNodes):
        """
        Arguments:
        numHiddenNodes (int) -- number of nodes in hidden layer

        Returns:
        PDESolver object (neural network) with two attributes:
        fc1 (fully connected layer) -- linear transformation of hidden layer
        fc2 (fully connected layer) -- linear transformation of outer layer
        """
        super(PDESolver, self).__init__()
        self.fc1 = torch.nn.Linear(in_features = 2, out_features = numHiddenNodes)
        self.fc2 = torch.nn.Linear(in_features = numHiddenNodes, out_features = 1)

    def forward(self, input):
        """
        Function which connects inputs to outputs in the neural network.

        Arguments:
        input (PyTorch tensor shape (batchSize,2)) -- input of neural network

        Returns:
        z (PyTorch tensor shape (batchSize,1)) -- output of neural network
        """
        # tanh activation function used on hidden layer
        h = torch.tanh(self.fc1(input))
        # Linear activation function used on outer layer
        z = self.fc2(h)
        return z
```

Figure C.15: Creating a basic neural network with two inputs, one output and one hidden layer as a subclass of the Module class.

```python
class DataSet(torch.utils.data.Dataset):
    """
    An object which generates the (x,y) values for the input node
    """
    def __init__(self, xRange, yRange, numSamples):
        """
        Arguments:
        xRange (list of length 2) -- lower and upper limits for input values x
        yRange (list of length 2) -- lower and upper limits for input values y
        numSamples (int) -- number of training data samples along each axis

        Returns:
        DataSet object with one attributes:
            dataIn (PyTorch tensor of shape (numSamples^2,2)) -- 'numSamples'^2
                evenly-spaced grid points from (xRange[0], yRange[0]) to (xRange[1], yRange[1])
        """
        X  = torch.linspace(xRange[0],xRange[1],numSamples, requires_grad=True)
        Y  = torch.linspace(yRange[0],yRange[1],numSamples, requires_grad=True)
        grid = torch.meshgrid(X,Y)
        # meshgrid takes Cartesian product of tensors X and Y, returns a tuple of tensors
        # (x-values, y-values) each of shape (numSamples, numSamples)
        self.data_in = torch.cat((grid[0].reshape(-1,1),grid[1].reshape(-1,1)),1)
        # reshape data into shape (numSamples^2,2)

    def __len__(self):
        """
        Arguments:
        None
        Returns:
        len(self.dataIn) (int) -- number of training data points
        """
        return self.data_in.shape[0]

    def __getitem__(self, i):
        """
        Used by DataLoader object to retrieve training data points

        Arguments:
        idx (int) -- index of data point required

        Returns:
        [x,y] (tensor shape (1,2)) -- data point at index 'idx'
        """
        return self.data_in[i]
```

Figure C.16: Creating an even lattice of points across the interval $[0,1] \times [0,1]$.

```
def train(network, loader, lossFn, optimiser, numEpochs):
    """
    A function to train a neural network to solve a
    2-dimensional PDE with Dirichlet boundary conditions

    Arguments:
    network (Module) -- the neural network
    loader (DataLoader) -- generates batches from the training dataset
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    numEpochs (int) -- number of training epochs

    Returns:
    cost_list (list of length 'numEpochs') -- cost values of all epochs
    """
    cost_list=[]
    network.train(True)
    for epoch in range(numEpochs):
        for batch in loader:
            n_out = network(batch) # network output

            dn  = grad(n_out, batch, torch.ones_like(n_out), retain_graph=True, create_graph=True)[0]
            dn2 = grad(dn, batch, torch.ones_like(dn), retain_graph=True, create_graph=True)[0]

            # Get first partial derivatives of neural network output: n_x , n_y
            n_x, n_y = torch.split(dn, split_size_or_sections=1, dim=1)
            # Get second derivatives of neural network output: n_xx,  n_yy
            n_xx, n_yy = torch.split(dn2, split_size_or_sections=1, dim=1)

            x, y = torch.split(batch, 1, dim=1) # separate batch into x- and y-values
            # Get second derivatives of trial solution: f_{xx}(x,y) and f_{yy}(x,y)
            trial_dx2 = dx2_trial(x,y,n_out,n_x,n_xx)
            trial_dy2 = dy2_trial(x,y,n_out,n_y,n_yy)
            # Get value of LHS of differential equation D(x,y) = 0
            D = diffEq(x,y, trial_dx2, trial_dy2)

            cost = lossFn(D, torch.zeros_like(D)) # calculate and store cost

            cost.backward() # perform backpropagation
            optimiser.step() # perform parameter optimisation
            optimiser.zero_grad() # reset gradients to zero

        cost_list.append(cost.item())
    network.train(False)
    return cost_list
```

Figure C.17: Training a network to solve example 5 from the Lagaris paper.

```
# learningRates = [1e-10, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
learningRates = [1e-3, 2e-3, 3e-3, 4e-3, 5e-3, 6e-3, 7e-3, 8e-3, 9e-3, 1e-2]

try: # load saved network if possible
    checkpoint = torch.load('problem5InitialNetwork.pth')
    network    = checkpoint['network']
except: # create new network
    network    = PDESolver(numHiddenNodes=16)
    checkpoint = {'network': network}
    torch.save(checkpoint, 'problem5InitialNetwork.pth')

xRange = [0,1]
yRange = [0,1]
numSamples = 10
numEpochs = 1000
costListDict = {}

lossFn      = torch.nn.MSELoss()
trainSet    = DataSet(xRange,yRange,numSamples)
trainLoader = torch.utils.data.DataLoader(dataset=trainSet, batch_size=int(numSamples**2), shuffle=True)

for lr in learningRates:
    checkpoint = torch.load('problem5InitialNetwork.pth')
    network    = checkpoint['network'] # reset network to initial values
    optimiser  = torch.optim.Adam(network.parameters(), lr = lr)
    costList = []
    epoch = 0
    totalEpochs = 10000
    while epoch < totalEpochs:
        costList.extend(train(network, trainLoader, lossFn, optimiser, numEpochs))
        epoch += numEpochs
```

Figure C.18: Code to carry out 10,000 epochs of training to solve example 5 from the Lagaris paper. By creating a model and saving its initial state, we guarantee that the different networks we wish to compare have the same initial parameter values. Each network is trained with a different learning rate.

```
def trial_term(x,y):
    """
    First term F(x,y) in trial solution that helps to satisfy BCs f(0,y) = f(1,y) = f(x,0) = 0, f_{y}(x,1) = 2*sin(pi*x)
    F(x,y) = y * 2 * sin(pi*x)

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    Returns:
    y * 2 * sin(pi*x) (PyTorch tensor shape (batchSize,1)) -- value of first term F(x,y) in trial solution
    """
    return y**2 * torch.sin(np.pi * x)

def trial(x,y,n_outXY,n_outX1,n_outX1_y):
    """
    Trial solution to Lagaris problem 8: y * 2 * sin(pi*x) + x*(1-x)*y*[N(x,y) - N(x,1) - N_{y}(x,1)]

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    n_outXY (PyTorch tensor shape (batchSize,1)) -- N(x,y), neural network outputs at (x,y)
    n_outX1 (PyTorch tensor shape (batchSize,1)) -- N(x,1), neural network outputs at (x,1)
    n_outX1_y (PyTorch tensor shape (batchSize,1)) -- N_{y}(x,1) partial derivative w.r.t. y of neural network at (x,1)
    Returns:
    f(x,y) (PyTorch tensor shape (batchSize,1)) -- trial solution at (x,y)
    """
    return trial_term(x,y) + x*(1-x)*y*(n_outXY - n_outX1 - n_outX1_y)

def dx_trial(x,y,n_outXY, n_outX1, n_outX1_y, n_outXY_x, n_outX1_x, n_outX1_xy):
    """
    First derivative w.r.t. x of trial solution at (x,y):
    f_{x}(x,y) = y**2*pi*cos(pi*x) + y * [(1-2*x)(N - N(x,1) - N_{y}(x,1)) + x(1-x)(N_{x} - N_{x}(x,1) - N_{xy}(x,1)]

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    n_outXY (PyTorch tensor shape (batchSize,1)) -- N(x,y), neural network outputs at (x,y)
    n_outX1 (PyTorch tensor shape (batchSize,1)) -- N(x,1), neural network outputs at (x,1)
    n_outX1_y (PyTorch tensor shape (batchSize,1)) -- N_{y}(x,1)
    n_outXY_x (PyTorch tensor shape (batchSize,1)) -- N_{x}(x,y)
    n_outX1_x (PyTorch tensor shape (batchSize,1)) --  N_{x}(x,1)
    n_outX1_xy (PyTorch tensor shape (batchSize,1)) --  N_{xy}(x,1)
    Returns:
    f_{x}(x,y) (PyTorch tensor shape (batchSize,1)) -- first derivative w.r.t. x of trial solution at (x,y)"""
    return ( y**2 *np.pi * torch.cos(np.pi*x) + y * ((1-2*x) * (n_outXY - n_outX1 - n_outX1_y)
                + x*(1-x)*(n_outXY_x - n_outX1_x - n_outX1_xy)))
```

Figure C.19: 1/3: Defining functions needed to solve example 8 from the Lagaris paper (i.e. the trial solution, the trial solution's derivatives, the differential equation applied to the trial solution, and the exact solution for comparison).

