Ai4Sci



Scientific Machine Learning

AI for mathematics, and mathematics for AI

Chris Budd OBE, mascjb@bath.ac.uk

and

Aengus Roberts asr66@bath.ac.uk

Dept of Mathematical Sciences
University of Bath
Bath, BA2 7AY

Contents

0. Foreword
1. An introduction to scientific machine learning
2. Machine learning architectures, and generative AI
3. Machine learning for differential equations I: PINNS and the DRM
4. Machine learning for differential equations II: Neural operators and their applications
5. Neural ODEs and discovering dynamics with machine learning
6. A case study in the use of PINNS
7. Useful references

0. Foreword

This guide has been developed for the Knowledge Exchange Hub in Mathematical Sciences (KE Hub) to provide mathematical scientists in Academia and/or in Business, Industry and Government (BIG) with a simple to understand introduction to Scientific Machine Learning (SciML).

The intended audience are those who need to use AI for what may be described as 'scientific applications'. Rather than exploring all of AI we focus on the rapidly growing field of SciML. This is a set of specific AI tools designed to look at, and solve, problems, which have at their heart a scientific formulation. These may include problems formulated using differential equations, inverse problems, problems in identification, control, and engineering design. The sort of questions that we will address are 'can a machine learning method make an accurate medical diagnosis', 'can machine learning forecast the weather', and 'can machine learning reliably design an aeroplane' (for the record, at the current state of development the answers are: yes, maybe, and not yet).

In this guide we will introduce practitioners to the algorithms of SciML, discuss how to use them, their strengths, weaknesses, and limitations, and explain what to expect in future developments. The guide provides a mix of theory, applications, and a concrete set of case studies. The emphasis will always be on the explainability, effectiveness, and trustworthiness of these methods.

It is fair to say that SciML has yet to achieve the same level of reliability as more established tools, such as the Finite Element Method, but the ease of using SciML methods, the impressive results that they can achieve, and the speed at which these methods are developing, make it worth the investment of time to see what might be appropriate for the scientific problems that you may be encountering in your work. At the same time, like any new tool, they should be used with a degree of caution. We hope that this guide will achieve the objectives of showing you what is possible, while also allowing you to make careful judgements of whether a SciML tool is right for the problem that you wish to solve, and how to solve it in a responsible manner.

By its very nature, SciML works best when those using it are able to combine the best ideas of both science *and* AI. As such, this guide is meant for those who have a basic scientific and mathematical training, as they will be the most likely practitioners for the sort of problems that SciML is best placed to address. However (unlike many papers and books on machine learning) we have tried to keep the jargon, and the mathematical technicalities, to the minimum level needed to both understand the theory behind the methods, and to use them in a responsible manner. The guide will be illustrated throughout using scientific problems, usually posed in terms of differential equations.

The guide starts with a general introduction to SciML. We then discuss the various AI architectures used to solve scientific problems. We follow this with two sections looking at SciML methods for solving differential equations, leading to a description of methods for inferring the nature of a system given data measurements. The final section contains

a set of case studies linked to a GitHub repository. We urge you to try these to enrich your understanding of this rapidly growing, and exciting, branch of artificial intelligence.

Chris Budd OBE, and Aengus Roberts. University of Bath

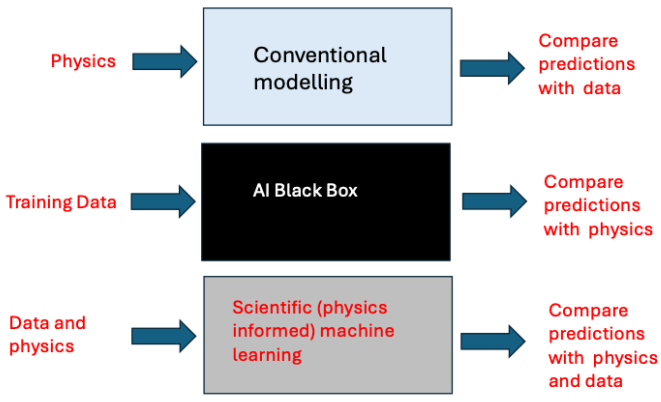
October 2026

1. Introduction

1.1 Overview

AI, and in particular SciML, is rapidly making a huge impact on nearly all areas of mathematics, science, and engineering. These applications include classifying images, solving differential equations, solving inverse problems, predicting the weather, protein folding, and constructing a (dynamical) model to fit some data. All of these areas have the potential to greatly benefit science, and engineering, and many aspects of Business, Industry, Charities, and Government Departments.

Some reasons for this are (i) that AI methods (based usually on neural network (NN) architectures) are very *expressive* so that complex systems can be represented by a NN with (relatively) few, trainable, parameters, (ii) they make very effective use of data, (iii) good software exists to set them up and train them, and (iv) they are effective in higher dimensional problems, where more conventional methods struggle. However, this popularity comes at a price. At the moment, such methods are often seen as opaque black boxes, where it is hard, or impossible, to see what is going on inside them. It is also difficult (unlike say more transparent Finite Element Methods) to get reliable error estimates for their use, they can hallucinate (produce very wrong answers), and the output of such methods can be unstable to small perturbations. We can compare the use of such opaque black box methods with more 'conventional, and transparent' scientific 'white box' modelling which aims to use human ingenuity to derive the laws of physics, and to then solve a system (usually through some form of numerical discretisation). Whilst this methodology has been used with success for many years (since Newton if not earlier) it can be very hard to implement, and often does not make effective use of data. SciML (illustrated below) is a 'grey box' approach that fuses these two approaches, and aims to produce reliable AI-based methods, with guaranteed accuracy. In these notes we will introduce you to some popular SciML methods, and will discuss the strengths and weaknesses of each.



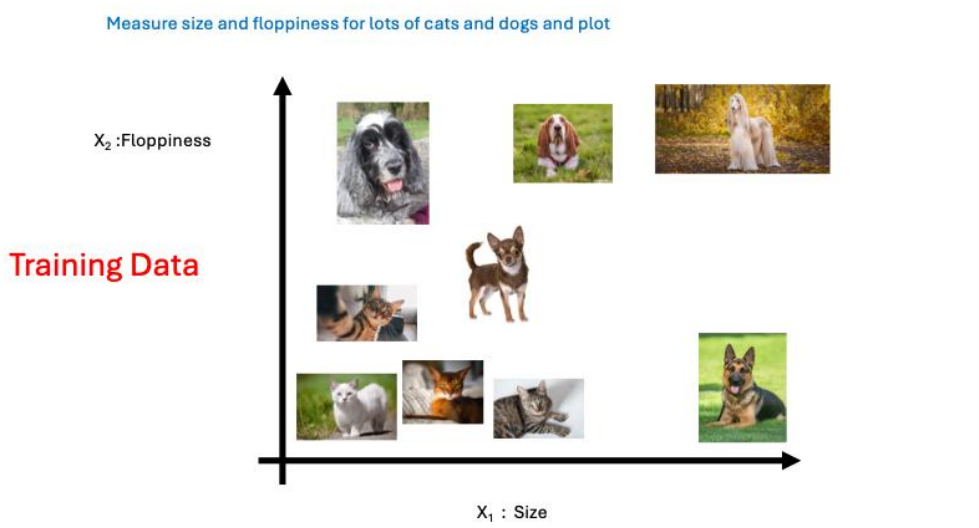
Good accounts of the general use of machine learning (ML) techniques, and, in particular the use of Python to implement the ML algorithms may be found in Lynch [1], Chollet [2], and an excellent account, written for (applied) mathematicians is presented in Higham & Higham [3] .

1.2 A very simple example of SciML in action

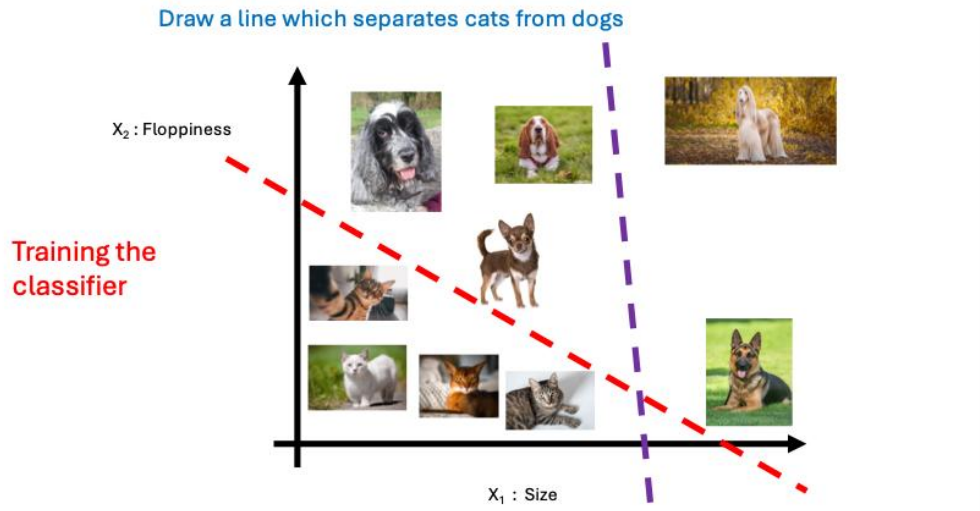
Let's suppose that we want to train a computer to distinguish a cat from a dog. Naively we might think that, as dogs are bigger than cats, all we need to do is to measure the *size* of the image, and make a decision on this single measurement. However, as the picture below shows, some cats are bigger than some dogs.



Thus we need to introduce another measurement, which we will call the *floppiness* of the image. We may then take a large 'training' set of classified images of cats and dogs and plot them according to their size and floppiness. The result might look something like this:



We may now try to separate cats from dogs by drawing a straight line through the training data. This line will have the property that any image *above it* is a dog, and every image *below it* is a cat. We have done this on the image below. The blue dashed line does a poor job at separating the data between the cats and the dogs, whereas the red dashed line has done a good job.



Now to tell a cat from a dog in the future, all we need to do is to plot its image as above, and see which side of the red line it lies on.

We now describe what is going on in more mathematical terms. If x_1 is the *size* of the image, and x_2 is the *floppiness* of the image, then a line in the data space is given by

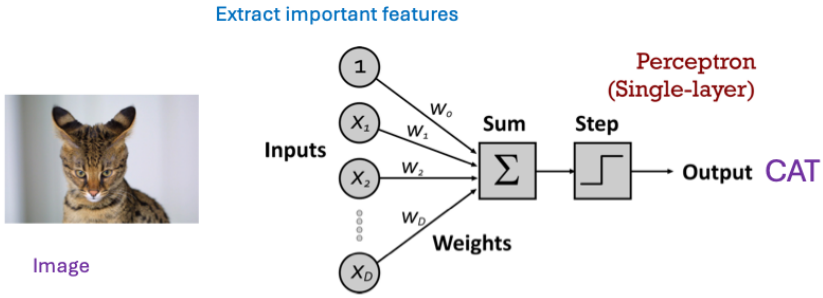
$$w_1 x_1 + w_2 x_2 + w_0 = 0.$$

Images above the line 'dogs' have $w_1 x_1 + w_2 x_2 + w_0 > 0$ and images below 'cats' have $w_1 x_1 + w_2 x_2 + w_0 < 0$. The coefficients w_1,w_2, and w_0 have to be 'trained' so that these inequalities hold for each of the labelled training images. Once we have done this, then for any other image we can tell whether it is a cat or a dog by evaluating this expression. More generally, we introduce an *activation function* $\sigma(z)$, which takes us from the expression

$$z = w_1 x_1 + w_2 x_2 + w_0,$$

to a classification of the image. In this case the activation function is the Heaviside (or step) function which satisfies the condition $\sigma(z) = 1$ if z > 0 and $\sigma(z) = 0$ if z < 0. The operation $\sigma(w_1 x_1 + w_2 x_2 + w_0)$ gives the value of 1 if we have a dog, and 0 if we have a cat. Hence we have a simple classifier for our image. This classifier is called a 'perceptron' , and this idea was first introduced in 1963. It is the simplest example of a *neural net*.

More generally if we have inputs $x_1, \ldots, x_D$ then the output has the form $\sigma(w_1 x_1 + \ldots + w_D x_D + w_0)$ and this is called a *shallow neural network of width D*. This is illustrated below.



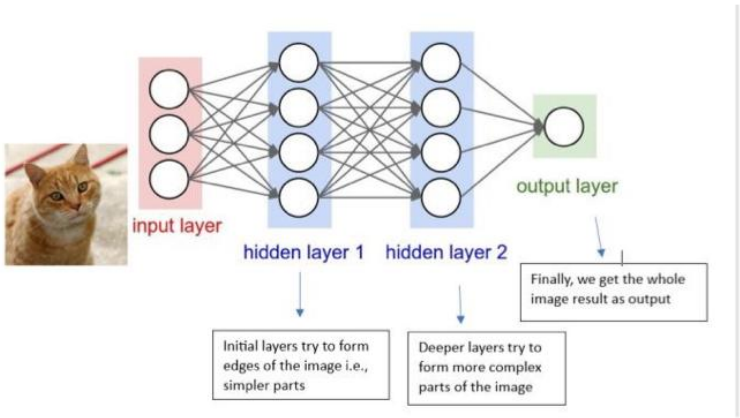
Basic architecture of a neural net classifier (1963)

The values of w_i for i=1,2,.. D are called the *weights*, and w_0 is called the *bias*.

Now, suppose that we want to classify a more diverse set of images (say cats, dogs, horses, cows, ..) , and also suppose that we can't do this by simply drawing a straight line through them. We need to use a more complex neural net to do this.

One way to do this is increasing the width D of the (shallow) neural network. However a much more efficient method is to use a *deep neural network* with many layers, where the output of one layer is fed into another layer, and then another etc. This is illustrated below.

Deep neural nets (2010) use many more layers and can be used to make much more complex decisions



In this example the first layer can be used to extract features from the image (say the outline of the ears of the cat), and the subsequent layers can then process these features to make the classification of the image.

In a general deep neural network, each layer has its own set of weights and biases, and they process the image in different ways. The number L of layers is called the depth of the network. The 'activation function' σ applied to the output of each layer, need not be the step function above. Popular choices are the piece-wise linear *ReLU* function $\sigma(x) = x^+ = (x + |x|)/2$ which is the generic choice for many applications, and the function $\sigma(x) = \tanh(\lambda x)$ which is used where differentiability is important, for example in the solution of differential equations. The choices of D,L and $\sigma(x)$ are called *meta-parameters*, which are fixed in advance, and are often chosen by experiments to simply see what works best. The set θ of all of the weights and biases are the free parameters of the network, which are determined by training.

We now introduce a few important definitions.

Architecture: The overall structure, and the way that the layers are connected together, is called the architecture of the neural network, and the choice of this depends on the task the network is to perform, and how expressive and transparent it is. The example above is a fully connected architecture. We will see in Section 2, how different architectures are used in other applications.

Expressivity: This is a measure of how well a perfectly trained network is able to do the task it is designed for, for example accurately classifying multiple images. The more expressive a network is, the more images we can classify (or the more differential equations we can solve).

Trainability: Training a neural network is the process of finding the parameters θ so that it does a pre assigned task. Trainability is a measure of how well we are able to do this.

Transparency: This is a measure of how well we understand what the network is doing, and the impact of the different parameters in the network on the output of the network. A major criticism of AI systems is that we don't know how they are making their decisions (for example in deciding whether to hire someone based on their CV). SciML methods, based on physics, are more transparent because we have a clearer understanding of what is going on inside them.

The deeper the network, the more expressive it is, but also the harder it is to train, and the less transparent it becomes. Using a deep neural network it is possible to classify many more types of images, or indeed any other form of data. An important example of this is the use of such architectures by doctors to diagnose clinical conditions given a medical image such as an X-Ray. As an example, Murphy et. al. [4] use a deep neural network to detect hip fracture very reliable from a raw X-Ray image.

1.3 Training a neural network

Training a neural network to find the optimal set of parameters is expensive, and often takes up a lot of time and energy! In the cats and dogs example we saw that we trained the parameters of the model to find a line which separated the data. This is an example of *supervised learning* where we have a pre-labled training data set. Other types of learning include *reinforcement learning* where there is some internal measure of success, such as finding the solution to a differential equation.

Training a neural network requires the following ingredients.

(i) A *training data set* x and an independent *validation data set* y. This can be made by partitioning the original data set. Typically an 80:20 split is good;

(ii) A *loss function* $\mathcal{L}(x, \theta)$ which you want to be close to zero on both the training and the validation set if the network is performing correctly (for example is it correctly identifying a dog as a dog);

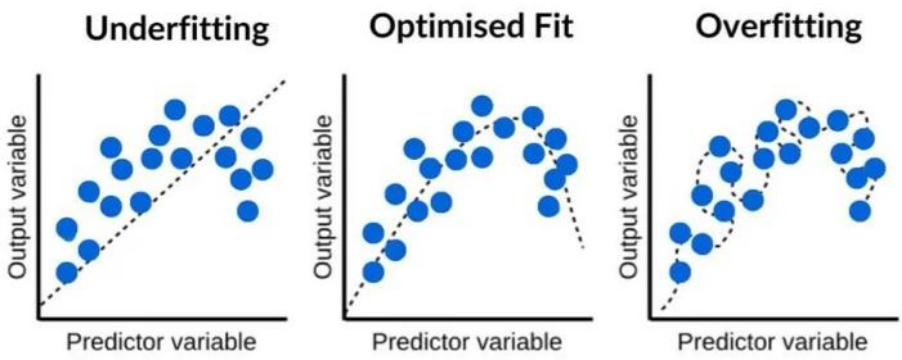
(iii) A measure of how $\mathcal{L}$ changes as you change θ. This is called *back-propagation* and is achieved in (say) PyTorch by automatic differentiation;

(iv) An *optimisation procedure*, together with a learning rate (which is generally found by experiment), which will determine the optimal parameters θ^* which minimise the loss function $\mathcal{L}$ over all elements of the training set x;

(v) A *validation step* in which the loss function is evaluated over y. This is used to check whether the optimisation process is working, and not *overfitting* the neural network to the training data (see later for a fuller explanation of this).

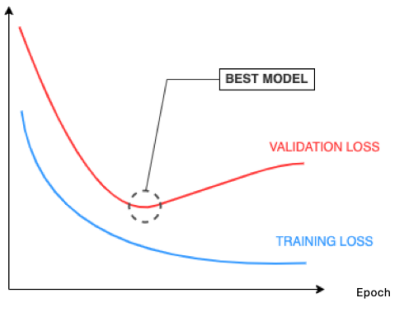
Step (iv) is usually done using some form of (stochastic) gradient descent method based on the calculations in (ii). Some excellent optimisers have been developed in recent years for ML applications, of which ADAM [3] is effective (usually), popular and easy to use.

Training is not easy. Firstly, the loss function $\mathcal{L}$ is, in general, a highly non-convex function of the parameters, and it is easy not to find the minimum of the loss function, or to get stuck in a local minimum. This is a primary cause of the 'hallucinations' of AI procedures, where the network does not perform as required, and misclassifications occur. The more information we can incorporate about the problem into the network design, for example by building in the physics, the better we can construct the loss function, and the more transparent the network will be. Secondly the optimisation problem can be ill-conditioned. This can make the convergence very slow and also highly sensitive to changes in the parameters. Thirdly, we can *overfit* to the training data x so that we fit to the noise in the data rather than to the signal. This is illustrated in the figure below where are trying to fit a smooth curve (which is the output of a neural network) through some noisy data. On the left a too simple approximation is used and the fitted curve does not approximate the data well. On the right, the approximation is too complex, and tries to go though all of the noise on the data. Such a curve would be useless when making future preductions In the middle we see the optimised fit which is a good predictor.



The

The impact of over fitting is reduced by using the validation set y (which should ideally be chosen to be significantly different from the training set x.) As the optimiser works through the process of finding parameters θ which minimise the loss function $\mathcal{L}$ on the training data x (typically by looking at random samples of this set) the loss on the validation set y is also monitored. Usually the loss function initially decreases with the iterations, on both x and y, as the network starts to fit to the data. However, it then typically starts to rise on the validation set as illustrated above on the figure below. This is because the optimiser is over fitting to the (noise on) the training set x. At the point when the loss function starts to rise on y the optimisation process is terminated at the best model, as illustrated below. The model is then optimally trained and can be used (and tested) on new data.



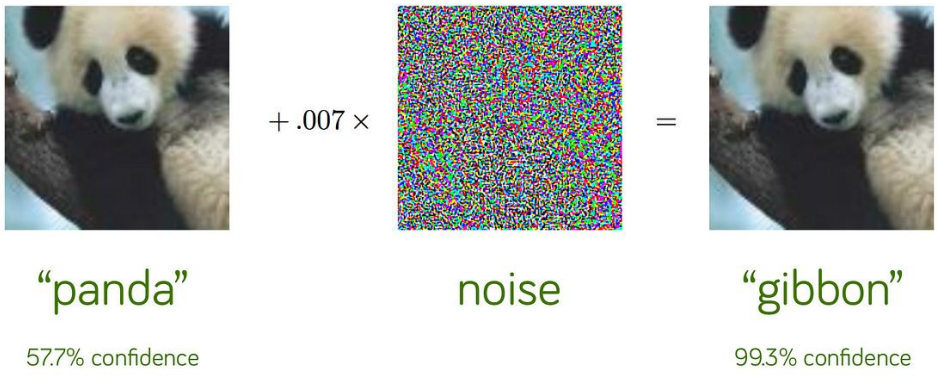
1.4 Some advantages and disadvantages of this approach.

Once trained a deep neural network can be very effective as an image classifier, or for carrying out many other tasks, including image classification, forecasting the weather, and solving certain differential equations. However there are certainly problems associated with this approach which we now describe.



Bias: The training can be influenced by bias in the training data set. As an example, the above image of a stop sign, was misidentified as a 45mph sign after training. The reason is that the training images for 45mph signs all had blue backgrounds, and the system then identified anything with a blue background as a stop sign. Clearly this is a serious issue in the use of AI in self-driving cars. Similar serious issues of bias have been seen in the (mis)use of AI systems for grading job applications, medical diagnosis, and in deciding on prison sentences. It is very important that all forms of bias are eliminated from a training set, and this is where heavy duty statistical testing is very important.

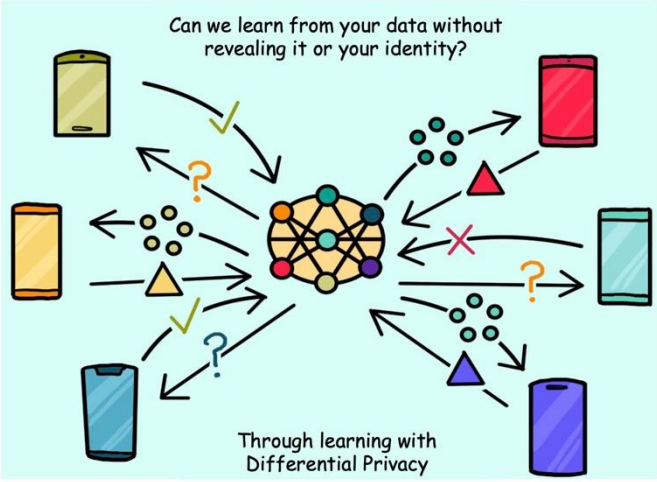
Adversarial Attack: When a neural network is trained from a data set, to recognise an image, it is acting as a black box, and it is often not clear how, given another image it makes its classifications. In particular, it can 'hallucinate' and make a completely 'wrong' classification. One impact of this uncertainty is that it can be subject to *adversarial attack*. In an adversarial attack a small, but carefully selected, perturbation is be made to the image to force an incorrect classification. This is an ill-conditioning problem in numerical analysis, as described in Bungert et. al [5]. As an example, we take a picture classified as a panda, add a small amount of noise (specifically designed to 'trick' the neural network) and get a picture which is then identified as a gibbon.



It is important to design image classifiers which are not vulnerable to such form of attack, as this greatly degrades their ability to make reliable classifications.

Moving away from the training data (stable system bias): We will see in Section 4 that SciML methods (specifically Neural Operators) can be very effective in forecasting the weather. But this begs the question of whether a weather forecast trained on Californian data could predict a storm in the UK? The answer is n, and this gets to the heart of many AI methods. They are (excellent) *interpolators* of data, but poor *extrapolators* from data. Learning only from data makes it hard to make decisions away from that data. This is why it is essential to augment the data-driven approach with extra physical information. We will see many examples of this in Sections 3, 4 and 5.

Data security, or not: When you train a neural network on a data set x, an echo of that training set is left on the parameters of the network. It is then often possible to trace back from these parameters to learn details of the training set. In principle it may be able to reverse engineer a neural network architecture to see what it has been trained on. This can cause a problem if the training data contains personal, and private, information, for example images from patient records, or data from a census. The methods of *differential privacy* produce synthetic data, which is based on the statistical properties of true data, but anonymises it, to train the neural network on. These methods developed, for example, by Carlin et. al. [6], are designed to keep the details of the training data secure.



In contrast, we might want to be able to detect whether neural network has been trained on illegal data, such as pornographic images, or images with copyright. At the moment this is possible, but difficult, using machine learning forensics.

1.5 In conclusion

In the next sections we will look more carefully at various methods for applying machine learning to a wide variety of scientific problems, and will look at the strengths and weakness of each. But the bottom line is: Maths is our best way of explaining how AI is working and answering questions about its limitations and its possibilities. AI doesn't replace maths, **AI is maths**!

2. Machine learning architectures, and generative AI

There are many different neural network architectures designed for both general and specific tasks (both scientific and well beyond). In this section we will attempt to summarise some of these architectures. A much more comprehensive account of these architectures, is described in Grohs and Kutyniok [7], and Drori [8].

2.1 The Multi-Layer Perceptron (MLP)

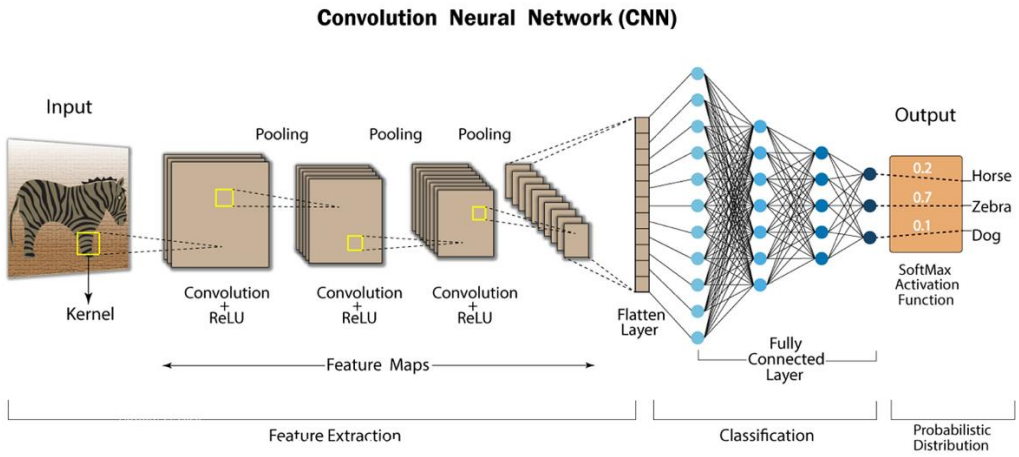
The basic architecture described in the Introduction is usually called a (fully connected) multi-layer perceptron or MLP. Such architectures can be adapted to a wide variety of tasks. Software packages such as PyTorch are set up to allow you to easily construct and then train, a MLP for a task. However, they may not be optimal choices for certain specific applications and we will discuss such applications next. What the MLP does well are task such as: the classification of data without any special structure, the approximation of high dimensional functions the solution of differential equations using Physics Informed Neural Networks (PINNS) and deep variational methods (see Section 3). Key to the success of the MLP is its 'expressivity', which (as mentioned in Section 1) its ability to represent a function in terms of its trainable parameters. A lot of work has gone into studying this problem, see [7], and we now have a good mathematical theory for it. To summarise this (without going into detail), if an MLP has width D, and depth L, and we use a continuous activation function such as the ReLU and tanh functions described in Section 1, then it is (in theory) possible to find a set of parameters θ so that the error in approximating a function f(x) is proportional to D^{-n} and to $e^{-\alpha L}$ for appropriate values of n and α. In other words, the error is a polynomialy decreasing function of the width D but an exponentially decreasing function of the depth L. This is in contrast with (say) a standard spline approximation method which only a exhibits polynomial decrease in the error. Here we see the power of using a deep method for the approximation. It offers very small errors in approximating a function with only a small number of parameters, and certainly far fewer parameters than more conventional. The catch behind this, is that training the MLP architecture to achieve this optimal result may be very difficult, if not impossible. See [7] for more details of the problems associated with training an MLP. This is why different, more trainable, and more relaiable, architectures have been designed to deal with specific tasks.