```
def dx2_trial(x,y,n_outXY, n_outX1, n_outX1_y, n_outXY_x, n_outX1_x, n_outX1_xy, n_outXY_xx, n_outX1_xx, n_outX1_xxy):
    """
    Second derivative w.r.t. x of trial solution at (x,y):
    f_{xx}(x,y) = -y**2*pi^2*sin(pi*x) + y [ (-2)*(N - N(x,1) - N_{y}(x,1) + 2(1-2x)((N_{x} - N_{x}(x,1) - N_{xy}(x,1))
                    + x(1-x)(N_{xx} - N_{xx}(x,1) - N_{xxy}(x,1))]

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    n_outXY (PyTorch tensor shape (batchSize,1)) -- N(x,y), neural network outputs at (x,y)
    n_outX1 (PyTorch tensor shape (batchSize,1)) -- N(x,1), neural network outputs at (x,1)
    n_outX1_y (PyTorch tensor shape (batchSize,1)) -- N_{y}(x,1)
    n_outXY_x (PyTorch tensor shape (batchSize,1)) -- N_{x}(x,y)
    n_outX1_x (PyTorch tensor shape (batchSize,1)) --  N_{x}(x,1)
    n_outX1_xy (PyTorch tensor shape (batchSize,1)) -- N_{xy}(x,1)
    n_outXY_xx (PyTorch tensor shape (batchSize,1)) -- N_{xx}(x,y)
    n_outX1_xx (PyTorch tensor shape (batchSize,1)) -- N_{xx}(x,1)
    n_outX1_xxy (PyTorch tensor shape (batchSize,1)) -- N_{xxy}(x,1)

    Returns:
    f_{xx}(x,y) (PyTorch tensor shape (batchSize,1)) -- second derivative w.r.t. x of trial solution at (x,y)
    """
    return ( -y**2 *(np.pi)**2 * torch.sin(np.pi*x) + y * ( (-2) * (n_outXY - n_outX1 - n_outX1_y)
                + 2*(1-2*x) * (n_outXY_x - n_outX1_x - n_outX1_xy) + x*(1-x)*(n_outXY_xx - n_outX1_xx - n_outX1_xxy)))

def dy_trial(x,y, n_outXY, n_outX1, n_outX1_y, n_outXY_y):
    """
    First derivative w.r.t. y of trial solution at (x,y):
    f_{y}(x,y) = 2ysin(pi*x) + x(1-x)[(N(x,y) - N(x,1) - N_{y}(x,1)) + y * N_{y}(x,y)]

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    n_outXY (PyTorch tensor shape (batchSize,1)) -- N(x,y), neural network outputs at (x,y)
    n_outX1 (PyTorch tensor shape (batchSize,1)) -- N(x,1), neural network outputs at (x,1)
    n_outX1_y (PyTorch tensor shape (batchSize,1)) -- N_{y}(x,1)
    n_outXY_y (PyTorch tensor shape (batchSize,1)) -- N_{y}(x,y)

    Returns:
    f_{y}(x,y) (PyTorch tensor shape (batchSize,1)) -- first derivative w.r.t. y of trial solution at (x,y)
    """
    return (2*y*torch.sin(np.pi *x) + x*(1-x) * ((n_outXY - n_outX1 - n_outX1_y) + (y* n_outXY_y)))
```

Figure C.20: 2/3: Defining functions needed to solve example 8 from the Lagaris paper (i.e. the trial solution, the trial solution's derivatives, the differential equation applied to the trial solution, and the exact solution for comparison).

```
def dy2_trial(x,y,n_outXY_y,n_outXY_yy):
    """
    Second derivative w.r.t. y of trial solution at (x,y):
    f_{yy}(x,y) = 2sin(pi*x) + x(1-x)[2N_{y}(x,y) + y * N_{yy}(x,y)]

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    n_outXY_y (PyTorch tensor shape (batchSize,1)) -- N_{y}(x,y)
    n_outXY_yy (PyTorch tensor shape (batchSize,1)) -- N_{yy}(x,y)

    Returns:
    f_{yy}(x,y) (PyTorch tensor shape (batchSize,1)) -- second derivative w.r.t. y of trial solution at (x,y)
    """
    return (2*torch.sin(np.pi *x) + x * (1-x) * (2 * n_outXY_y + y * n_outXY_yy))

def diffEq(x,y,trialFunc, trial_dy, trial_dx2, trial_dy2):
    """
    Returns D(x,y) from differential equation D(x,y) = 0, Lagaris problem 8

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs
    trialFunc (PyTorch tensor shape (batchSize,1)) -- f(x,y) trial solution at (x,y)
    trial_dy (PyTorch tensor shape (batchSize,1)) -- f_{y}(x,y)
    trial_dx2 (PyTorch tensor shape (batchSize,1)) -- f_{x}(x,y)
    trial_dy2 (PyTorch tensor shape (batchSize,1)) -- f_{yy}(x,y)

    Returns:
    D(x,y) (PyTorch tensor shape (batchSize,1)) -- D(x,y) from DE D(x,y) = 0, Lagaris problem 8
    """
    RHS = torch.sin(np.pi*x)*(2 - np.pi**2*y**2 + 2*y**3*torch.sin(np.pi*x))
    return trial_dx2 + trial_dy2 + trialFunc * trial_dy - RHS

def solution(x, y):
    """
    Analytic solution to Lagaris problem 8, f(x,y) = (y**2) * torch.sin(np.pi * x)

    Arguments:
    x (PyTorch tensor shape (batchSize,1)) -- x-values of neural network inputs
    y (PyTorch tensor shape (batchSize,1)) -- y-values of neural network inputs

    Returns:
    (y**2) * torch.sin(np.pi * x) (PyTorch tensor shape (batchSize,1)) -- true solution at (x,y)
    """
    return (y**2) * torch.sin(np.pi * x)
```

Figure C.21: 3/3: Defining functions needed to solve example 8 from the Lagaris paper (i.e. the trial solution, the trial solution's derivatives, the differential equation applied to the trial solution, and the exact solution for comparison).

```python
def train(network, loader, lossFn, optimiser, numEpochs):
    """
    A function to train a neural network to solve a 2-dimensional PDE with mixed boundary conditions

    Arguments:
    network (Module) -- the neural network
    loader (DataLoader) -- generates batches from the training dataset
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    numEpochs (int) -- number of training epochs

    Returns:
    cost_list (list of length 'numEpochs') -- cost values of all epochs
    """
    cost_list=[]
    network.train(True)
    for _ in range(numEpochs):
        if samplingMethod == "Uniform": # sample new points uniformly every epoch
            trainData = UniformDataSet(xRange,yRange,numSamples)
            loader = torch.utils.data.DataLoader(dataset=trainData, batch_size=int(numSamples**2), shuffle=True)
        if samplingMethod == "Normal": # sample new points from Normal distribution every epoch
            trainData = NormalDataSet(xRange,yRange,numSamples)
            loader = torch.utils.data.DataLoader(dataset=trainData, batch_size=int(numSamples**2), shuffle=True)
        for batch in loader:
            x, y = torch.split(batch,1, dim=1) # separate batch into x- and y-values
            y_ones = torch.ones_like(y)     # create tensor of ones
            x1 = torch.cat((x,y_ones),1)    # Coordinates (x,1) for all x in batch

            n_outXY = network(batch)    # Neural network output at (x,y)
            n_outX1 = network(x1)       # Neural network output at (x,1)

            # Get all required derivatives of n(x,y)
            grad_n_outXY = grad(n_outXY, batch, torch.ones_like(n_outXY), retain_graph=True, create_graph=True)[0]
            n_outXY_x, n_outXY_y = torch.split(grad_n_outXY,1,dim=1) # n_x , n_y

            grad_grad_n_outXY = grad(grad_n_outXY, batch, torch.ones_like(grad_n_outXY), retain_graph=True, create_graph=True)[0]
            n_outXY_xx, n_outXY_yy = torch.split(grad_grad_n_outXY , 1, dim = 1 ) # n_xx , n_yy