2.2 The Convolutional Neural Network (CNN)

The CNN is a neural network designed for image processing. Conventional image processing relies on applying linear filters to an image to extract features such as edges, and the reduce the effects of noise. If the image is thought of as a high dimensional vector **x** (with the terms of each such vector corresponding to the intensity of each of the pixels in the image), then the action of the linear filter corresponds to the operation A**x** where the matrix A is a *convolution operator* [7]. Such an operator acts locally on the image, applying the same operation to nearby clusters of pixels, and well defined, and sparse,

structure. This structure follows from the observation that nearby pixels in an image are often closely related to each other, but are often very different from more distant pixels. An example of a convolution operator is a low pass filter, such as a moving point average, which replaces a group of pixels with the average of their intensity.

As a convolution operator has a particular structure, it has far fewer free parameters than a general matrix of the same size. It is therefore much earier to find these parameters (and to train the operator). In a CNN this structure is exploited, and the first few layers apply a series of such operators, combined with (say) a ReLU activation function and an averaging (pooling) operator, to extract the essential features of the image (for example denoising it and then extracting the edges). The deeper layers can then have an MLP architecture to learn information from these features, so that the image can be identified from the extracted features. We saw this methodology in action in the cat and dog example in Section 1. There are two advantages of a CNN over a more general MLP. Firstly, it uses trusted image processing technology, which is well understood and predictable, to do reliable feature extraction. Secondly, as described, the convolutional operators have far fewer terms than a fully connected MLP and thus can be trained much more easily. PyTorch allows you to construct, and train, a CNN with ease.



2.3 The Graph Neural Network (GNN)

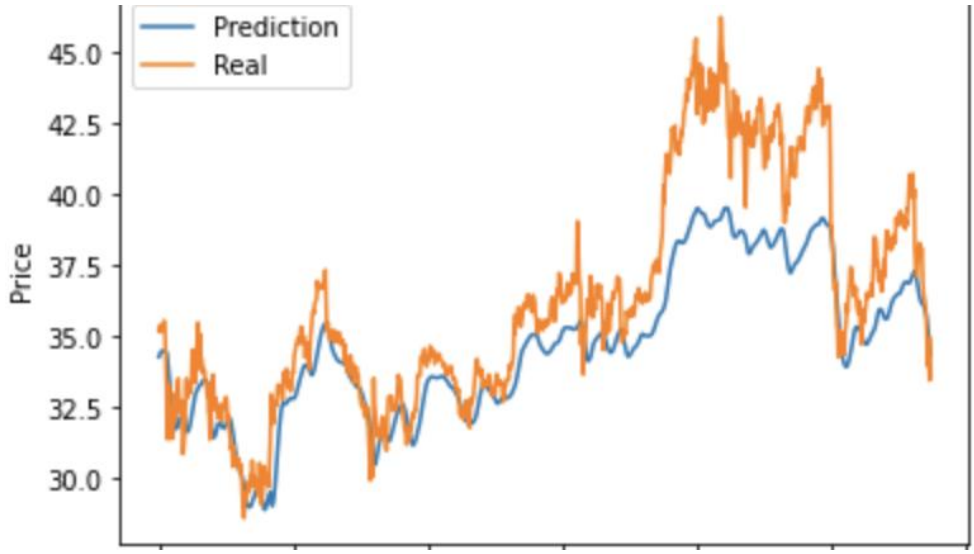
The CNN is effective because it treats nearby pixels as being related. In a GNN architecture this observation is taken further to look at more general data with terms related through the edges of a graph, of which the data points are nodes. In image processing, for example, we might want to relate all data points corresponding to a particular type of feature, such as human faces in a crowd. As well as in image processing, GNNs are used across various many other fields. These include the following. Molecular drug design where the efficacy of molecules is based on their atomic structure, Social network analysis where GNNs are use to identify communities and influential users. Modelling unstructured grid data and optimizing mesh relocation

in finite element methods (FEMs). Healthcare, where GNNs are used to identify progression pathways for diseases like Alzheimer's directly from patient records.

2.4 Recurrent neural networks (RNNs) and Long and Short Term Memory (LSTM) methods

Often the data we are interested in working with is not a single image, but is a stream of information x_i at times t_i. One example of this is in speech recognition, another is the location of taxis and their customers. Given any such time series for i=1 .. N an important question is, *can we predict future values of* x_i *for* $i = N+1 .. N+m$. Such a questions is of significant importance in for example, the financial industries, the power generation and transport industries, and in allocating resources in the emergency services. Traditionally this question of prediction in a time series has been addressed by statistical methods such as linear predictors or ARIMA (auto-regressive) methods. The use of neural networks allows us to bring more powerful methods to bear on this problem, which exploit both the high expressivity of the neural network, and the sequential nature of the input data. The primary tool for doing this is the 'recurrent neural network' (RNN)which are described in detail in [7]. RNNs are a type of neural network which take information from previous inputs, which then influences the current input and output, so they feed back into themselves. RNNs are used for prediction, translation, composition, speech recognition, and natural language processing. An RNN processes *input* data sequences x_i, via *hidden states* h_i ,to form sequences of *outputs* y_i. At every time step, an RNN updates its hidden state by combining the new input with the previous hidden state, passing it through activation functions, sharing the parameter matrices across all of the time steps. The parameter matrices for (say) a prediction are then trained using a suitable training sequence as before.

One of the problems with such an approach is a rapid loss of information (fading) through the network from one layer to the next, where data from early in the sequence gets diluted and overwritten by newer inputs. RNNs remember the immediate past well, but struggle to remember more distant events. During training via backpropagation, error signals which are sent backward through many time steps shrink exponentially. This means that the RNN cannot update its parameters based on early events, which makes training an RNN difficult. The Long and short term memory (LSTM) architecture is a particular example of an RNN, introduced by Hochreiter and Schmidhuber in 1997, designed to address the problem of information fading, by carefully storing the information at each stage and allowing control on what information is allowed to fade. LSTMs work well, and since their introduction, have established a good track record of use in applications [1,7]. In particular, LSTMs are proving very effective in prediction, and are used, for example, by the power industry and by Uber to predict future behaviour (such as customer demand) and to then allocate appropriate resources to cope with this demand. For more details of how LSTMs are used in practice, see [1] and Korstanje [10]. Below is an example from [1] of using an LSTM to predict the price of a stock:



2.5 Other architectures

Transformers

There are many other machine learning architectures, designed for both general and specific tasks, and we do not have space for a good description of them all here. We refer the reader to [8] for a comprehensive account. A recent, and explosive development, has been that of Transformer Networks. The 'T' in Chat-GPT refers to transformer, and transformer models may have trillions of parameters. Transformer networks are based on 'attention mechanisms' (see Vaswami [10]), and stacks of encoders passing information on to each other, usually performing computations in parallel. Transformer networks have state of the art performance for many tasks, starting with natural language processing, e.g. Large Language Models (LLMs), but extending their applicability to computer vision and audio processing as well as many other tasks.

Echo State/Reservoir Networks

Training a full neural network is a complex and expensive task. A much simpler process is to preset the coefficients in all the layers apart from the last, and to only train the output layer. This is a straightforward exercise in linear algebra. The other layers, often called the *reservoir* of the network, are usually given random parameters. Whilst much simpler to train than a usual network, such reservoir or 'echo state' networks have proved to be

remarkably effective in reproducing, and even predicting, the dynamics of chaotic dynamical systems.

2.6 Neural network based methods for inverse problems

Inverse problems are a key area in which scientific machine learning is making an important contribution. An example of an inverse problem is a medical imaging procedure, such as an MRI or a CAT scan, where noisy and incomplete, measurements are made, and from these measurements we have to infer the state of the system. Such problems are typically ill-posed (they usually have non unique solutions, and have extreme sensitivity to the input data), and require some form of *regularisation* to work.

As an important example of an inverse problem, is the linear problem of finding the state of a system x given measurement data y, where

$$A\,x = y + \text{noise}.$$

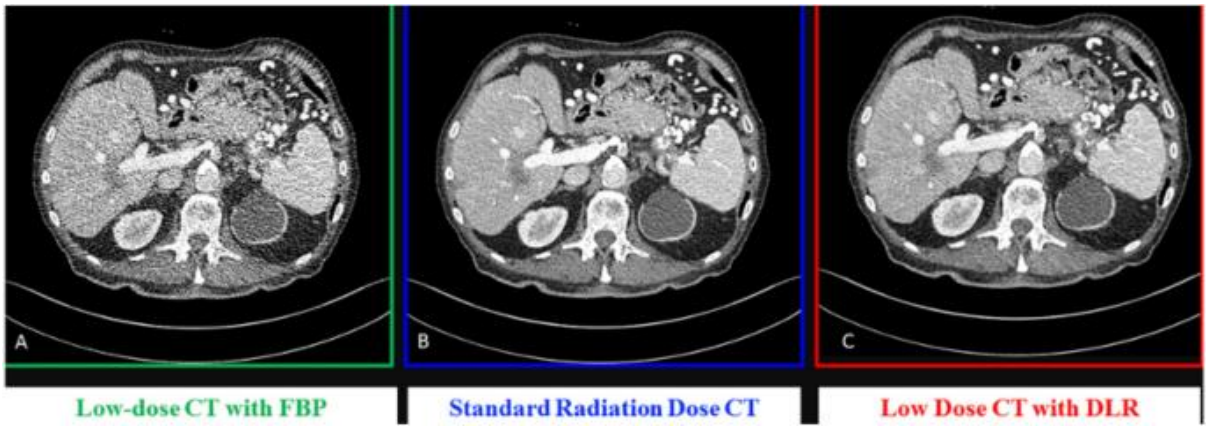
This problem lies at the heart of medical imaging using computer tomography (CT) and magnetic resonance imaging (MRI), and also remote sensing methods in the geo-sciences. In these applications A is usually an ill-conditioned matrix, and both x and y are high dimensional vectors (typically the dimension of y, whilst large, is much smaller than that of x).

The linear equation can be 'solved' by finding the vector x given by

$$x = \ argmin_x \ |A\ x\ - y|^2 + \mathrm{R(x)}.$$

Here the first term expresses the closeness of Ax to the data, and R(x) is the Regulariser which guides the solution procedure to finding a realistic state for the system. The choice of this is critical. Examples which do not use machine learning are (standard but of mixed effectiveness) Tikhonov regularisation where $\mathrm{R(x)} = ||x||^2$, and (the very effective but complex to apply) sparsity preserving Total Variation (TV) regularisation $R(x) = \| \ |\nabla x| \ \|$. However none of these regularisers take into account the qualitative nature of the solution. As an example a doctor looking at an MRI image of a knee would use their experience of typical shapes of a knee to select a good solution. This information of what makes an appropriate image can be included into a *learned regulariser* . In this procedure R(x) is trained in advance using a neural network approach, by solving the inverse problem on a labled set of images of known shapes. Learned regularisers are proving very effective not only in medical imaging but other areas such as the geo-sciences and even in art restoration. (See Hertrich et al. [11] for a review of the methodology of *deep learning reconstruction* (DLR).) Key benefits of using a learned regulariser are: faster computations, *lower radiation doses*, less image noise, and a better image quality (such as a better texture)

The figure below (from [11] with permission) shows three images CT images. The one on the left is constructed using standard CT methods showing low resolution. In the middle the resolution is improved with the standard CT method, but at the expense of a higher radiation does. On the right the DLR method is used, giving comparable resolution, but with a much lower radiation dose.



2.7 Generative AI

We conclude this section with a brief description of generative AI. This is perhaps the most rapidly growing (and controversial) area of machine learning, with direct impact on all of us.

In the problem in the introduction, we showed that we could train an ML system to classify an image given a set of training labels and images. Generative AI performs the opposite task. Given a (large) set of images which are assumed to be samples of a large distribution of similar images, Generative AI attempts to find the image from the overall distribution which most closely matches the given label. Generative AI is a very versatile tool. We can, for example, ask a generative platform to compose a poem about ML in the style of Brahms, write an essay for us on the same topic, or even give a lecture about it using the authors own voice.

The original methods used by Generative AI , used auto-regression to predict the next most likely part of a sequence, given a lot of training examples. This process is used in the auto-fill algorithms used in mobile phones. As an illustration, when faced with the sentence

The quick brown fox jumped over the lazy ...