            # Get all required derivatives of n(x,1):
            grad_n_outX1 = grad(n_outX1, x1, torch.ones_like(n_outX1), retain_graph=True, create_graph=True)[0]
            n_outX1_x, n_outX1_y = torch.split(grad_n_outX1 , 1, dim = 1 )     # n_x |(y=1) , n_y |(y=1)

            dn2dx_x1 = grad(n_outX1_x, x1, torch.ones_like(n_outX1_x), retain_graph = True, create_graph = True)[0]
            n_outX1_xx , n_outX1_xy = torch.split(dn2dx_x1, 1, dim = 1)     # n_xx |(y=1), n_xy |(y=1)

            grad_n_outX1_xy = grad(n_outX1_xy, x1, torch.ones_like(n_outX1_xy), retain_graph = True, create_graph = True)[0]
            n_outX1_xxy ,  _= torch.split(grad_n_outX1_xy, 1, dim=1)     # n_xxy |(y=1)

            # Get trial solution
            trialFunc = trial(x, y, n_outXY, n_outX1, n_outX1_y)
            # Get first partial derivative (w.r.t y) of trial solution
            trial_dy  = dy_trial(x, y, n_outXY, n_outX1, n_outX1_y, n_outXY_y)
            # Get second partial derivatives of trial solution
            trial_dx2 = dx2_trial(x,y,n_outXY, n_outX1, n_outX1_y, n_outXY_x, n_outX1_x,
                                  n_outX1_xy, n_outXY_xx, n_outX1_xx, n_outX1_xxy)
            trial_dy2 = dy2_trial(x,y,n_outXY_y,n_outXY_yy)

            # Calculate LHS of differential equation D(x,y) = 0
            D = diffEq(x, y, trialFunc, trial_dy, trial_dx2, trial_dy2)

            cost = lossFn(D, torch.zeros_like(D))   # calculate cost
            cost.backward()     # perform backpropagation
            optimiser.step()    # perform parameter optimisation
            optimiser.zero_grad()   # reset gradients to zero

        cost_list.append(cost.item()) # store final cost of every epoch
    network.train(False)
    return cost_list
```

Figure C.22: Training a network to solve example 8 from the Lagaris paper.

```python
class UniformDataSet(torch.utils.data.Dataset):
    """
    An object which generates uniformly sampled (x,y) values for the input node
    """
    def __init__(self, xRange, yRange, numSamples):
        """
        Arguments:
        xRange (list of length 2) -- lower and upper limits for input values x
        yRange (list of length 2) -- lower and upper limits for input values y
        numSamples (int) -- number of training data samples along each axis

        Returns:
        DataSet object with one attributes:
            dataIn (PyTorch tensor of shape (numSamples^2,2)) -- 'numSamples'^2
                uniformly sampled grid points from (xRange[0], yRange[0]) to (xRange[1], yRange[1])
        """
        # uniformly sample numSamples^2 x-values in xRange
        X  = torch.distributions.Uniform(xRange[0],xRange[1]).sample((int(numSamples**2),1))
        X.requires_grad = True
        # uniformly sample numSamples^2 y-values in yRange
        Y  = torch.distributions.Uniform(yRange[0],yRange[1]).sample((int(numSamples**2),1))
        Y.requires_grad = True

        # format these numSamples^2 (x,y) coordinates in a tensor of shape (numSamples^2,2)
        self.data_in = torch.cat((X,Y),1)

    def __len__(self):
        """
        Arguments:
        None
        Returns:
        len(self.dataIn) (int) -- number of training data points
        """
        return self.data_in.shape[0]

    def __getitem__(self, i):
        """
        Used by DataLoader object to retrieve training data points
        Arguments:
        idx (int) -- index of data point required
        Returns:
        [x,y] (tensor shape (1,2)) -- data point at index 'idx'
        """
        return self.data_in[i]
```

Figure C.23: Creating uniformly-sampled pairs of points across the domain $[0,1] \times [0,1]$.

```
class NormalDataSet(torch.utils.data.Dataset):
    """
    An object which generates Normally sampled (x,y) values for the input node
    """
    def __init__(self, xRange, yRange, numSamples):
        """
        Arguments:
        xRange (list of length 2) -- lower and upper limits for input values x
        yRange (list of length 2) -- lower and upper limits for input values y
        numSamples (int) -- number of training data samples along each axis

        Returns:
        DataSet object with one attributes:
            dataIn (PyTorch tensor of shape (numSamples^2,2)) -- 'numSamples'^2
                Normally-sampled grid points from (xRange[0], yRange[0]) to (xRange[1], yRange[1])
        """
        # sample numSamples^2 x-values in xRange from N(mu, sigma), where mu is the midpoint of xRange
        # and sigma is 1/9  of the width of xRange: this means all values are virtually guaranteed to lie in xRange
        # for different value of numSamples, adjust denominator of sigma
        X  = torch.normal((xRange[1]- xRange[0])/2, (xRange[1]- xRange[0])/9, (int(numSamples**2),1))
        X.requires_grad = True
        # same process for y-values
        Y  = torch.normal((yRange[1]- yRange[0])/2, (yRange[1]- yRange[0])/9, (int(numSamples**2),1))
        Y.requires_grad = True
        # format these numSamples^2 (x,y) coordinates in a tensor of shape (numSamples^2,2)
        self.data_in = torch.cat((X,Y),1)

    def __len__(self):
        """
        Arguments:
        None
        Returns:
        len(self.dataIn) (int) -- number of training data points
        """
        return self.data_in.shape[0]

    def __getitem__(self, i):
        """
        Used by DataLoader object to retrieve training data points

        Arguments:
        idx (int) -- index of data point required
        Returns:
        [x,y] (tensor shape (1,2)) -- data point at index 'idx' """
        return self.data_in[i]
```

Figure C.24: Creating a Normally-sampled pairs of points across the domain $[0, 1] \times [0, 1]$.

```
numSamples  = 10
xRange      = yRange    = [0,1]
numEpochs   = 1000
totalEpochs = 10000
networkDict = costListDict = {}

datasetDict = {"Normal" : NormalDataSet(xRange,yRange,numSamples),
               "Uniform" : UniformDataSet(xRange,yRange,numSamples),
               "Lattice" : LinearDataSet(xRange,yRange,numSamples)}

for samplingMethod in datasetDict:
    networkDict = {}
    trainData = datasetDict[samplingMethod]

    x, y = torch.split(trainData.data_in, 1, 1)
    plt.plot(x.detach().numpy(),y.detach().numpy(), 'b.')
    plt.xlabel("x",fontsize = 16)
    plt.ylabel("y", fontsize = 16)
    plt.title("Data Points, Sampling Method: " + samplingMethod, fontsize = 16)
    plt.show()

    try: # load saved network if possible
        checkpoint = torch.load('problem8InitialNetwork.pth')
        network    = checkpoint['network']
    except: # create new network
        network    = PDESolver(numHiddenNodes=16)
        checkpoint = {'network': network}
        torch.save(checkpoint, 'problem8InitialNetwork.pth')

    lossFn      = torch.nn.MSELoss()
    optimiser   = torch.optim.Adam(network.parameters(), lr = 1e-3)
    trainLoader = torch.utils.data.DataLoader(dataset=trainData, batch_size=int(numSamples**2), shuffle=True)
    epoch = 0
    costList = []

    start = time.time()
    while epoch < totalEpochs:
        costList.extend(train(network, trainLoader, lossFn, optimiser, numEpochs))
        epoch += numEpochs
    end = time.time()
    print("total training time = ", end-start, " seconds")

    costListDict[samplingMethod] = costList
    networkDict["costList"] = costList
    networkDict["network"] = network
    torch.save(networkDict, 'problem8' + samplingMethod + '.pth')
```

Figure C.25: Code to carry out 10,000 epochs of training to solve example 8 from the Lagaris paper. By creating a model and saving its initial state, we guarantee that the different networks we wish to compare have the same initial parameter values. Each network is trained using a different sampling method to generate training data points.