We immediately say *Dog* as the next word. If we are now given the sentence

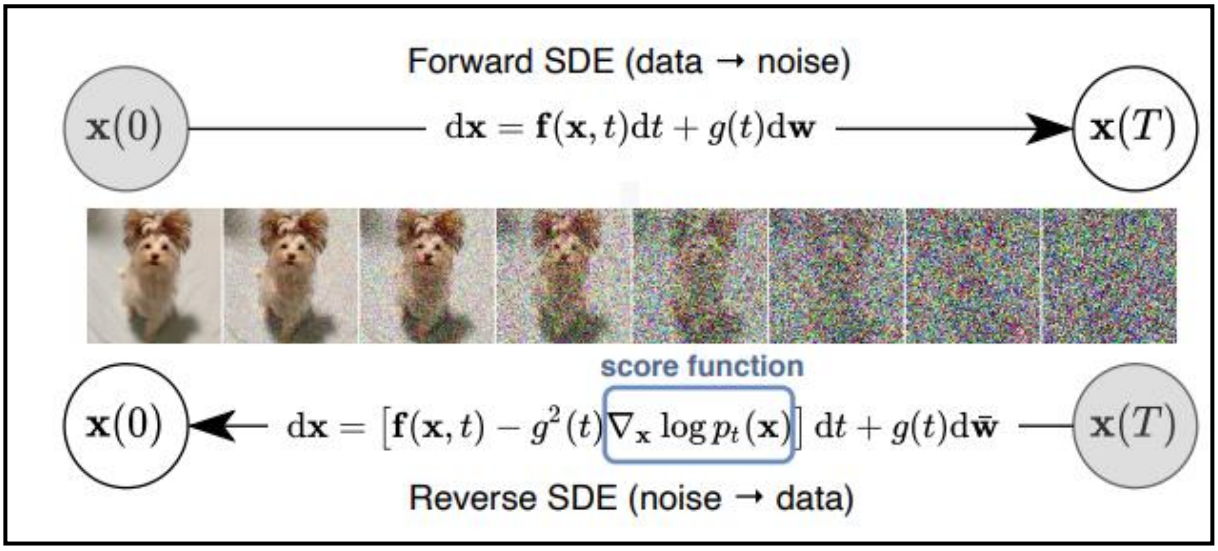
The quick brown fox jumped over the ..

then we fill in with *Lazy Dog* etc.

Similarly, given an image with a pixel missing, we can train an algorithm to fill in the most likely remaining value for the absent pixel. If an image has more pixels missing, this process can be repeated, with an auto-regressive algorithm successively filling in more and more values for the deleted pixels. In all cases the algorithm will be guided by the labels that we require the image to satisfy. Finally we can do the whole auto-regressive process starting from a random set of pixels. By doing this we will create a whole new image.

Such auto-regressive methods give good answers, and are they used in large language model such as Chat-GPT. However they are rather slow. They have been over taken or extended by transformers, and other technologies.

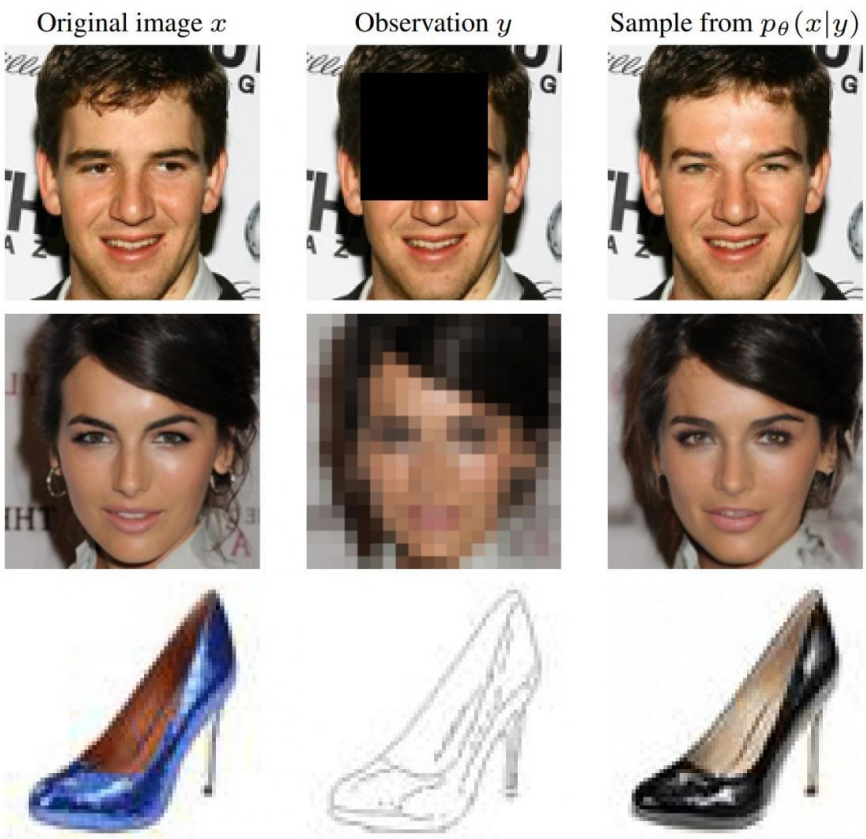
An alternative, fast, and very effective, approach to generative AI is to use stochastic differential equations, and (score based) stable diffusion methods. A 'classical' problem in image processing is de-noising in which given a noisy image we attempt to find an image which is close to the data but smoother than the original. This can be done in a learned manner, finding the most appropriate image from a distribution. More generally we can use this process to generate an image, taking a random start, and denoising it using a stochastic differential equation (conditioned by the labelling) to produce a new image.



A full description of how the stable diffusion methodology works is given in Batzolis et. al. [12], and the Wikipedia article https://en.wikipedia.org/wiki/Stable_Diffusion .

Below are some examples of image process using stable diffusion, taken from [12] with permission. In the top row we see an original image x, followed by the same image with part removed y, and finally the image in which the removed part is restored using stable diffusion (a process called in-painting). This is done by finding the closest (i.e most likely)

image from the training set, which is consistent with the data y. You can inspect this final image to decide how good it is. In the second row the original image x is blurred to give the image y, and then the original image restored using stable diffusion. In the final row the image x is processed so that only its outline remains. The original image is then restored. Note that this process has got the colour wrong! Similar diffusion based methods can be used to (for example) add in extra detail to a meteorological forecast of rainfall.



Stable diffusion methods are fast, and produce high quality photo-realistic 'fake' images from random seeds. Examples of these images (see [12] with permission) are shown below. Note that none of these images are real people.



This procedure raises significant ethical issues about the use of generative AI to produce fake images as seen by this fake picture (taken from an outreach postcard produced by Maths4DL) of the late *Pope in a Coat*. Such issues are of concern to policy makers, the media, and well beyond.



There is currently an ‘arms race’ between mathematical methods that can tell whether an image has been faked (by seeing the imprint of the machine learning algorithm), and the design of algorithms to produce images which cannot be detected as such.

Generalised Adversarial Networks (GANs) build this into their architecture, simultaneously training and image generator and an image discriminator, to produce ‘undetectable’ realistic images. These work well, but we must be ever mindful of the ethical issues of using generative AI in this way.

3. Machine Learning for Differential Equations I: PINNS and the DRM

Differential equations have existed for over 400 years as a method for describing dynamical systems. From planetary and particle motion to robotics control and drug development, ODEs and PDEs have often been the language of choice to mathematically model the behaviours observed by scientists, engineers and others. For almost as long, numerical methods have existed as a way to approximate solutions to these models when analytical methods fail.

The advent of the computer in the 20th century has rapidly accelerated the development of these methods, allowing numerical methods to solve problems over complex domains in two and three dimensions with strong guarantees on the accuracy of the approximated solution. These include, ODEs, boundary value problems and PDEs such as:

$$\frac{du}{dt} = u^2, \quad u(0) = 1,$$

$$-\varepsilon^2 \frac{d^2u}{dx^2} + u = 1, \quad u(0) = u(1) = 0,$$

$$iu_t + \nabla^2 u + u|u|^2 = 0, \qquad u(x,0) = u_0(x).$$

Where we impose suitable eg. Dirichlet, Neumann, or Periodic boundary conditions.

We will first introduce three "classical" (non-machine learning based) methods which aim to solve ODE/PDE problems of the form

$$u_t = F(x, u, \nabla u, \nabla^2 u)$$

with appropriate boundary conditions.

We will then introduce some *machine learning based numerical solvers* in particular PINNs (physics informed neural networks) and the DRM (Deep Ritz Method)

We will show how to use these methods, and give examples, before covering some of the theoretical literature surrounding their accuracy

3.1 Classical Numerical Methods

3.1.1 The Finite Difference Method

Finite Difference Methods (FDMs) approximate derivatives as the difference between neighbouring grid points. By picking a finite number of points across our domain, for example $x_i, \ i \in [1, \dots, N]$, we can approximate the function at these points by $u_i \approx u(x_i)$. This allows us to make approximations of derivates of the function such as

$$u'(x_i) \approx \frac{u_{i+1} - u_i}{h},$$

where h is the step size, given by $h = x_{i+1} - x_i$.

Applying this logic to the heat equation given by $\frac{\partial u}{\partial t} = \nabla^2 u$ we need to discretise in both time and space using the approximation $u_i^n = u(x_i, t_n)$ which in turn allows us to discretise the whole problem as:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{2}\left[\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}\right]$$

for example. Here we have used the Crank-Nicholson method which uses the mean of the 2nd spatial derivative at timesteps n and $n+1$ to evaluate the time derivative on the left side of the equation. This produces a scheme that is second-order accurate in time and unconditionally stable for diffusion-type problems, though it requires solving a linear system at each time step.

Error Estimates

The numerical error is the difference between the numerical approximation and the solution at the equivalent point. For FDMs this can be bounded and depends on the spatial and temporal step sizes together with a problem-dependent constant in the following way

$$\left|\left|u_i^n - u(x_i, t_n)\right|\right| < C(u)\Delta t^p \Delta x^q.$$

The error may be reduced (i.e. reducing $C(u)$ or increasing p or$q$) by redefining our grid discretisation such as with adaptive methods. These become useful when the solution to the problem varies rapidly requiring a finer grid size in certain sections of the domain.

Extending to higher dimensions can, however, cause issues. As we have seen, FDM requires a structured grid - typically a uniformly spaced rectangular mesh. In higher dimensions this gives many points in the domain to solve for, and this can be further exacerbated by irregular or complicated geometries. Here, we require a grid that covers the domain, often leading us to solve for grid points that fall outside the boundary.

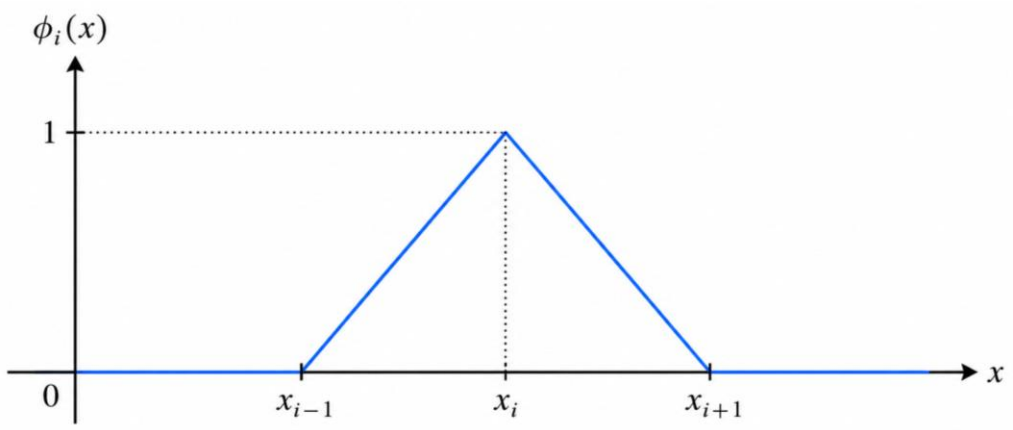
3.1.2 The Finite Element Method

Finite element methods (FEMs) provide us with a way to deal with complicated domains and provide a solution approximation as a function compared to the finite set of points we encountered in the previous chapter. FEMs aim to build a solution as a linear sum of simple basis functions where each basis function acts over a subset of the domain and

is zero elsewhere (a property known as local support). This gives us a solution approximation of the form

$$u(x,t) \approx U(x,t) = \sum_{i=0}^{N} U_i(t)\phi_i(x)$$

where $\phi_i(x)$ is zero everywhere except for $x \in [x_{i-1}, x_{i+1}]$ and explicitly zero on the boundary (i.e. $\phi_i(x_{i+1}) = 0$). In the interior of this region it might look like a hat function (if the approximate solution was to be piecewise linear) which looks like:



We can also make a different choice of basis functions, including higher-order polynomials and spline functions. In this chapter we restrict attention to piecewise linear basis functions for simplicity. The coefficients $U_i(t)$ represent the degrees of freedom of the approximation and are determined by solving a system of equations obtained from the weak formulation of the governing differential equation.

The Weak Formulation

To determine the u_i we aim to minimise the energy, defined as .

$$F(u) = \int_\Omega f(u, \nabla u, x)dx.$$

We generate an energy like this by integrating the weak form of the PDE, that is to multiply the PDE through by a test function $v(x,t)$ and using integration by parts to shift derivatives onto the test function. By then replacing $u(x,t)$ with $U(x,t)$ we can set up a system of linear equations which, when solved, will minimise the energy.

Error Estimates

Overall, FEMs offer several advantages compared to other numerical methods when solving PDEs on complex domains. The mesh can be adapted to the geometry of the problem, allowing irregular boundaries and local features such as boundary layers to be represented more naturally than with structured grids. A large ecosystem of mature software libraries also exists for FEM, making practical implementation straightforward.