```
class DataSet(torch.utils.data.Dataset):
    """Samples 'batchSize' random samples of initial values (x_0, y_0, u_0, v_0) and times t from
        X_0 x tRange where X_0 = xRange x yRange x uRange x vRange, the ranges of initial conditions """
    def __init__(self, xRange, yRange, uRange, vRange, tRange, batchSize):
        """
        Arguments:
        xRange (list of length 2) -- lower and upper limits for initial conditions x_0
        yRange (list of length 2) -- lower and upper limits for initial conditions y_0
        uRange (list of length 2) -- lower and upper limits for initial conditions u_0
        vRange (list of length 2) -- lower and upper limits for initial conditions v_0
        tRange (list of length 2) -- lower and upper limits for time values t
        batchSize (int) -- number of training data samples

        Returns:
        DataSet object with one attribute:
        dataIn (tuple of 5 PyTorch tensor of shape (batchSize,1)) -- 'batchSize' neural network inputs
            of the form (x_0, y_0, u_0, v_0, t)
        """
        global device
        # uniformly sample values of x_0, y_0, u_0, v_0 and t from their respective ranges
        # move all tensors to GPU
        X = torch.distributions.uniform.Uniform(xRange[0],xRange[1]).sample([batchSize,1]).to(device)
        Y = torch.distributions.uniform.Uniform(yRange[0],yRange[1]).sample([batchSize,1]).to(device)
        U = torch.distributions.uniform.Uniform(uRange[0],uRange[1]).sample([batchSize,1]).to(device)
        V = torch.distributions.uniform.Uniform(vRange[0],vRange[1]).sample([batchSize,1]).to(device)
        T = torch.distributions.uniform.Uniform(tRange[0],tRange[1]).sample([batchSize,1]).to(device)
        # set requires grad = True to create a computation graph and allow gradient calculation
        X.requires_grad_()
        Y.requires_grad_()
        U.requires_grad_()
        V.requires_grad_()
        T.requires_grad_()

        # return inputs as a tuple of 5 tensors (these must be concatenated before passed to network)
        self.data_in = (X,Y,U,V,T)

    def __len__(self):
        """
        Arguments:
        None

        Returns:
        len(self.dataIn) (int) -- number of training data points
        """
        return self.data_in.shape[0]

    def __getitem__(self, idx):
        """
        Used by DataLoader object to retrieve training data points
        Arguments:
        idx (int) -- index of data point required

        Returns:
        (x_0, y_0, u_0, v_0, t) (tensor of shape (1,5)) -- data point at index 'idx'
        """
        return self.data_in[idx]
```

Figure C.26: Code to create a uniformly sampled data set of initial conditions and time values for the planar circular restricted three-body problem.

```python
def trialSolution(varInitial, varOut, t):
    """
    Trial solution for a given variable varOut at times t with initial values varInitial

    Arguments:
    varInitial (tensor of shape (batchSize,1)) -- initial values of given variable
    varOut (tensor of shape (batchSize,1)) -- network output for variable at times t
    t (tensor of shape (batchSize,1)) -- times at which variable is evaluated

    Returns:
    trialSoln (tensor of shape (batchSize,1)) -- trial solution for given variable at
        times t with initial values varInitial"""
    trialSoln = varInitial + (1 - torch.exp(-t)) * varOut
    return trialSoln

def dTrialSolution(varOut, dVarOut, t):
    """
    Derivative w.r.t. t of trial solution for a given variable varOut at times t with initial values varInitial

    Arguments:
    varOut (tensor of shape (batchSize,1)) -- network output for variable at times t
    dVarOut (tensor of shape (batchSize,1)) -- derivative of network output for variable w.r.t. t at times t
    t (tensor of shape (batchSize,1)) -- times at which variable is evaluated

    Returns:
    trialSoln (tensor of shape (batchSize,1)) -- trial solution for given variable at
        times t with initial values varInitial"""
    dTrialSoln = ((1 - torch.exp(-t)) * dVarOut) + (torch.exp(-t)) * varOut
    return dTrialSoln

def diffEqX(u0, uOut, xOut, dxOut, t):
    """
    Returns LHS of differential equation for x(t)

    Arguments:
    u0 (tensor of shape (batchSize,1)) -- initial values of u at time t_0
    uOut (tensor of shape (batchSize,1)) -- network output for u at times t
    xOut (tensor of shape (batchSize,1)) -- network output for x at times t
    dxOut (tensor of shape (batchSize,1)) -- derivative of network output for x w.r.t. t at times t
    t (tensor of shape (batchSize,1)) -- times

    Returns:
    dxTrial - u (tensor of shape (batchSize,1)) -- LHS of DE for x(t) at times t
    """
    u = trialSolution(u0, uOut, t)
    dxTrial = dTrialSolution(xOut, dxOut, t)
    return dxTrial - u

def diffEqY(v0, vOut, yOut, dyOut, t):
    """
    Returns LHS of differential equation for y(t)

    Arguments:
    v0 (tensor of shape (batchSize,1)) -- initial values of v at time t_0
    vOut (tensor of shape (batchSize,1)) -- network output for v at times t
    yOut (tensor of shape (batchSize,1)) -- network output for y at times t
    dyOut (tensor of shape (batchSize,1)) -- derivative of network output for y w.r.t. t at times t
    t (tensor of shape (batchSize,1)) -- times

    Returns:
    dyTrial - v (tensor of shape (batchSize,1)) -- LHS of DE for y(t) at times t
    """
    v = trialSolution(v0, vOut, t)
    dyTrial = dTrialSolution(yOut, dyOut, t)
    return dyTrial - v
```

Figure C.27: 1/2: Code to evaluate the trial solutions, their derivatives, and the differential equations for the planar circular restricted three-body problem.

```
def diffEqU(x0, y0, v0, xOut, yOut, vOut, uOut, duOut, t, mu):
    """
    Returns LHS of differential equation for u(t)

    Arguments:
    x0 (tensor of shape (batchSize,1)) -- initial values of x at time t_0
    y0 (tensor of shape (batchSize,1)) -- initial values of y at time t_0
    v0 (tensor of shape (batchSize,1)) -- initial values of v at time t_0
    xOut (tensor of shape (batchSize,1)) -- network output for x at times t
    yOut (tensor of shape (batchSize,1)) -- network output for y at times t
    uOut (tensor of shape (batchSize,1)) -- network output for u at times t
    vOut (tensor of shape (batchSize,1)) -- network output for v at times t
    duOut (tensor of shape (batchSize,1)) -- derivative of network output for u w.r.t. t at times t
    t (tensor of shape (batchSize,1)) -- times
    mu (float) -- non-dimensionalised mass of the second body

    Returns:
    diffEqULHS (tensor of shape (batchSize,1)) -- LHS of DE for u(t) at times t
    """
    x = trialSolution(x0, xOut, t)
    y = trialSolution(y0, yOut, t)
    v = trialSolution(v0, vOut, t)
    duTrial = dTrialSolution(uOut, duOut, t)
    diffEqULHS = duTrial - (x - mu + 2 * v - (((mu * (x - 1)) / ((x - 1) ** 2 + y**2) ** (3 / 2))
        + (((1 - mu) * x) / (x**2 + y**2) ** (3 / 2))))
    return diffEqULHS

def diffEqV(x0, y0, u0, xOut, yOut, uOut, vOut, dvOut, t, mu):
    """
    Returns LHS of differential equation for v(t)

    Arguments:
    x0 (tensor of shape (batchSize,1)) -- initial values of x at time t_0
    y0 (tensor of shape (batchSize,1)) -- initial values of y at time t_0
    u0 (tensor of shape (batchSize,1)) -- initial values of u at time t_0
    xOut (tensor of shape (batchSize,1)) -- network output for x at times t
    yOut (tensor of shape (batchSize,1)) -- network output for y at times t
    uOut (tensor of shape (batchSize,1)) -- network output for u at times t
    vOut (tensor of shape (batchSize,1)) -- network output for v at times t
    dvOut (tensor of shape (batchSize,1)) -- derivative of network output for v w.r.t. t at times t
    t (tensor of shape (batchSize,1)) -- times
    mu (float) -- non-dimensionalised mass of the second body

    Returns:
    diffEqVLHS (tensor of shape (batchSize,1)) -- LHS of DE for v(t) at times t
    """
    x = trialSolution(x0, xOut, t)
    y = trialSolution(y0, yOut, t)
    u = trialSolution(u0, uOut, t)
    dvTrial = dTrialSolution(vOut, dvOut, t)
    diffEqVLHS = dvTrial - (y - 2 * u - (((mu * y) / ((x - 1) ** 2 + y**2) ** (3 / 2))
        + (((1 - mu) * y) / (x**2 + y**2) ** (3 / 2))))
    return diffEqVLHS
```

Figure C.28: 2/2: Code to evaluate the trial solutions, their derivatives, and the differential equations for the planar circular restricted three-body problem.