In addition, theoretical results such as Céa's Lemma described in Ciarlet [13] provide guarantees on the accuracy of the finite element approximation. In particular, the FEM solution is quasi-optimal in the sense that the error is bounded by the best approximation available in the chosen finite element space, i.e.

$$\big||u - U|\big| \leq C \inf_{v_h \in V_h} \big||u - v_h|\big|.$$

This means that the finite element solution is, up to a constant factor C, as accurate as the best possible approximation that could be obtained using the chosen basis functions.

3.1.3 The Collocation Method

As with FEMs, in Collocation Methods, we approximate the solution using a linear combination of basis functions. However, rather than enforcing the ODE in an integral sense, we require that the ODE is satisfied exactly at a finite set of points within the domain, known as collocation points.

We again write an approximation of the form

$$u(x,t) \approx U(x,t) = \sum_{i=0}^{N} U_i(t)\phi_i(x).$$

Here $\phi_i(x)$ are chosen basis functions and $U_i(t)$ are coefficients that must be determined. Substituting this approximation into the governing differential equation produces a residual since the approximation will not satisfy the equation exactly everywhere.

In collocation methods we choose a set of points within the domain and require that the residual vanishes at these locations. For a general PDE , $D(u) = 0$, where D is the differential operator acting on $u(x)$, we therefore impose the condition

$$D\left(U(x_j,t)\right) = 0, \quad j = 1, \dots, N.$$

This generates a system of equations for the unknown coefficients $U_j(t)$. Solving this system yields the approximate solution.

The choice of collocation points and basis functions plays an important role in the stability and accuracy of the method. With the correct choice of collocation points, very high accuracy can be obtained. In practice, collocation is often used together with polynomial or spline basis functions, and in some cases spectral basis functions can be used to obtain very high accuracy for smooth solutions.

Error Estimates

Collocation methods also admit error bounds dependant on the number of basis functions used in the approximation. In particular, the error

$$|u - U|_{C^2} \leq C(u)N^{-\alpha},$$

where N is the number of basis functions and C(u) is a problem–dependent constant. By carefully choosing the collocation points it is often possible to reduce C(u) and increase α.

Collocation methods are widely implemented in numerical software for solving boundary value problems. For example, they form the basis of routines such as solve_bvp in Python's SciPy library and related packages such as COLSYS. However, Collocation methods are not well defined in more than one dimension and therefore are of limited use for high dimensional problems, which is precisely where machine learning methods do a great job.

3.1.4 Issues with Classical Methods

As we have seen, classical numerical methods such as finite differences, finite elements and collocation are well established and supported by strong theoretical convergence results. However, the accuracy of these approaches depends heavily on the discretisation of the domain. In particular, the choice and placement of mesh or collocation points can have a significant impact on the quality of the approximation, and capturing local features such as sharp gradients or boundary layers often requires careful mesh design.

In addition, these methods can require considerable effort to implement effectively, particularly for complex geometries or higher dimensional problems. Although modern software frameworks such as Firedrake and other PDE libraries simplify many aspects of implementation, constructing suitable meshes and discretisation remains non-trivial. Another issue is that all three approaches suffer from aspects of the curse of dimensionality, where the number of degrees of freedom required to resolve the solution grows rapidly with the dimension of the problem.

3.2 Machine Learning Methods I: Physics Informed Neural Networks (PINNs)

We now introduce Physics Informed Neural Networks (PINNs) which are a recent, machine learning (ML) based numerical solver proposed by Raissi et. el. in 2019 [14]**.** They were originally proposed as a "mesh-free" and self adaptive numerical method and therefore were less effected by the curse of dimensionality. Here we will give a brief introduction to them before moving onto the idea of a convergence theory, akin to the error estimates seen in the previous chapter and highlighting the current gaps in this theory.

3.2.1 What Are PINNs?

PINNs (Physics Informed Neural Networks) aim to use neural networks to find approximate solutions of differential equations of the form $Du = f \quad \text{on a domain } \Omega.$ with appropriate initial and boundary conditions. In particular they use a neural network as described in Section 1, with an input x (which may be a vector in d-dimensions) to construct an approximation $u_\theta(x)$ to the function $u(x)$ parametrised by the neural network coefficients θ. The reason for using a PINN is that the expressivity of the neural network (first and foremost the universal approximation theory, Cybenko [15]), means (in principle) that we can find a very good approximation $u_\theta(x)$ to $u(x)$ with (relatively) few parameters. This is especially useful for problems posed in high dimensions.

To construct a PINN we firtly consider the residual of this differential equation and take the norm of it defined by:

$$r(u) = \big||Du - f|\big|$$

To train the PINN we need to define a loss function. We observe that the residual above is minimised precisely when u is a solution to our original differential equation. If the output of the PINN is $u_\theta(x)$ we then aim to

Train the coefficients θ to *minimise the residual* $r\big(u_\theta(x)\big)$.

To do this we assume that the Neural Network architectures allows us to differentiate the function $u_\theta(x)$ at all points in the domain (a tanh activation function for example will let us to this). We the evaluate the residual at a *finite set of collocation points* within the domain and minimise the function obtained by summing the square of the residual over these locations. In this sense, PINNs closely resemble collocation methods, where the governing differential equation is enforced at a discrete set of collocation points

$$x_j, \quad j = 1 \dots N.$$

The key difference is that instead of approximating the solution using polynomial or spline basis functions, PINNs use a neural network to represent the solution. By doing this we obtain an interior loss function $\mathcal{L}_i$ which takes the form:

$$\mathcal{L}_i = \sum_{j=1}^{N} \left| Du_\theta(x_j) - f(x_j) \right|^2$$

Suppose that we also have Dirichlet boundary conditions of the form

$$u(x) = g(x)$$

on the boundary of the domain Ω. Then we extend (and scale) the interior loss function to include these via

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{j=1}^{N} \left| Du_\theta(x_j) - f(x_j) \right|^2 + \beta \frac{1}{M} \sum_{k=1}^{M} |u_\theta(x_k) - g(x_k)|^2 \quad (*)$$

Here we have N collocation points x_j within the domain Ω and M collocation points on the boundary of Ω. (For a one-dimensional problem the boundary would simplify to 2 points.) The β parameter here allows us to weight how much we want the boundary terms to bias our loss function and is often in practice chosen empirically based on the problem at hand. Initial conditions (as we will see) can be included into the loss function in a similar manner

The operation of the PINN then works as follows.

1. We set up a neural network (usually a multi-layer perceptron) with the parameters θ giving the weights and biases (and usually use a tanh activation function).

2. We then find the optimal value of these parameters by minimising the function defined by (*) (often experimenting with the width, depth, and β to get the best results).

3. The output of the PINN is then the function $u_\theta(x)$ which is defined at all points in the domain.

3.2.2 Examples

To get a better understand of PINNs we will work through two examples of converting the differential equation problem statement into a loss function we can minimise.

Example 1: A Two-Point Boundary Value Problem

We consider the regular two-point boundary value problem

$$-\partial_{xx}u = f(x, u, \partial_x u), \qquad u(0) = a, \qquad u(1) = b,$$

In a PINN approach, we approximate the solution using a neural network $u_\theta(x)$. The differential equation residual is then

$$r(x;\theta) = \partial_{xx}u_\theta + f(x, u_\theta, \partial_x u_\theta).$$

The network is trained by minimising the loss function

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{j=1}^{\mathrm{N}} \left|r\left(x_j;\theta\right)\right|^2 + \frac{\beta}{2}(|u_\theta(0) - a|^2 + |u_\theta(1) - b|^2)$$

where x_j are the collocation points in (0,1). As described earlier, the first term encourages the network to satisfy the differential equation inside the domain, while the second term penalises violations of the boundary conditions.

Example 2: Solving Parabolic PDEs with PINNs

Consider the semilinear parabolic PDE

$$u_t = \partial_{xx}u + f(x, u, u_x), \qquad u(0,t) = a, \qquad u(1,t) = b \qquad u(x,0) = u_0(x).$$

One approach for solving such problems with PINNs is to combine the neural network approximation with a classical time-stepping scheme. Suppose we discretise time using steps $t_n = n\Delta t$ and denote the approximate solution by $U^n(x) \approx u(n\Delta t, x)$. Using an implicit time discretisation, the PDE becomes

$$\frac{U^{n+1}(x) - U^n(x)}{\Delta t} = \partial_{xx}U^{n+1} + f(x, U^{n+1}, \partial_x U^{n+1}) \equiv F(U^{n+1}).$$

We begin with the initial condition $U^0(x) = u_0(x)$, and at each time step train a neural network $u_\theta(x)$ to approximate $U^{n+1}(x)$. The residual for the PINN is then

$r(x;\theta) = u_\theta(x) - U^n(x) - \Delta t F(u_\theta)$, where $F(u_\theta) = \partial_{xx}u_\theta(x) + f(x, u_\theta, \partial_x u_\theta)$

Therefore the residual when solving for the approximation is given by

.

As in the previous examples, the network parameters θ are determined by minimising a loss function based on the residual evaluated at collocation points:

$$\mathcal{L}(\theta) = \frac{1}{\mathrm{N}}\sum_{j=1}^{\mathrm{N}}\left|r(x_j;\theta)\right|^2 + \frac{\beta}{2}(|u_\theta(0)-a|^2 + |u_\theta(1)-b|^2),$$

where x_j are collocation points in the spatial domain. Once the loss has been minimised, we set $U^{n+1}(x) = u_\theta(x)$ and repeat the procedure for the next time-step.

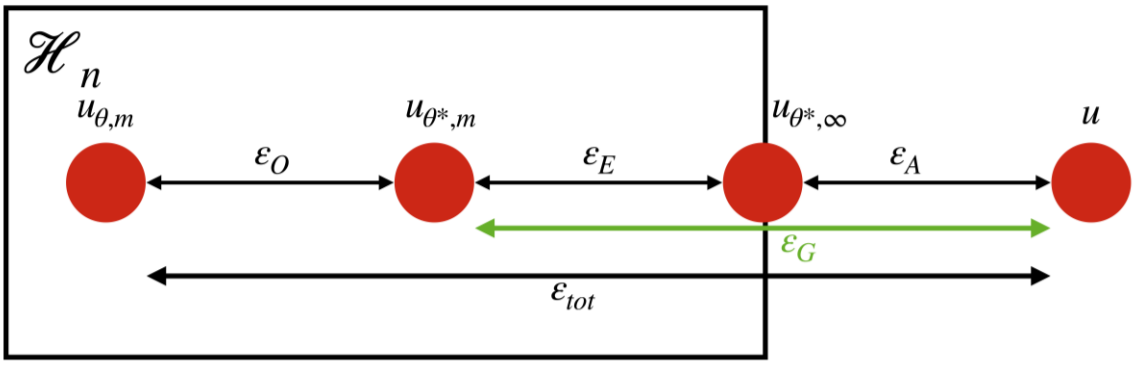
An alternative strategy is to use a PINN in which the neural network directly represents the solution as a function of both space and time, $u_\theta(x,t)$. In this case the residual

$$r(x,t) = \partial_t u_\theta - \partial_{xx} u_\theta - f(x, u_\theta, \partial_x u_\theta)$$

is evaluated at collocation points in both space and time. While this approach removes the need for explicit time-stepping, it can be more challenging in practice since the network must simultaneously learn behaviour in both space and time, which often have very different scales in (for example) parabolic problems.

3.3 A Convergence Theory for PINNs

As with the classical methods discussed earlier, we would like to understand how accurately a PINN approximation converges to the true solution of the underlying differential equation. While the theoretical understanding of PINNs is still developing, it is useful to decompose the total error into several distinct components. The figure below illustrates the relationship between these errors.



Let $u(x)$ denote the true solution of the differential equation. We consider a class $\mathcal{H}_n$ of all functions that can be represented by a neural network with n parameters θ

In practice, after training the network using a finite set of m collocation points, we obtain an approximation $u_{\theta,m}$ which differs from the true solution by an error. This error is influenced by several limitations in the learning process.

Firstly, the training procedure for the PINN may not find the optimal parameters within the chosen function class. The function $u_{\theta,m}$ is therefore typically less accurate than the

function $u_{\theta^*,m}$, which represents the result of perfectly minimising the loss function using the same finite set of collocation points. The difference between these two functions corresponds to the **optimisation error** ε_O

Even if perfect minimisation were achieved, the use of a finite number m of collocation points introduces another source of error in the PINN. If the residual of the differential equation were enforced at infinitely many points, we would obtain an idealised solution $u_{\theta^*,\infty}$. The difference between $u_{\theta^*,m}$ and $u_{\theta^*,\infty}$ represents the **estimation error** ε_E arising from the finite sampling of collocation points.

Finally, the PINN itself may not be able to represent the true solution exactly within the chosen function class $\mathcal{H}_n$. The difference between $u_{\theta^*,\infty}$ and the true solution u is known as the **approximation error** ε_A, which reflects the expressive power of the neural network.

The total error that we observe in practice can therefore be decomposed as

$$\varepsilon_{TOT} = \varepsilon_O + \varepsilon_E + \varepsilon_A.$$

Theoretical work for PINNs has bounded both the approximation error and estimation error (often coupled together as the **generalisation error** ε_G). However, there does not exist an easily obtainable bound on the optimisation error, and this often dominates the total error observed in practice.

3.4 Why do PINNs sometimes fail?

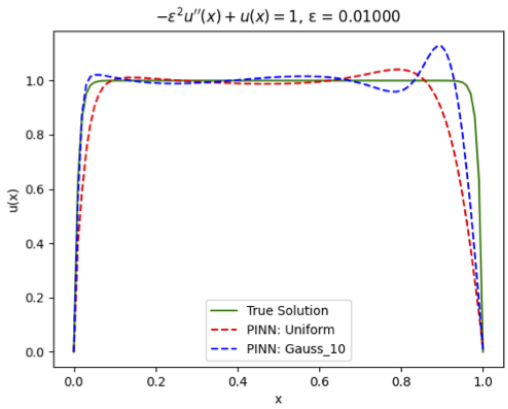
Although PINNs provide a flexible framework for solving differential equations, their practical performance can differ significantly from the theoretical picture outlined in the previous section. In particular, the *optimisation error* introduced during training can dominate the overall error and prevent the network from converging to an accurate solution. This is often due to a lack of convexity, and an ill-conditioning of the optimisation problems.