```python
def train(network, lossFn, optimiser, scheduler, xRange, yRange, uRange, vRange, tRange, batchSize, mu, lmbda):
    """
    A function to train a neural network on a batch of size 'batchSize' to approximate the solution to the
    planar-restricted three-body problem for a bundle of initial conditions

    Arguments:
    network (Module) -- the neural network
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    scheduler (Learning Rate Scheduler) -- reduces learning rate if cost value is plateauing
    xRange (list of length 2) -- lower and upper limits for initial conditions x_0
    yRange (list of length 2) -- lower and upper limits for initial conditions y_0
    uRange (list of length 2) -- lower and upper limits for initial conditions u_0
    vRange (list of length 2) -- lower and upper limits for initial conditions v_0
    tRange (list of length 2) -- lower and upper limits for time values t
    batchSize (int) -- number of training samples to use
    mu (float) -- non-dimensionalised mass of the second body
    lmbda (float) -- factor in the weighting function exp(-lmbda * t) in the cost function

    Returns:
    cost (float) -- network's cost evaluated on single batch of training data"""
    global device
    network.train(True) # set network into training mode
    x, y, u, v, t  = DataSet(xRange,yRange,uRange,vRange,tRange,batchSize).data_in # generate data set
    batch = torch.cat((x,y,u,v,t),1) # input of neural network must be of shape (batchSize, 5)
    out = network.forward(batch) # pass training batch through network
    xOut, yOut, uOut, vOut = torch.split(out, 1, dim = 1) # separate outputs

    # Get derivative of every variable w.r.t. t
    dxOut = grad(xOut,t,torch.ones_like(xOut),retain_graph=True, create_graph=True)[0]
    dyOut = grad(yOut,t,torch.ones_like(yOut),retain_graph=True, create_graph=True)[0]
    duOut = grad(uOut,t,torch.ones_like(uOut),retain_graph=True, create_graph=True)[0]
    dvOut = grad(vOut,t,torch.ones_like(vOut),retain_graph=True, create_graph=True)[0]

    # evaluate each of the 4 differential equations
    dxEq = diffEqX(u, uOut, xOut, dxOut, t)
    dyEq = diffEqY(v, vOut, yOut, dyOut, t)
    duEq = diffEqU(x, y, v, xOut, yOut, vOut, uOut, duOut, t, mu)
    dvEq = diffEqV(x, y, u, xOut, yOut, uOut, vOut, dvOut, t, mu)

    # evaluate cost function with weighting factor exp(-lambda * t)
    dxCost = lossFn( torch.exp(-lmbda*t) * dxEq, torch.zeros_like(dxEq))
    dyCost = lossFn( torch.exp(-lmbda*t) * dyEq, torch.zeros_like(dyEq))
    duCost = lossFn( torch.exp(-lmbda*t) * duEq, torch.zeros_like(duEq))
    dvCost = lossFn( torch.exp(-lmbda*t) * dvEq, torch.zeros_like(dvEq))
    cost = (dxCost + dyCost + duCost + dvCost)

    cost.backward() # perform back propagation
    optimiser.step() # optimise parameters
    # reset gradients to None instead of zero; this saves memory without altering computation
    optimiser.zero_grad(set_to_none =True)
    scheduler.step(cost) # update scheduler, tracks cost and updates learning rate if on plateau

    network.train(False) # set network out of training mode
    return cost.detach().cpu().numpy() # store cost in a numpy array on cpu to allow plotting
```

Figure C.29: Code to train the network on one randomly generated batch of training data for the planar circular restricted three-body problem.

```
class SolutionBundle(torch.nn.Module):
    """
    A deep neural network object, with 5 nodes in the input layer, 1 node in the
    output layer, and 'numHiddenLayers' hidden layers each with 'numHiddenNodes' nodes.
    """
    def __init__(self, numHiddenNodes, numHiddenLayers):
        """
        Arguments:
        numHiddenNodes (int) -- number of nodes in hidden layers
        numHiddenLayers (int) -- number of hidden layers

        Returns:
        SolutionBundle object (neural network) with three attributes:
        fc1 (fully connected layer) -- linear transformation of first layer
        fcs (list of fully connected layers) -- linear transformations of hidden layers
        fcLast (fully connected layer) -- linear transformation of outer layer
        """
        super(SolutionBundle, self).__init__()
        # create first layer, apply Xavier initialisation
        self.fc1 = torch.nn.Linear(5, numHiddenNodes)
        self.fc1.apply(self.initWeightsXavier)
        # create list of hidden layers, apply Xavier initialisation
        self.fcs = torch.nn.ModuleList([torch.nn.Linear(numHiddenNodes, numHiddenNodes)
                    for _ in range(numHiddenLayers)])
        self.fcs.apply(self.initWeightsXavier)
        # create final layer, apply Xavier initialisation
        self.fcLast = torch.nn.Linear(numHiddenNodes, 4)
        self.fcLast.apply(self.initWeightsXavier)

    def forward(self, input):
        """
        Function which performs forward propagation in the neural network.

        Arguments:
        input (PyTorch tensor shape (batchSize, 5)) -- input of neural network
        Returns:
        output (PyTorch tensor shape (batchSize, 4)) -- output of neural network
        """
        hidden = torch.tanh(self.fc1(input))
        # pass through all hidden layers
        for i in range(len(self.fcs)):
            hidden = torch.tanh(self.fcs[i](hidden))
        output = self.fcLast(hidden)
        return output

    def initWeightsXavier(self, layer):
        """
        Function which initialises weights according to Xavier initialisation

        Arguments:
        layer (Linear object) -- weights and biases of a layer
        Returns:
        None
        """
        if type(layer) == torch.nn.Linear:
            torch.nn.init.xavier_uniform_(layer.weight, gain = torch.nn.init.calculate_gain('tanh'))
```

Figure C.30: Code to create a neural network solution bundle for the planar circular restricted three-body problem. Multiple network layers can be created using a ModuleList object. Note the we initialise the weights according to the Xavier initialisation method.

```
462  if torch.cuda.is_available():
463      print("GPU available")
464      device=torch.device("cuda")
465  else:
466      print("no GPU available")
467      device=torch.device("cpu")
468
469  xRange = [1.05,1.052]
470  yRange = [0.099, 0.101]
471  uRange = [-0.5,-0.4]
472  vRange = [-0.3,-0.2]
473  tRange = [-0.01,3]
474  batchSize = 10
475  mu = 0.01
476  lmbda = 2
477  numTimeSteps = 1000
478  numTotalBatches = 3000000
479
480  try: # load model if possible
481      checkpoint  = torch.load('threeBodyOriginalMethod.pth')
482      batchNum    = checkpoint['batchNum']
483      network     = checkpoint['network']
484      optimiser   = checkpoint['optimiser']
485      scheduler   = checkpoint['scheduler']
486      costs       = checkpoint['costs']
487      print("model loaded")
488  except:
489      try: # load initial state of model
490          checkpoint = torch.load('threeBodyInitialNetwork.pth')
491          network = checkpoint['network']
492          print("initial model loaded")
493      except:  # create new model and save its initial state
494          network     = SolutionBundle(numHiddenNodes=128, numHiddenLayers=8)
495          checkpoint  = {'network' : network}
496          torch.save(checkpoint, 'threeBodyInitialNetwork.pth')
497          print("new model created")
498      batchNum = 0
499      optimiser = torch.optim.Adam(network.parameters(), lr = 1e-3)
500      scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimiser, factor = 0.5, patience = 200000,
501                                  threshold = 0.5, min_lr = 1e-6, verbose = True)
502      costs = []
503  network = network.to(device) # move network to GPU if available
504  lossFn  = torch.nn.MSELoss()
505
506  while batchNum <= numTotalBatches:
507      newCost = train(network, lossFn, optimiser, scheduler, xRange,
508                      yRange, uRange, vRange, tRange, batchSize, mu, lmbda)
509      costs.append(newCost)
510      if batchNum % 50000 == 0 : # save network every 50000 batches
511          plotNetwork(network, mu, batchNum,
512              xRange, yRange,uRange,vRange,tRange, numTimeSteps)
513          checkpoint = {'batchNum': batchNum, 'network': network, 'optimiser': optimiser,
514                        'scheduler': scheduler, 'costs': costs}
515          torch.save(checkpoint, 'threeBodyOriginalMethod.pth')
516          print("model saved")
517      batchNum += 1
```

Figure C.31: Code to run 3,000,000 batches of training on GPU for our neural network solution bundle to the planar circular restricted three-body problem, with a learning rate scheduler.

```
351  def dxdt(x,y,u,v,mu):
352      """
353      Returns RHS of differential equation for x'(t) at time t
354      Arguments:
355      x (float) -- value of x at time t
356      y (float) -- value of y at time t
357      u (float) -- value of u at time t
358      v (float) -- value of v at time t
359      Returns:
360      x'(t) (float) -- value of x'(t) at time t
361      """
362      return u
363
364  def dydt(x,y,u,v,mu):
365      """
366      Returns RHS of differential equation for y'(t)
367      Arguments:
368      x (float) -- value of x at time t
369      y (float) -- value of y at time t
370      u (float) -- value of u at time t
371      v (float) -- value of v at time t
372      Returns:
373      y'(t) (float) -- value of y'(t) at time t
374      """
375      return v
376
377  def dudt(x,y,u,v,mu):
378      """
379      Returns RHS of differential equation for u'(t)
380      Arguments:
381      x (float) -- value of x at time t
382      y (float) -- value of y at time t
383      u (float) -- value of u at time t
384      v (float) -- value of v at time t
385      Returns:
386      u'(t) (float) -- value of u'(t) at time t
387      """
388      return (x - mu + 2*v - (((mu*(x-1)) / ((x-1)**2 + y**2)**(3/2))
389                  + ((1-mu)*x / (x**2 + y**2)**(3/2))))
390
391  def dvdt(x,y,u,v,mu):
392      """
393      Returns RHS of differential equation for v'(t)
394      Arguments:
395      x (float) -- value of x at time t
396      y (float) -- value of y at time t
397      u (float) -- value of u at time t
398      v (float) -- value of v at time t
399      Returns:
400      v'(t) (float) -- value of v'(t) at time t
401      """
402      return (y - 2*u - (((mu * y) / ((x-1)**2 + y**2)**(3/2))
403                  +((1-mu)*y / (x**2 + y**2)**(3/2) )) )
```

Figure C.32: 1/2: Code to evaluate the fourth-order Runge-Kutta solution to the planar circular restricted three-body problem.

```
405  def rungeKutta(x0, y0, u0, v0, t0, mu, tFinal, numTimeSteps):
406      """
407      Implements 4th-Order Runge-Kutta Method to evaluate system of ODEs from time t0 to time tFinal
408      Arguments:
409      x0 (float) -- initial value of x at time t0
410      y0 (float) -- initial value of y at time t0
411      u0 (float) -- initial value of u at time t0
412      v0 (float) -- initial value of v at time t0
413      t0 (float) -- initial time value
414      mu (float) -- non-dimensionalised mass of second body
415      tFinal (float) -- final time value
416      numTimeSteps (int) -- number of time step evaluations between t0 and tFinal
417      Returns:
418      xList (list of length numTimeSteps) -- x-values at each time step value
419      yList (list of length numTimeSteps) -- y-values at each time step value
420      """
421      # Find size of time step by dividing interval width by numTimeSteps
422      timeStepSize = int((tFinal - t0)/numTimeSteps)
423
424      x, y, u, v, t = x0, y0, u0, v0, t0
425      xList, yList = [x0], [y0]
426      for _ in range(numTimeSteps):  # Iterate for number of time steps
427          # Apply Runge Kutta Formulas to find next values of x, y, u, v
428          a1 = timeStepSize * dxdt(x,y,u,v,mu)
429          b1 = timeStepSize * dydt(x,y,u,v,mu)
430          c1 = timeStepSize * dudt(x,y,u,v,mu)
431          d1 = timeStepSize * dvdt(x,y,u,v,mu)
432
433          a2 = timeStepSize * dxdt(x + 0.5*a1, y + 0.5*b1, u + 0.5*c1, v + 0.5*d1, mu)
434          b2 = timeStepSize * dydt(x + 0.5*a1, y + 0.5*b1, u + 0.5*c1, v + 0.5*d1, mu)
435          c2 = timeStepSize * dudt(x + 0.5*a1, y + 0.5*b1, u + 0.5*c1, v + 0.5*d1, mu)
436          d2 = timeStepSize * dvdt(x + 0.5*a1, y + 0.5*b1, u + 0.5*c1, v + 0.5*d1, mu)
437
438          a3 = timeStepSize * dxdt(x + 0.5*a2, y + 0.5*b2, u + 0.5*c2, v + 0.5*d2, mu)
439          b3 = timeStepSize * dydt(x + 0.5*a2, y + 0.5*b2, u + 0.5*c2, v + 0.5*d2, mu)
440          c3 = timeStepSize * dudt(x + 0.5*a2, y + 0.5*b2, u + 0.5*c2, v + 0.5*d2, mu)
441          d3 = timeStepSize * dvdt(x + 0.5*a2, y + 0.5*b2, u + 0.5*c2, v + 0.5*d2, mu)
442
443          a4 = timeStepSize * dxdt(x + a3, y + b3, u + c3, v + d3, mu)
444          b4 = timeStepSize * dydt(x + a3, y + b3, u + c3, v + d3, mu)
445          c4 = timeStepSize * dudt(x + a3, y + b3, u + c3, v + d3, mu)
446          d4 = timeStepSize * dvdt(x + a3, y + b3, u + c3, v + d3, mu)
447
448          # Update next value of x, y, u, v
449          x += (1.0 / 6.0)*(a1 + 2 * a2 + 2 * a3 + a4)
450          y += (1.0 / 6.0)*(b1 + 2 * b2 + 2 * b3 + b4)
451          u += (1.0 / 6.0)*(c1 + 2 * c2 + 2 * c3 + c4)
452          v += (1.0 / 6.0)*(d1 + 2 * d2 + 2 * d3 + d4)
453
454          # Store x- and y-values
455          xList.append(x)
456          yList.append(y)
457          # Update next value of t
458          t += timeStepSize
459      return xList, yList
```

Figure C.33: 2/2: Code to evaluate the fourth-order Runge-Kutta solution to the planar circular restricted three-body problem.

```
519  while batchNum <= numTotalBatches:
520      # train on different curriculum depending on current batch number
521      if batchNum < int(numTotalBatches/4):
522          tRange = [-0.01,1]
523      elif batchNum < int(numTotalBatches/2):
524          tRange = [1,2]
525      elif batchNum < int(3*numTotalBatches/4):
526          tRange = [2,3]
527      else:
528          tRange = [-0.01,3]
529      newCost = train(network, lossFn, optimiser, scheduler, xRange,
530                      yRange, uRange, vRange, tRange, batchSize, mu, lmbda)
531      costs.append(newCost)
532      if batchNum % 50000 == 0 : # save network every 50000 batches
533          plotNetwork(network, mu, batchNum,
534              xRange, yRange,uRange,vRange,tRange, numTimeSteps)
535          checkpoint = {'batchNum': batchNum, 'network': network, 'optimiser': optimiser,
536                        'scheduler': scheduler, 'costs': costs}
537          torch.save(checkpoint, 'threeBodyOriginalMethod.pth')
538          print("model saved")
539      batchNum += 1
```

Figure C.34: Code to train a neural network on separate curricula for the planar circular restricted three-body problem.

```
541  while batchNum <= numTotalBatches:
542      # widen time interval based n current batch number
543      tFinal = min(3, np.exp( (3 * np.log(6) * batchNum) / (2.5 * numTotalBatches)) / 2)
544      tRange = [-0.01, tFinal]
545      newCost = train(network, lossFn, optimiser, scheduler, xRange,
546                      yRange, uRange, vRange, tRange, batchSize, mu, lmbda)
547      costs.append(newCost)
548      if batchNum % 50000 == 0 : # save network every 50000 batches
549          plotNetwork(network, mu, batchNum,
550              xRange, yRange,uRange,vRange,tRange, numTimeSteps)
551          checkpoint = {'batchNum': batchNum, 'network': network, 'optimiser': optimiser,
552                        'scheduler': scheduler, 'costs': costs}
553          torch.save(checkpoint, 'threeBodyOriginalMethod.pth')
554          print("model saved")
555      batchNum += 1
```

Figure C.35: Code to train a neural network on the exponential curriculum for the planar circular restricted three-body problem. Training on the linear and logarithmic curricula can be implemented similarly, changing only line 544.

```python
class DataSet(torch.utils.data.Dataset):
    """Samples 'numSamples' random samples of (x, t, u(x,t)) training data from data set of 25,600"""
    def __init__(self, XT, u_exact, numSamples):
        """
        Arguments:
        XT (array of shape (25600, 2)) -- (x,t) input samples
        u_exact (array of shape (25600, 2)) -- exact values u(x,t) for training
        numSamples (int) -- number of training data samples required

        Returns:
        DataSet object with one attribute:
            dataIn (tuple of PyTorch tensors of shape (numSamples,2) and (numSamples,1)) -- 'numSamples'
                randomly sampled (x,t) points with their corresponding function values u(x,t)
        """
        # generate numSamples random indices to get training samples
        idx = np.random.choice(XT.shape[0], numSamples, replace=False)

        # store (x,t) values and u(x,t) values in two separate tensors, and convert them to 32-bit
        XT_train = torch.tensor(XT[idx,:], requires_grad=True).float()
        u_train = torch.tensor(u_exact[idx,:], requires_grad=True).float()

        # input of forward function must have shape (batch_size, 2)
        # u-values for training must have shape (batch_size, 1)
        # we load this data as a tuple of tensors
        self.data_in = (XT_train, u_train)

    def __len__(self):
        """
        Arguments:
        None
        Returns:
        len(self.dataIn) (int) -- number of training data points
        """
        return self.data_in[0].shape[0]

    def __getitem__(self, idx):
        """
        Used by DataLoader object to retrieve training data points

        Arguments:
        idx (int) -- index of data point required

        Returns:
        ((x,t) , u(x,t)) (tuple of tensors shape (1,2) and (1,1)) -- data point (x,t) and u(x,t) at index 'idx'
        """
        return (self.data_in[0][idx,:], self.data_in[1][idx])
```

Figure C.36: Code to create a random sample of training data from the total data set of 25,600 values $(x, t, u(x, t))$ where $u(x, t)$ is the solution to Burger's equation.