To illustrate this, consider the singularly perturbed two-point boundary value problem

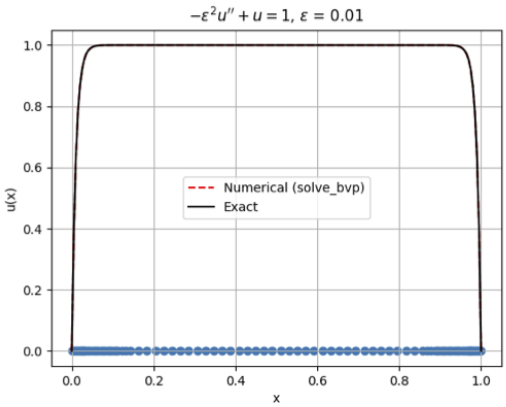
$$-\varepsilon^2 u_{xx} + u = 1, \qquad u(0) = u(1) = 0, \qquad 0 < \varepsilon \ll 1.$$

This problem exhibits boundary layers near x=0 and x=1 of width $O(\varepsilon)$ and we take $\epsilon = 0.01$.

We consider solving this with a PINN and give the results in the figure below (left), compared with those of a classical collocation method (right)



PINN



Collocation

We observe the PINN failing to reproduce the boundary layers and instead developing oscillations. One thing to note is that this failure occurs regardless of the choice of collocation points; both uniformly distributed and Gauss-Legendre points [16] produce similar behaviour. In contrast, a classical collocation solver resolves the boundary layers accurately.

This behaviour highlights one of the key issues with PINNs: although the neural network may have sufficient expressive power to approximate the solution accurately (i.e. the approximation error may be small), the training process may fail to locate this solution in practice. In terms of the error decomposition discussed previously, the optimisation error may dominate the overall error.

Recent research has identified several factors that contribute to these training difficulties. Firstly, PINNs often struggle to learn functions containing high-frequency or multi-scale features. This phenomenon is the *spectral bias* of neural networks issue we encountered. Neural networks tend to learn low-frequency components of a solution before higher-frequency components. Secondly, the loss function associated with PINNs is *highly non-convex*, meaning that optimisation algorithms may become trapped in poor local minima or flat regions of parameter space. Thirdly the optimisation problem itself may be highly ill-conditioned [17]

One cause of the non-convexity is due to weights and biases in the neural network performing vastly different roles as part of our approximate solution. Adding in the activation function to the network then creates a non-linear relationships between the network parameters and, in turn, the non-convexity we encounter.

These difficulties explain why, despite their theoretical expressivity, the optimisation error for PINNs cannot be bounded without adaptations made to the method. As a result, recent work has focused on improving the optimisation process, including the development of better training strategies, preconditioning techniques [17], and methods based on neural tangent kernels [18].

3.5 The Deep Ritz Method

An alternative neural network-based approach for solving partial differential equations is the Deep Ritz Method (DRM), introduced in 2017 by E and Yu [17]. One benefit of the method is that it implicitly deals with the spectral bias encountered with PINNs.

3.5.1 Definition and Motivation

Rather than minimising the residual of the differential equation as in PINNs, the Deep Ritz Method minimises an energy functional whose minimiser corresponds to the solution of the PDE. Let $u(x)$ denote the solution of a variational problem defined through an energy functional

$$F(u) = \int_{\Omega} f(u, \nabla u, x) dx.$$

Classically, many elliptic PDEs can be formulated as minimisation problems of this form. For example, the solution $u(x)$ of the Poisson equation $-\nabla^2 \mathrm{u} = \mathrm{f}$ with Dirichlet boundary conditions $u = g$, can be found by minimising the functional

$$F(v) = \int_{\Omega} \left(\frac{1}{2} |\nabla v|^2 - f(x) v \right) dx + \beta \int_{\partial\Omega} (v - \mathrm{g})^2 ds \qquad (**).$$

The Deep Ritz Method approximates the solution $u(x)$ using a neural network with output $u_\theta(x)$. To do this it uses a suitable optimisation method and seeks to find the function u_θ^* which *minimises the functional* (**) so that

$$u_\theta^* = \arg\min_{\theta} F(u_\theta).$$

Here the integrals used in the definition of F, are numerically approximated by using a set of quadrature points.

A key motivation for this formulation is that the optimisation is often better behaved in this case than when trying to minimise the PINN residual based loss function. The PINN loss involves multiple derivatives and multi-scale residual terms, which can lead to ill-conditioning and training difficulties. By instead minimising an energy functional, the Deep Ritz Method mitigates some of the optimisation issues associated with spectral bias and poor gradient flow observed in PINN training.

3.5.2 Comparison with FEM

The Deep Ritz Method is closely related to classical FEM, since both approaches minimise an energy functional defined over a space of admissible functions. In FEM, the approximation space is typically a finite-dimensional linear space spanned by basis functions associated with a mesh. In contrast, DRM replaces this with a nonlinear function class defined by a neural network.

This change introduces both advantages and challenges. Neural networks can provide highly expressive approximations and may achieve good accuracy with relatively few

degrees of freedom. They are also naturally mesh-free and can adapt to complicated geometries without requiring a structured discretisation of the domain.

However, the nonlinear parameterisation of the solution means that the optimisation problem becomes non-convex and often poorly conditioned. In contrast, many finite element discretisations lead to convex optimisation problems that can be solved reliably using linear algebra techniques. FEM also benefits from strong theoretical guarantees such as Céa's Lemma [13], which bounds the approximation error in terms of the best approximation available in the chosen finite element space. No comparable general result currently exists for neural network based methods such as the Deep Ritz Method.

As a result, while DRM can offer improved expressivity (so that it gives a more accurate result with fewer parameters) when compared with classical discretisations, it inherits many of the optimisation difficulties associated with neural networks.

4. Machine Learning for Differential Equations II: Neural Operators and their applications

4.1 Overview

In the previous section we saw how we can use a ML network such as a PINN or a DRM to find the pointwise solution of a differential equation. This an *unsupervised learning approach* in that the information given to the methods was only the differential equation itself. The *neural operator method* takes a different approach and makes use of *known solutions to the differential equation* to train the network. We describe this method as follows.

Suppose that we have a (partial) differential equation which we solve from an initial time t = 0 to a final time t = T so that:

$$u_t = f(x, u, u_x, u_{xx}), \quad u(x,0) = u_0(x), \qquad u(x,T) = u_T(x)$$

together with suitable boundary conditions.

The function u_T(x) is then a (possibly nonlinear and infinite dimensional) function

$$u_T(x) = F(u_0(x))$$

of the input function $u_0(x)$. We now assume as training data that we have a (large) number of *known* solutions $u_T^i(x)$ to the differential equation at time T , each corresponding to one of a large number of *different initial conditions* u_0^i for i = 1... N. We call two functions (u_0^i, u_T^i) a *solution pair*. Here the solutions in the training data set are accurately computed in advance using (say) a finite element or spectral method. We then aim to use these solution pairs to train a neural network (called a *Neural Operator*) to *emulate* the numerical method and find the (infinite dimensional function) F above. Then given a new initial condition u_0 as an input to the neural network it can find the future value u_T of the solution without having to solve the differential equation explicitly. In a crude way this can be thought of as interpolating the training data. Once so trained the Neural Operator is often very much faster in operation than the original numerical method. The advantage of this method over using a PINN is that it is trained on *solutions* of the differential equation with guaranteed error bounds. The disadvantages (shared with any ML method) are that the training process is slow and possibly unreliable, and that it is generally only accurate if the input u_0 is close to the training data. The use of additional physics can increase the accuracy of this method.

To construct such a Neural Operator suppose that we have a neural network f(u_0, θ) , with input (some finite dimensional representation of) the function u_0. Then we find the parameters in the Neural Operator by training the values of θ to minimise the loss function $\mathcal{L}(\theta)$given by

$$\mathcal{L}(\theta) = \sum_{i=1} \| \; u_T^i - f\left(u_0^i, \theta\right) \; \|$$

or similar.

4.2 Three examples of the Neural Operator Approach

(i) A linear ODE

Consider the linear ordinary differential equation

$$\frac{du}{dt} = A\,u, \;\; u(0) = \; u_0\,.$$

Where A is a constant matrix. This has the *exact* solution

$$u(t) = \; e^{At}\,u_0,$$

Where $e^A = \mathrm{I} + \mathrm{A} + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots$. Thus

$$u_T = F(u_0) \; \equiv B\,u_0 = \; e^{AT} u_0.$$

If we can find the *linear* operator B then we can find u_T for a general input u_0 *without having to solve the differential equation*. This is a relatively straightforward application of the training procedure described above.

Important note A neural network is a *nonlinear function of the input*. Thus it will approximate the linear operator B by a nonlinear operator. This may not be optimal in practice. If you know in advance that the operator is linear then it is possible to use a more direct method, such as the Moore-Penrose pseudo inverse, to approximate B using the training data.

(ii) The linear heat equation

The one-dimensional linear heat equation with *periodic boundary* conditions on the interval $[0,2\pi]$ is given by

$$u_t = u_{xx}\,, \;\; u(0,x) = \; u_0(x).$$

In this case we have that the solution at time T is given by a convolution (a global integral operator) of a Kernal function G(z) (in this case it is the error function) with the initial data, so that

$$u_T(x) = G * u_0 = \int_0^{2\pi} G(T, x-y)\, u_0\,(y)\, dy.$$

If function G can be determined (from training data as above) then we can find u_T from u_0 without having to solve the PDE.

There is a short cut to doing this which we will exploit presently. Suppose that $u_0(t,x)$ has a Fourier Series given by:

$$u_0(x) = \sum c_n\, e^{inx} \qquad \text{then} \qquad u_T(x) = \sum e^{-n^2 T} c_n\, e^{inx}$$

It follows that the Fourier coefficients of u_0 and u_T are related by the simple map:

$$c_n \;\rightarrow\; e^{-n^2 T} c_n.$$

Provided that we can find the Fourier coefficients of u_0 (say using the FFT) then we can easily learn this linear map from the training data. This observation motivates the FNO algorithm of the next section

(iii) The Darcy problem

The Darcy problem describes the velocity u(**x**) of a fluid through a permeable rock layer of permeability k(x) subject to a forcing f(**x**). It is given by:

$$-\nabla.\big(k(x)\,\nabla\, u(\boldsymbol{x})\big) = f(x).$$

This is a *linear elliptic partial differential equation* for u(**x**), and for a *given* function a(**x**) can be readily solved using the Finite Element Method or similar. This means that u(x) is a (readily computable) *nonlinear* function F(k) of the input data k(**x**). As above, we can learn this function using a Neural Operator such as the FNO described next.

4.3 The FNO method

There are now many methods for computing a Neural Operator. One of the first of these was the Deep-O-Net method, Lu et. al. [17], which projects the input u_0 onto a lower dimensional space, then computes the Neural Operator in a latent space, before projecting back up. The popular Fourier Neural Operator (FNO) method introduced in 2021 by Li et. al. [18], mimics the approach used to solve the linear heat equation by first sampling the input function at a set of special points to create a vector which is used as the input to a Neural Net, and then applying a sequence of Fast Fourier Transforms to the resulting vector.

The FNO architecture is based on the process of solving the linear heat equation, but also works for nonlinear problems. The FNO constructs a Neural Map f acting on u_0 parametrised by θ, which is a deep NN of depth D, constructed as follows:

$$f(u_0,\theta) \;\equiv Q\circ L_D\circ\ldots\circ L_2\circ L_1\circ P(u_0) \;\approx\; u_T = F(u_0),$$

With the operation at each level given by

$$L_n(v)(x,\theta) = \sigma(W_n\, v(x)+b + K_n(v))$$

Here at each level W_n is a pointwise linear local map, $K_n(v)$ is a global convolutional integral operator, with Kernal $G_n(\theta)$, and Q and P are projection operators.

As in the case of the linear heat equation, the global operator $K_n(v)$ is evaluated using an FFT via the operation:

$$FFT(K_n(v)) = FFT(G_n)\ FFT(v).$$

In the operation of the FNO the FFT is restricted to M modes. Nonlinearity and higher order modes are introduced via the nonlinear activation function σ, which is usually taken to be ReLU.

The FNO is widely used, and well supported on GitHub with code, test problems, and training documentation. See https://github.com/neuraloperator/neuraloperator for example.

4.4 Convergence

As with PINNS there has been a lot of recent work studying the convergence of neural operator methods. The picture is similar, with the caveat that neural operator methods are attempting to approximate an infinite dimensional operator, which is always going to be problematic. Given a sufficiently rich neural network, it is possible to find a neural operator which accurately approximate the operator on a (compact) set containing the training data. As with the earlier discussion on ML methods, this result only applies if the training procedure has worked well enough to find the optimal solution. The effectiveness of the neural operator also depends heavily on the richness of the training set (which has to span a subset of an infinite dimensional set). In the original FNO paper the training set is taken to be a set of random functions. However, this is not often a good choice. Generating a good training set is crucial and can be slow. It is key to the success of the method that the training set should be as representative as possible of the required solutions. FNO struggles away from the training set in general. However an important application of FNO methods is for MCMC emulators used in Uncertainty Quantification where the input will always be close to the training set. For this they are the right tool for the job!

More generally more work has to be done to broaden its scope and extend the theorems on its convergence. For example, we observe poor learning of conservation laws at the moment. Also none of this theory applies, for example, to the nonlinear heat equation

$$u_t = u_{xx} + u^2, \qquad t = 0,\ T = 1,$$

which has a singular solution if $||u_0||$ is sufficiently large. This solution will not be detected if the Neural Operator is only trained on regular data with a smaller norm.