```
class BurgersEquationSolver(torch.nn.Module):
    """
    A deep neural network object, with 2 nodes in the input layer, 1 node in the
    output layer, and 'numHiddenLayers' hidden layers each with 'numHiddenNodes' nodes.
    """
    def __init__(self, numHiddenNodes, numHiddenLayers):
        """
        Arguments:
        numHiddenNodes (int) -- number of nodes in hidden layers
        numHiddenLayers (int) -- number of hidden layers
        Returns:
        BurgersEquationSolver object (neural network) with three attributes:
        fc1 (fully connected layer) -- linear transformation of first layer
        fcs (list of fully connected layers) -- linear transformations of hidden layers
        fcLast (fully connected layer) -- linear transformation of outer layer
        """
        super(BurgersEquationSolver, self).__init__()
        # create first layer with 2 inputs (x and t), apply Xavier initialisation
        self.fc1 = torch.nn.Linear(2, numHiddenNodes)
        self.fc1.apply(self.initWeightsXavier)
        # create list of hidden layers, apply Xavier initialisation
        self.fcs = torch.nn.ModuleList([torch.nn.Linear(numHiddenNodes, numHiddenNodes)
                    for _ in range(numHiddenLayers)])
        self.fcs.apply(self.initWeightsXavier)
        # create final layer with one output (u(x,t)), apply Xavier initialisation
        self.fcLast = torch.nn.Linear(numHiddenNodes, 1)
        self.fcLast.apply(self.initWeightsXavier)

        self.lmbda = torch.nn.Parameter(torch.rand(1)) # initialise parameter lambda
        self.nu = torch.nn.Parameter(torch.rand(1)) # initialise parameter nu

    def forward(self, input):
        """
        Function which performs forward propagation in the neural network.
        Arguments:
        input (PyTorch tensor shape (batchSize, 2)) -- input of neural network
        Returns:
        output (PyTorch tensor shape (batchSize, 1)) -- output of neural network
        """
        hidden = torch.tanh(self.fc1(input))
        # pass through all hidden layers
        for i in range(len(self.fcs)):
            hidden = torch.tanh(self.fcs[i](hidden))
        output = self.fcLast(hidden)
        return output

    def initWeightsXavier(self, layer):
        """
        Function which initialises weights according to Xavier initialisation
        Arguments:
        layer (Linear object) -- weights and biases of a layer
        Returns:
        None
        """
        if type(layer) == torch.nn.Linear:
            torch.nn.init.xavier_uniform_(layer.weight, gain = torch.nn.init.calculate_gain('tanh'))
```

Figure C.37: Code to create a neural network to solve Burger's equation and approximate its unknown parameters λ and ν.

```python
def train(network, lossFn, optimiser, scheduler, loader, numEpochs):
    """
    A function to train a neural network to approximate the solution u(x,t) to Burger's equation,
    while simultaneously estimating the parameters lambda and nu from the equation

    Arguments:
    network (Module) -- the neural network
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    scheduler (Learning Rate Scheduler) -- reduces learning rate if cost value is plateauing
    loader (DataLoader) -- generates batches from the training dataset
    numEpochs (int) -- number of training epochs

    Returns:
    costList (list of length 'numEpochs') -- cost values of all epochs
    lambdaList (list of length 'numEpochs') -- lambda values of all epochs
    nuList (list of length 'numEpochs') -- nu values of all epochs
    """
    costList = []
    lmbdaList = []
    nuList = []
    network.train(True) # set network into training mode
    for _ in range(numEpochs):
        for batch in loader:
            input, batch_u_exact = batch # separate inputs (x,t) and exact values u(x,t)
            u_out = network.forward(input) # pass inputs (x,t) through network
            # calculate first- and second-order partial derivatives of network output
            du = grad(u_out, input, torch.ones_like(u_out), retain_graph=True, create_graph=True)[0]
            d2u = grad(du, input, torch.ones_like(du), retain_graph=True, create_graph=True)[0]

            # separate partial derivatives to attain u_t, u_x, u_xx
            # use torch.split to preserve grad history
            u_x, u_t = torch.split(du, 1, dim =1)
            u_xx, u_tt = torch.split(d2u, 1, dim =1)

            # evaluate differential equation
            # since we know nu will always be positive, we train with exp(nu)
            diffEqLHS = u_t + (network.lmbda * u_out * u_x) - (torch.exp(network.nu) * u_xx)

            # calculate J_u and J_D, cost functions for approximation of u(x,t) and DE respectively
            uCost = lossFn(u_out, batch_u_exact)
            DECost = lossFn(diffEqLHS, torch.zeros_like(diffEqLHS))
            cost = uCost + DECost

            cost.backward() # perform back propagation
            optimiser.step() # update parameters
            optimiser.zero_grad() # reset gradients to zero

        scheduler.step(cost) # update scheduler, reduces learning rate if on plateau
        # store cost, lambda- and nu-value of each epoch
        costList.append(cost.detach().numpy())
        lmbdaList.append(network.lmbda.item())
        nuList.append(torch.exp(network.nu).item())

    network.train(False) # set network out of training mode
    return costList, lmbdaList, nuList
```

Figure C.38: Code to train a neural network to solve Burger's equation and simultaneously approximate its unknown parameters λ and ν.

```
# load and format sample data (dictionary) for u(x,t), there are 25600 samples in total
data = scipy.io.loadmat('burgersData.mat')
t = data['t'].flatten()[:,None]
x = data['x'].flatten()[:,None]
Exact = np.real(data['usol']).T
X, T = np.meshgrid(x,t)

XT = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
u_exact = Exact.flatten()[:,None]
numSamples = 2000 # number of training samples

try: # load saved network if possible
    checkpoint = torch.load('burgersSimultaneousTanh32bit.pth')
    epoch = checkpoint['epoch']
    network = checkpoint['network']
    trainData = checkpoint['trainData']
    optimiser = checkpoint['optimiser']
    scheduler = checkpoint['scheduler']
    costs = checkpoint['costs']
    lmbdas = checkpoint['lmbdas']
    nus = checkpoint['nus']
    print("model loaded")
except:
    try: # load initial state of model
        checkpoint  = torch.load('burgersTanhInitialNetwork.pth')
        network     = checkpoint['network']
        trainData   = checkpoint['trainData']
        print("initial model loaded")
    except:  # create new model and save its initial state
        network     = BurgersEquationSolver(numHiddenNodes=32, numHiddenLayers=8)
        trainData   = DataSet(XT, u_exact, numSamples)
        checkpoint  = {'network' : network, 'trainData' : trainData}
        torch.save(checkpoint, 'burgersTanhInitialNetwork.pth')
        print("new model created")
    epoch = 0
    optimiser = torch.optim.Adam(network.parameters(), lr = 1e-3)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimiser, factor = 0.5, patience = 500,
                threshold = 1e-4, min_lr = 1e-6, verbose = True)
    costs, lmbdas, nus = [], [], []

trainLoader = torch.utils.data.DataLoader(dataset=trainData, batch_size=numSamples, shuffle=True)
lossFn   = torch.nn.MSELoss()

numTotalEpochs = 100000
numEpochs = 10000 # number of epochs to train each iteration
while epoch < numTotalEpochs:
    newCost, newLmbda, newNu = train(network, lossFn, optimiser, scheduler, trainLoader, numEpochs)
    costs.extend(newCost)
    lmbdas.extend(newLmbda)
    nus.extend(newNu)
    epoch += numEpochs

    checkpoint = {'epoch': epoch, 'network': network, 'trainData' : trainData, 'optimiser': optimiser,
                  'scheduler': scheduler, 'costs': costs, 'lmbdas' : lmbdas, 'nus' : nus}
    torch.save(checkpoint, 'burgersAdam2.pth') # save network every 'numEpochs' epochs
```

Figure C.39: Code to format training data and run training for 100,000 epochs on a neural network to solve Burger's equation and simultaneously approximate its unknown parameters λ and ν.