4.5 Example: Weather forecasting using ML

An important application of Neural Operator, and other ML methods lies in weather forecasting.

In 2019 a meeting was held at Oxford to discuss whether AI systems could ever predict the weather 'on their own'. The conclusion at the time was that this was not possible. **Seven years later, the whole landscape has changed**, with AI methods now playing an increasingly important role in weather forecasting.

The conventional *physics based method* for weather forecasting comprises the following:

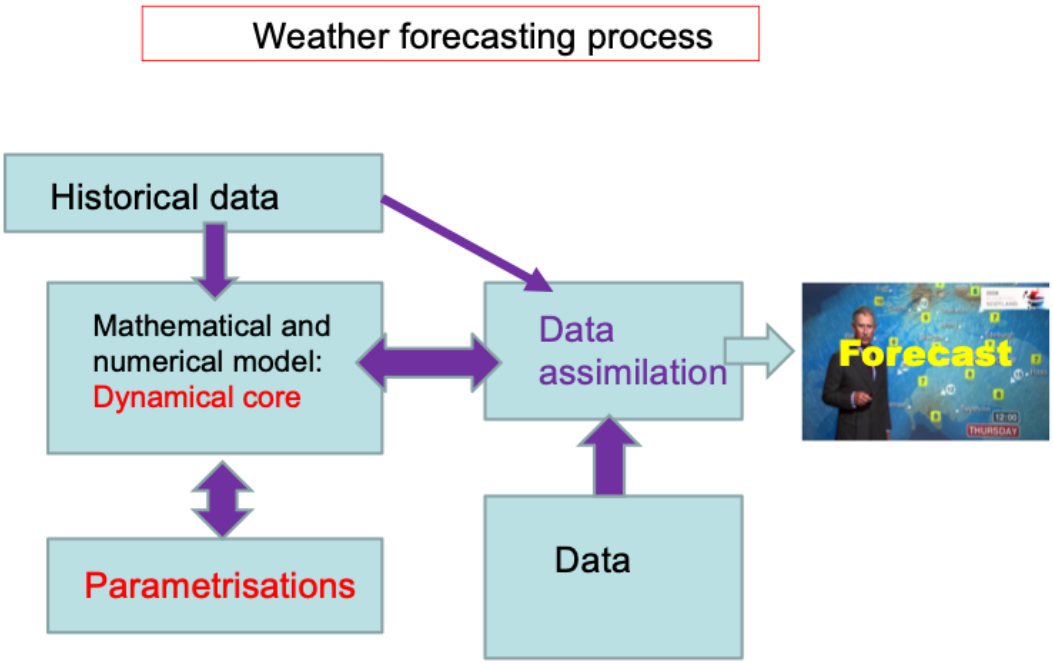
1. A set of PDEs describing the oceans and atmosphere etc. is constructed using physical reasoning based on historic data. This is called the 'dynamic core'. It is solved using a suitable numerical discretisation.

2. A set of routines (also based on historic data) which provide extra physical information to the dynamic core which is often at a smaller scale than the grid used to solve the PDEs. These are called the *parametrisations*. An example being the calculation of cloud cover.

3. An evolving set of data (say from satellites, or rain radar stations)

4. A *data assimilation* process which combines the forecasts made in 1 and 2 above with the data in 3.

This procedure is used by the Met Office and other weather centres to produce their forecasts. An example being the HIRES forecast from ECMWF, the forecast of which over the last 40 years are archived in the ERA5 data set.



Scientific ML methods are now being used to augment and replace this process in a number of ways.

Firstly generative AI is used to increase the resolution of a forecast (often by including additional local knowledge) through the procedure of *downscaling*. By this means we can get more accurate local forecasts of wind and rainfall.

Secondly, the parametrisations in step 2 above (which are usually empirical models) can be replaced naturally using AI models trained on data. An example of this is cloud forecasting using the Met Office CARAMEL project https://weather.metoffice.gov.uk/learn-about/weather/types-of-weather/clouds.

Thirdly the dynamical core including the parametrisations, can be replaced by using a Neural Operator method. This method can trained on (say) the ERA5 data above, so that given (say) a good description of today and yesterday's weather, it can forecast the weather for tomorrow. This procedure is used in the DeepMind *GraphCast* model by Lam et. al. [19] and the Met Office *FastNet* model. This procedure is very fast when trained but suffers from two disadvantages. Firstly it relies on having an accurate description of the weather today. This can be hard to obtain as it relies on learning the correct atmospheric state from limited data. Secondly the forecasts from these models tend to be good for 'low resolution data' such as pressure, and poor for high resolution data such as rainfall. However, the models have shown notable accuracy (skill) in predicting the weather for 10 days in advance and are better than (say) HIRES for doing this (when measured in a suitable error norm). One application of such methods is to make a forecast using the

physics based approach to get the higher resolution data, and then, after (say) five days to ‘nudge’ it towards the prediction of the AI based method.

Fourthly It may be possible to replace *all* of the steps 1—4 so that the AI method predicts the weather purely from the satellite data. This is the claim of the recent AARDVARK model from Cambridge and the ATI , see Allen et. al. [20].

This is a subject of rapid evolution. Watch this space.

5. Scientific Machine Learning and dynamics.

In sections 3 and 4 we saw how ML methods can be used to find approximations to the solutions of differential equations. In this section we will look firstly at how differential equations can be used to simulate a ML architecture, and secondly how we can use ML to discover, and reproduce, the dynamics of a system from experimental data.

5.1 Simulating a ML architecture: Neural ODEs I

Neural ODES were introduced by Chen et. al. [21] and in their definition are differential equations that reproduced the 'dynamics' of a ML architecture. One of the problems with the conventional architecture is that of 'vanishing derivatives' where information (say on the effect of a change in the parameters) is lost (exponentially) from one layer to the next. This problem can be overcome in part by using a 'ResNet' architecture of the form:

$$x_{n+1} = x_n + \Delta t \sum_i c_i^n \sigma(A_i^n x^n + b_i^n)$$

With input x_0 and output x_L .

Now, if we assume that Δt is small, and the L is large, then we can see that this is the Forward Euler discretisation of the *(Neural) ODE*:

(*)
$$\frac{dx}{dt} = \sum_i c_i(t)\ \sigma(A_i(t)x^n + b_i(t))$$

with input x(0) and output x(T) where $T = L\ \Delta t$. The ODE can then be solved by a numerical method such as RK4 etc.

The ODE (*) can then be trained to find an appropriate value of the output x(T) given the input x(0). To train such a Neural ODE we need to find suitable functions A(t), b(t), c(t). This is a well understood problem in *optimal control theory* and can be solved using adjoint methods. An application of this is to use such methods to forecast the weather by replacing a Fourier Neural Operator by a Neural ODE. There are certain advantages in memory efficiency and back propagation for doing this, instead of using a deep neural network. However Neural ODEs can have problems separating tightly coupled data.

5.2 Reconstructing dynamics: Neural ODES II

Suppose that we have some physical system, with an output/state x_i at the time t_i. Now suppose that we record measurements y_i given by

$$y_i = H(x_i) + e_i$$

where e_i is a noisy error, and H is an operator (for example a projection) which relates to the way that we record the data (for example we may be only able to measure the position of a moving particle, not its velocity. A key question in science is then: can we find the process that gives us these measurements, and then can we use this process to predict the future behaviour of the system.

This question is as old as science (or indeed humanity) itself as we try to make sense of the world around us, given by the information provided to us by our senses, and other measurements. A very early example of this is the Ptolemaic model of the solar system This took a model in which the Earth was at the centre, and the planets, and Sun, orbited on a series of small circles (epicycles) moving on larger circles centred on the Earth. Each such planetary orbit was then described by the size of the circles, the speed of rotation around the circle, and phase (6 parameters). Hence the whole solar system (of 7 heavenly bodies) could be represented by a model with 42 trainable parameters. This could be achieved with the techniques of the time. The Ptolemaic model, whilst rather inelegant (this was part of its problem) worked and was highly predictive for its time. However, it had to be abandoned when new data (from telescopes) came along of, for example the moons of Jupiter or the phases of Venus. This is classical example of a model failing to predict away from its training set. It is also a cautionary tale for all machine learning. Despite giving (for a time) the right answers, the Ptolemaic system got the physics completely wrong, and gave no understanding, unlike Kepler's laws (also derived from data), or Newton's law of gravitation (an analytical model, the results of which agreed with Kepler's laws), of what was really going on in the physical world.

Suppose that we have a system with measured data, which we think is driven by a differential equation (for example the Solar system) of the form

$$\frac{dx}{dt} = f(x), \quad x \in R^n, \quad x(0) = x_{0,} \quad y_i = H\big(x(t_i)\big) + e_i\,, \quad i = 0\,..N \quad (*)$$

We can attempt to model this differential equation using (what is also, but confusingly called) a Neural ODE of the form:

$$\frac{dz}{dt} = F(z;\theta), \qquad z(0) = z_0 \qquad (**)$$

Where $F(z;\theta)$ is a MLP or similar. The system (**) can then be solved using a standard numerical method such as RK4 and the point values $z(t_i) = z_i$ evaluated.

Given the set of data points as above, we can then train the Neural ODE (**) using the loss function

$$L = \frac{1}{N} \sum_{i=0}^{N} | y_i - H(z_i)|^2$$

or similar.

As an example to demonstrate both the strengths and the weaknesses of this method we consider an example modified from the Matlab website on Neural ODES [22].

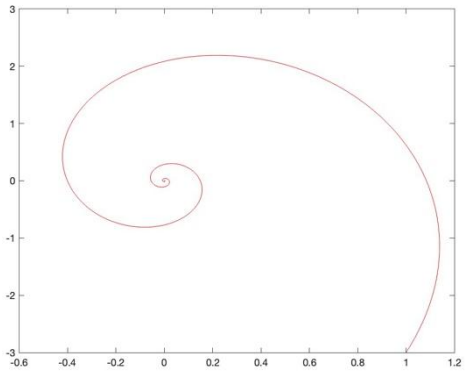
Let x satisfy the linear ODE

$$\frac{dx}{dt} = A\,x, \quad x \in R^2, \quad x(0) = x_0 \quad (\#)$$

with

$$A = \begin{pmatrix} -0.1 & -0.1 \\ 1 & -0.1 \end{pmatrix}$$

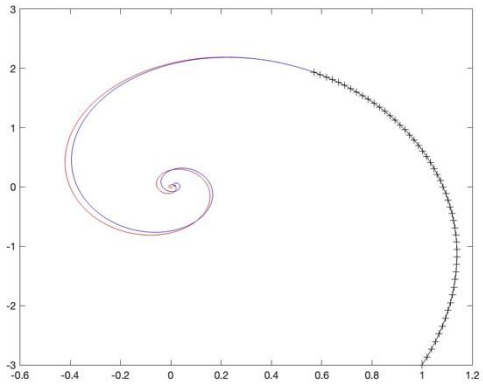
The linear differential equation (#) has solutions which spiral into the origin for all initial data. An orbit for which x(0) = [1,-3] is given below, computed using RK4 over the interval $t \in [0,500]$ with 5000 data points taken at time intervals of $\Delta t = 0.1$.



As a simple Neural Network we take a two-layer network with tanh activation given by:

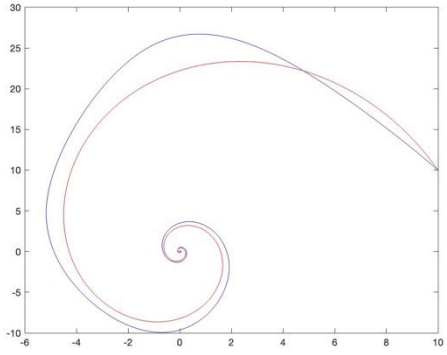
$$F(z) = W \tanh(M\,z + b) + c$$

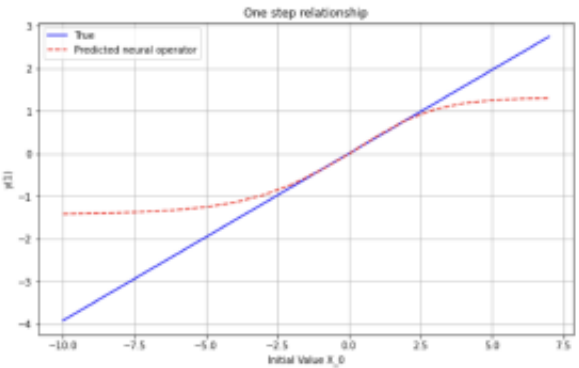
Here we need to find the matrices W,M and the vectors b,c. This network is then trained by solving the neural ODE numerically and using the first 50 data points. If this is done then the trained neural net finds the rest of the orbit with reasonable ease as seen below, in which red is the original orbit, blue the computed orbit, and the crosses show the training points. Note the slight deviation as the trajectory extends:



Similar good performance is also found if we start the neural ode at an initial state *close* to the training orbit.

However if we start the orbit further away from the training data then the loss rapidly increases. An example of this is Illustrated below on the left where we take (10,10) as the initial state, and show the difference between the true (red) and predicted (blue) orbits. We are not able to use the neural ode for a reliable prediction in this case.





The reason for the degrading of the performance is simple. We are trying to approximate a linear function A x, with a nonlinear one (tanh). This is asking for trouble. Whilst tanh is approximately linear for part of its range (such as that close to the training values), it rapidly saturates. On the right of the figure we see a one-dimensional cross-section the approximation of Ax compared to that of the neural ode. The impact of the saturation of tanh function is plain. This is an example of stable system bias, where the neural network is biased to give results close to its original training data, but deviates away from this data.

There is a simple 'fix' to this 'problem'. Instead of using a full neural net, use a bit of physical knowledge and consider the simple linear differential equation

$$\frac{dz}{dt} = B\,z, \qquad z \in R^2, \quad z(0) = x_0 \quad (\#\#)$$

We now try to find the best possible linear operator to fit the data. If we take only 100 points this estimate is given by:

$$B = \begin{pmatrix} -0.1001 & -1.0005 \\ 1.0004 & -0.0993 \end{pmatrix}$$

We can see that this is very close to the true value of A given above. Unlike the neural ODE the dynamics of the system (##) will be close to that of (#) *for all* initial data.

SINDY

The SINDY (Sparse Identification of Nonlinear Dynamics) by Brunton et. al. [23] approach to identifying an underlying dynamical system extends the linear system method described above. In the above example we used a linear function for f(x). In SINDY there is a 'library' of functions $\Theta(x)$ (such as polynomials, exponentials, trigonometric functions etc.) drawn on to construct f(x) and a best linear combination of these (parametrised by Ξ) is used to give a 'sparse' approximation (hence the name). In particular we have:

$$\Theta(\mathbf{x}) = \left[\mathbf{1} \mid \mathbf{x} \mid \mathbf{x}^{P_2} \mid \mathbf{x}^{P_3} \mid \sin(\mathbf{x}) \mid \cos(\mathbf{x}) \mid \ldots\right]$$

and if we set

$$\Xi = \left[\xi_1\ \xi_2\ \cdots\ \xi_n\right]$$

then the approximating ODE becomes:

$$\frac{dz}{dt} = \Theta(z)\,\Xi$$

As in the case of the Neural ODE, this system is trained by finding the best set of parameters to minimise the loss function as above. An example of SINDy in operation is given in [23] for the Lorenz equations. Notably the Lorenz system has a quadratic right hand side, and also chaotic dynamics which explores a lot of the phase space. This leads to a rich training set, making the SINDy approach especially effective in this case. It is a paradox that chaotic systems are often much more predictable than regular ones.

In conclusion Neural ODEs are often effective for both escribing the flow through a Neural Network and also for earning dynamics from data. However, they can degrade in performance when used away from their training data, and they work best when their design informed by the physics of the problem that they are trying to solve.

Section 6: A case study in PINNS

6.1 Introduction

This study will give you an introduction to PINNs and to PyTorch. It will give you an example of the use of PINNS to solve simple BVP and IVP differential equations in one-dimension. It will also allow us to study questions of the convergence and stability of the PINNS solver as a function of the architecture of the neural net and the location of the collocation points.

6.2 Solving a linear BVP

1. Open a Colab notebook session (you will need a Google account to do this) or your favourite Python IDE. Check that it is working by asking a simple question such as what is 1 + 1.

2. Copy the code PINNs Tutorial.py in the Python Examples Git-Hub folder into the Colab notebook. See

   https://github.com/ChrisBudd123/Ai4Sci/blob/main/Python_examples/PINNs_Tutorial.py

3. The code uses a simple 3-layer PINN to solve the linear BVP

   $$-u''(x) = f(x), \quad u(-1) = \alpha, \, u(1) = \beta$$

   In the code $f(x)$, α, β are chosen such that we have the exact solution

   $$u(x) = \tanh(x)$$

   The PINN is trained to minimise the residual

   $$r(x) = \left(-u'' - f(x)\right)^2$$

   evaluated at, and summed over, a set of collocation points xi combined with the boundary conditions. The ADAM optimiser is used in training. It is then compared to the exact solution. Read through the code to make sure that you understand how it works.

4. Run to code as given to check that it gives a sensible answer.

5. Now modify the code so that our problem has the exact solution $u(x) = \tanh(ax)$ for varying values of a . Run this modified code to see how the PINN performs as the value of a is increased.

Hint 1: Make sure to change the y true variable to plot $\tanh(ax)$.
Hint 2: $\frac{d^2}{dx^2}(\tanh(ax)) = -2a^2\tanh(ax)\cosh^{-2}(ax)$

6. Experiment with changing the width of the PINN and the number of collocation points to see how the solution converges (or not) to the exact solution, and how this convergence rate depends on the value of a .

6.3 Solving a linear IVP

We will now use a PINN to solve the linear second order initial value problem

$$u'' + \omega^2 u = 0,\ x \in [0,1]\ \ u(0) = 0,\ \ u'(0) = 1$$

1. Modify the previous example to solve this IVP for $x \in [0,1]$. Compare your solution with the exact solution for the case of $\omega = 5$. Investigate how the convergence depends on ω the number of collocation points etc.

2. Using the PINN trained over the interval [0,1] as above, apply it to solve our IVP for $x \in [0,10]$. Comment on your answer.

3. If you wish you can experiment with the related nonlinear equation

$$u'' + \omega^2 \sin(u) = 0,\ u(0) = 0,\ u'(0) = v$$

for varying values of $v > 0$ and $\omega > 0$. Note that theoretically there is a change in the qualitative behaviour of the solution to this equation as v increases through $v = 2\omega$. Does the PINNs solution capture this?

6.4 Solving a nonlinear IVP

We will finally use a PINN to solve the nonlinear first order initial value problem

$$u' = u^2,\quad x \in [0,1],\quad u(0) = \gamma$$

1. Modify the previous example to solve this IVP for $x \in [0,1]$. Compare your solution with the exact solution for the case of $\gamma = \frac{1}{2}$.

2. Now set $\gamma = 2$. What is the exact solution in this case? What does the PINN give? Does this depend on the parameters of the PINN?

Coding solutions to these problems can be found on the same GitHub repository as the case study code.

https://github.com/ChrisBudd123/Ai4Sci/blob/main/Python_examples/PINNs_Tutorial.py

7. Useful References

There is no good book that covers the whole of scientific machine learning. Indeed the subject is developing so rapidly that at the moment any such book would be out of date in six months or less. These references attempt to give a state of the art, but still accessible, overview.

Section 1: Introduction

[1] Stephen Lynch *Python for scientific computing and artificial intelligence*, (2023), CRC Press

A gold mine of useful information, and code, for a wide variety of applications.

[2] Francois Chollet, *Deep learning with Python,* (2018), Manning

[3] Catherine Higham and Desmond Higham *Deep Learning: An introduction for applied mathematicians*, (2019) SIAM Review 61.

An excellent introduction for a mathematical audience.

[4] Elaine Murphy et. al., *Machine learning outperforms clinical experts in classification of hip fractures,* Nature, Scientific Reports, **12**, Article number: 2058 (2022).

shows how machine learning methods can be used reliably to make a medical diagnosis.

[5] Leon Bungert, Desmond Higham and Laura Thesing, *Adversarial robustness of artificial intelligence*, (2026), EJAM

This is a collection of articles on adversarial attack.

[6] Nicholas Carlini et al. *Extracting training data from large language models*. 30th USENIX Security Symposium (USENIX Security 21). 2021.

Section 2: Architectures

[7] Philipp Grohs, Gitta Kutyniok, *Mathematical aspects of deep learning,* (2022), Cambridge

Mathematically very solid account of the way that neural networks are effective in expressing solutions, and an error analysis of this.

[8] Iddo Drori, *The science of deep learning,* (2023), Cambridge.

This gives a comprehensive account of many of the architectures discussed.

[9] Joos Korstanje, *Advanced forecasting with Python: With state-of-the-art-models including LSTMs, Facebook's Prophet, and Amazon's DeepAR*. (2021), APress, New York.

[10] Ashish Vaswani et. al. , *Attention is all you need,* (2017), NEURIPS 2017.

A true classic introducing the transformer architecture!

[11] J ohannes Hertrich et. al. *Learning Regularization Functionals for Inverse Problems: A Comparative Study* (2026) https://arxiv.org/pdf/2510.01755

[12] Georgios Batzolis, Jan Stanczuk, Carola-Bibiane Schönlieb, Christian Etmann, *Conditional Image Generation with Score-Based Diffusion Models,* (2011) https://arxiv.org/abs/2111.13606

See also. https://en.wikipedia.org/wiki/Stable_Diffusion

Section 3: PINNs

[13] Philippe Ciarlet, The Finite Element Method for Elliptic Problems, (1978), North-Holland.

[14] M. Raissi, P. Perdikaris, G. Kaniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,* (2019), Journal of Computational Physics 378, 686-707

[15] George Cybenko, Approximation by superpositions of a sigmoidal function, (1989), Mathematics of Control, Signals, and Systems 2(4), 303–314.

[16] National Institute of Standards and Technology, *Digital Library of Mathematical Functions, §3.5(v): Gauss Quadrature*, (n.d.), https://dlmf.nist.gov/3.5#v (accessed 22 October 2026).

[17] Tim De Ryck and Siddhartha Mishra, *Numerical analysis of physics-informed neural networks and related models in physics-informed machine learning*, (2024), Acta Numerica, 33, 633–713. https://doi.org/10.1017/S0962492923000089.

[18] Sifan Wang, Xinling Yu and Paris Perdikaris, *When and why PINNs fail to train: A neural tangent kernel perspective*, (2020), arXiv preprint, arXiv:2007.14527.

[19] Weinan E and Bing Yu, *The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems,* (2017), https://arxiv.org/abs/1710.00211

Section 4: Neural Operators and their applications

[17] Lu Lu, Pengzhan Jin, George Em Karniadakis, *DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators, (2019),* https://arxiv.org/abs/1910.03193

[18] Zongyi Li et. al. *Fourier Neural Operator for Parametric Partial Differential Equations,* (2021), ICLR. https://arxiv.org/abs/2010.08895

[19] Remi Lam et. al. *Learning skillful medium-range global weather forecasting,* (2023), ScienceVol. 382, No. 6677

This paper introduces the GraphCast algorithm for weather forecasting

[20] Anna Allen et. al. *End-to-end data-driven weather prediction*, (2025), Nature 641, 1172–1179 .

https://www.nature.com/articles/s41586-025-08897-0

This paper introduces the Aardvark algorithm for end-to-end weather forecasting.

Section 5: Dynamics

[21] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud, *Neural Ordinary Differential Equations,* (2018), https://arxiv.org/abs/1806.07366

[22] Matlab tutorial on neural ODEs

https://www.mathworks.com/help/deeplearning/ug/dynamical-system-modeling-using-neural-ode.html

[23] Steven L. Brunton, Joshua L. Proctor, J. Nathan Kutz, *Discovering governing equations from data: Sparse identification of nonlinear dynamical systems*, (2015). https://arxiv.org/abs/1509.03580

This paper introduces the SINDy algorithm.

What has mathematics done for AI?

Computational Mathematics

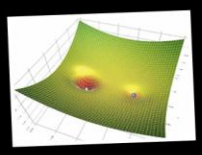
Work by 19th century mathematicians **Charles Babbage** and **Ada Lovelace**, and 20th century mathematicians **Alan Turing** and **John Von Neumann** leads to the invention of the modern computer







20th century developments in the **mathematics of optimisation** led to the rapid growth of machine learning



Statistics and Data Science

19th century mathematicians **Thomas Bayes** and **Carl Friedrich Gauss** transformed the way that we understand and manipulate data





$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$\theta^* = \underset{\theta\in\Theta}{\operatorname{argmin}} \left[\sum_{i=1}^{N} (f_\theta(x_i) - y_i)^2 + R(\theta) \right]$$

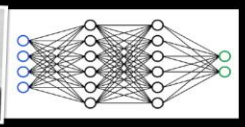
This mathematics is used in medical imaging



Algebra and Calculus

Matrices, invented by 19th century mathematician **Arthur Cayley**, provide the structure for the Internet and all AI architectures

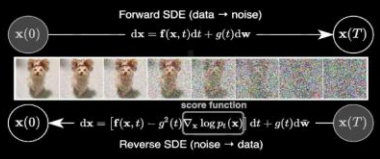




$$\boldsymbol{x}_{k+1} = \sigma\left(A_k \boldsymbol{x}_k + \boldsymbol{b}_k\right) \in \mathbb{R}^W$$



20th century mathematician **Kiyoshi Itô** developed stochastic calculus, the basis of generative AI



m4DL
www.maths4dl.ac.uk

About the authors

Chris Budd OBE, FAcadMathSci



Chris Budd OBE, FAcadMathSci, is Professor of Applied Mathematics at the University of Bath and Director of Knowledge Exchange at the Bath Institute for Mathematical Innovation (IMI) which has a primary mission of taking mathematics out to the broader community. He has been involved in knowledge exchange all of his career, both in academia and industry. He has particular interests in mathematical, scientific computing, and machine learning, and works closely with many industries, especially the environmental, electronic, power generating, and food industries. He has been a strong supporter of study groups all over the world, attending his first in 1984. He was a founder member of V-KEMS which pioneered the use of virtual study groups (VSGs) as a very effective way of bringing mathematicians together to work on the challenges posed by COVID-19. He is a Super Champion of the KE Hub, and a Fellow of the Academy of the Mathematical Sciences

Chris Budd is the director of the UKRI programme *Maths For Deep Learning (Maths4DL)* https://maths4dl.ac.uk/ which is a joint research project between Bath, Cambridge, and UCL, developing the theory, methods, and applications of scientific machine learning methods. The aim of this programme is to make scientific machine learning methods more reliable, accurate, and trustworthy, so that they can become reliable tools for industrial applications.

A passionate advocate for outreach in mathematics, he is also Professor of Mathematics at the Royal Institution and a fellow of Gresham College.

Aengus Roberts



Aengus Roberts is a mathematics PhD student at the University of Bath through the SAMBa CDT. He completed his Masters in Mathematics and Physics in 2024, also at the University of Bath where he researched using machine learning to learn dynamical systems.

He now spends his PhD untangling and demystifying machine learning for use within scientific computing with a special interest in solving differential equations using neural networks (see chapters 3 and 4). When not debugging his code, Aengus spends his time baking overly technical bakes and teaching Scouts how not to get lost wandering the Bath countryside.