```python
def trainU(network, lossFn, optimiser, scheduler, loader, numEpochs):
    """
    A function to train a neural network to approximate the solution u(x,t) to Burger's equation
    based on sample data from the exact solution

    Arguments:
    network (Module) -- the neural network
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    scheduler (Learning Rate Scheduler) -- reduces learning rate if cost value is plateauing
    loader (DataLoader) -- generates batches from the training dataset
    numEpochs (int) -- number of training epochs

    Returns:
    costList (list of length 'numEpochs') -- cost values of all epochs
    """
    costList=[]
    network.train(True) # set network into training mode
    for _ in range(numEpochs):
        for batch in loader:
            input, batch_u_exact = batch # separate (x,t) and u(x,t) values
            u_out = network.forward(input) # pass batch of values (x,t) through network

            cost = lossFn(u_out, batch_u_exact) # calculate cost
            cost.backward() # perform back propagation
            optimiser.step() # update parameters
            optimiser.zero_grad() # reset gradients to zero

        scheduler.step(cost) # update scheduler, reduces learning rate if on plateau
        costList.append(cost.detach().numpy())  # store final cost of each epoch

    network.train(False) # set network out of training mode
    return costList
```

Figure C.40: Code to train a neural network to approximate the solution to Burger's equation based on sample data from the exact solution.

```
def trainDE(network, lmbda, nu, lossFn, optimiser, scheduler, loader, numEpochs):
    """
    A function to approximate the parameter values lambda and nu in Burger's equation
    using a neural network which has been trained to approximate the solution function

    Arguments:
    network (Module) -- the neural network
    lmbda (tensor of shape (1)) -- the parameter lambda
    nu (tensor of shape (1)) -- the parameter nu
    lossFn (Loss Function) -- network's loss function
    optimiser (Optimiser) -- carries out parameter optimisation
    scheduler (Learning Rate Scheduler) -- reduces learning rate if cost value is plateauing
    loader (DataLoader) -- generates batches from the training dataset
    numEpochs (int) -- number of training epochs

    Returns:
    costList (list of length 'numEpochs') -- cost values of all epochs
    lmbdaList (list of length 'numEpochs') -- lambda values of all epochs
    nuList (list of length 'numEpochs') -- nu values of all epochs
    """
    costList, lmbdaList, nuList = [], [], []
    network.train(True) # set network into train mode
    for batch in loader:
        # calculate u(x,t) and its derivative only once
        input, batch_u_exact = batch # separate (x,t) and u(x,t) values
        u_out = network.forward(input) # pass batch of values (x,t) through network

        # calculate first- and second-order partial derivatives of network output
        du = grad(u_out, input, torch.ones_like(u_out), retain_graph=True, create_graph=True)[0]
        d2u = grad(du, input, torch.ones_like(du), retain_graph=True, create_graph=True)[0]

        # separate partial derivatives to attain u_t, u_x, u_xx
        # use torch.split to preserve grad history
        u_x, u_t = torch.split(du, 1, dim =1)
        u_xx, u_tt = torch.split(d2u, 1, dim =1)

        for _ in range(numEpochs): # with u and its derivatives fixed, train lambda and nu
            # since we know nu will always be positive, we train with exp(nu)
            diffEqLHS = u_t + (lmbda * u_out * u_x) - (torch.exp(nu) * u_xx)

            cost = lossFn(diffEqLHS, torch.zeros_like(diffEqLHS))

            cost.backward(retain_graph = True) # perform back propagation
            optimiser.step() # update parameters
            optimiser.zero_grad() # reset gradients to zero
            scheduler.step(cost) # update scheduler, reduces learning rate if on plateau

            # store final cost, lambda- and nu-values of each epoch
            costList.append(cost.item())
            lmbdaList.append(lmbda.item())
            nuList.append(torch.exp(nu).item())

    network.train(False)
    return costList, lmbdaList, nuList
```

Figure C.41: Code to approximate the parameters λ and ν from Burger's equation using a neural network which has been trained to approximate the solution function.

References

[1] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018. URL: http://neuralnetworksanddeeplearning.com.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[3] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[4] Tom Archibald, Craig Fraser, and Ivor Grattan-Guinness. "The History of Differential Equations, 1670–1950". In: *Oberwolfach Reports* 1.4 (2005), pp. 2729–2794.

[5] Randall J LeVeque. "Finite difference methods for differential equations". In: *Draft version for use in AMath* 585.6 (1998), p. 112.

[6] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178. URL: https://arxiv.org/abs/physics/9705023.

[7] The PyTorch Foundation. *PyTorch*. Version 1.12.1. June 28, 2022. URL: https://pytorch.org/.

[8] Isaac Hayden. *Neural Networks for Solving Differential Equations*. https://github.com/Isaac-Somerville/Neural-Networks-for-Solving-Differential-Equations. 2023.

[9] Meenal V Narkhede, Prashant P Bartakke, and Mukul S Sutaone. "A review on weight initialization strategies for neural networks". In: *Artificial intelligence review* 55.1 (2022), pp. 291–322.

[10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: 9 (May 2010). Ed. by Yee Whye Teh and Mike Titterington, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[11] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[12] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. "Understanding the exploding gradient problem". In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[13] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. "Automatic differentiation in machine learning: a survey". In: *CoRR* abs/1502.05767 (2015). arXiv: 1502.05767. URL: http://arxiv.org/abs/1502.05767.

[14] Griewank Andreas and Walther Andrea. "Evaluating derivatives". In: *Society for Industrial and Applied Mathematics* (2008).

[15] Jeffrey Siskind and Barak Pearlmutter. "Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation". In: *Optimization Methods and Software* 33 (Aug. 2017). DOI: 10.1080/10556788.2018.1459621.

[16] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. URL: https://www.sciencedirect.com/science/article/pii/089360809190009T.

[17] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80131-5. URL: https://www.sciencedirect.com/science/article/pii/S0893608005801315.

[18] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[19] NumPy Developers. *NumPy*. Version 1.23.5. June 22, 2022. URL: https://numpy.org/.

[20] The PyTorch Foundation. *PyTorch 2.0 Documentation*. URL: https://pytorch.org/docs/stable/index.html.

[21] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. In: *Computing in Science and Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[22] M. Riedmiller and H. Braun. “A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm”. In: *IEEE Transactions on Neural Networks* 1 (1993), pp. 586–591. URL: http://dx.doi.org/10.1109/ICNN.1993.298623.

[23] Tieleman, T. and Hinton, G. *Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude*. Coursera: Neural Networks for Machine Learning, 4, 26-31. 2012.

[24] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2015. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[25] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: https://arxiv.org/abs/1609.04747.

[26] Guy Hacohen and Daphna Weinshall. “On the power of curriculum learning in training deep networks”. In: *International Conference on Machine Learning* (2019), pp. 2535–2544.

[27] Xiaoxia Wu, Ethan Dyer, and Behnam Neyshabur. *When Do Curricula Work?* 2021. arXiv: 2012.03107 [cs.LG].

[28] Lawrence C Evans. *Partial differential equations*. Vol. 19. American Mathematical Society, 2022.

[29] Justin Sirignano and Konstantinos Spiliopoulos. “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of Computational Physics* 375 (Dec. 2018), pp. 1339–1364. DOI: 10.1016/j.jcp.2018.08.029. URL: https://arxiv.org/abs/1708.07469.

[30] Cedric Flamant, Pavlos Protopapas, and David Sondak. “Solving Differential Equations Using Neural Network Solution Bundles”. In: *CoRR* abs/2006.14372 (2020). arXiv: 2006.14372. URL: https://arxiv.org/abs/2006.14372.

[31] Zdzislaw E Musielak and Billy Quarles. “The three-body problem”. In: *Reports on Progress in Physics* 77.6 (2014), p. 065901.

[32] Kaichao You et al. “How does learning rate decay help modern neural networks?” In: *arXiv preprint arXiv:1908.01878* (2019).

[33] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. “Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations”. In: *CoRR* abs/1711.10566 (2017). arXiv: 1711.10566. URL: http://arxiv.org/abs/1711.10566.

[34] C Basdevant et al. “Spectral and finite difference solutions of the Burgers equation”. In: *Computers and Fluids* 14.1 (1986), pp. 23–41. ISSN: 0045-7930. DOI: https://doi.org/10.1016/0045-7930(86)90036-8. URL: https://www.sciencedirect.com/science/article/pii/0045793086900368.

[35] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. “Searching for Activation Functions”. In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.

[36] Kian Katanforoosh and Daniel Kunin. “Initializing neural networks”. In: *deeplearning.ai* (2019). URL: https://www.deeplearning.ai/ai-notes/initialization/index.html.