Ai4Sci

Scientific Machine Learning

AI for mathematics, and mathematics for AI

Chris Budd OBE, FAcadMathSci mascjb@bath.ac.uk

and

Aengus Roberts asr66@bath.ac.uk

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

April 2026

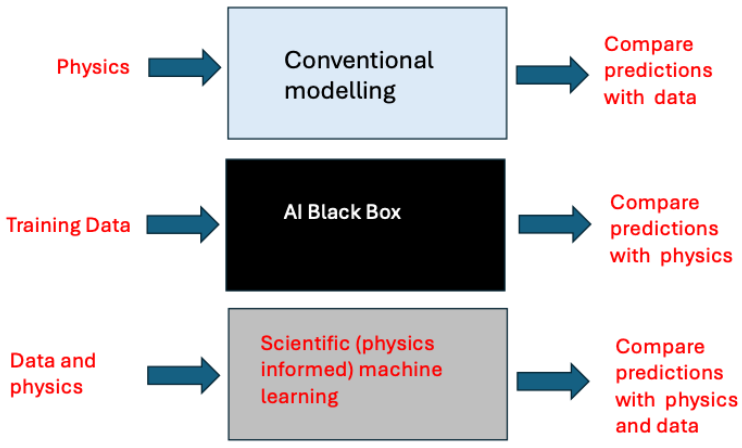
0. Contents

1. Introduction

1.1 Overview

AI, and in particular 'scientific' machine learning (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, or constructing a (dynamical) model to fit some data. All of these areas have the potential to greatly benefit science, and engineering, and all aspects of BIG.

Some reasons for this are (i) that Ai methods (based usually on neural network architectures) are very *expressive* so that complex systems can be represented with (relatively) few, trainable, parameters, (ii) they make very effective use of data, (iii) good software exists to set them up and train them (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 black boxes, where it is hard, or impossible, to see what is going on inside them. It is also difficult (unlike say finite element methods) to get reliable error estimates for their use, they can hallucinate (produce very wrong answers), and the output of such a method can be unstable to small perturbations. We can compare the use of such black box methods with more 'conventional' 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 use, and often does not make effective use of data. Scientific Machine Learning (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.



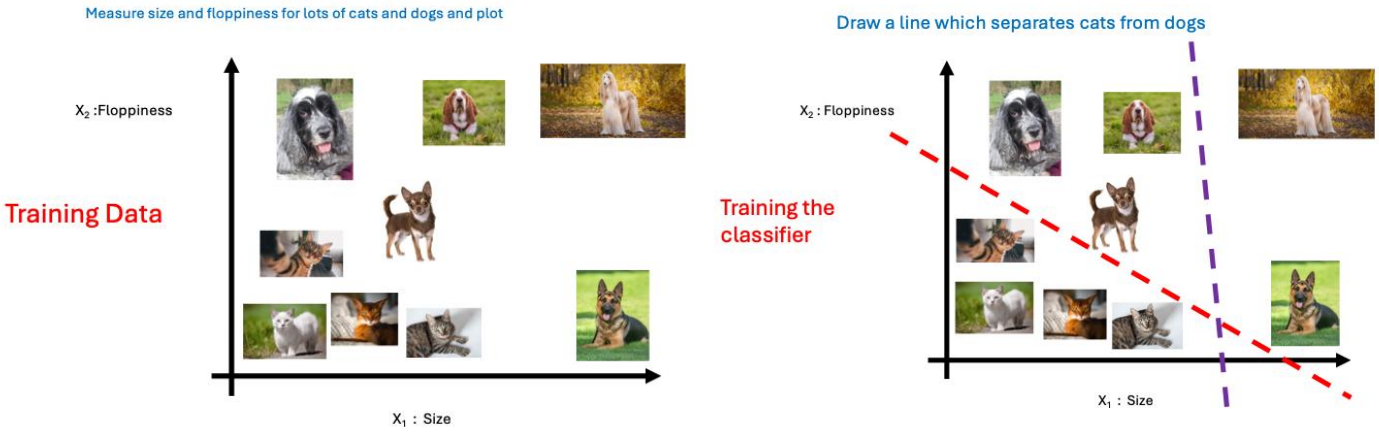
Good accounts of ML in general, and the use of Python to implement ML algorithms can be found in Lynch [1], Chollet [2], and an excellent account for mathematicians in Higham & Higham [3] .

1.2 A very simple example of scientific ML 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. However, as the picture below shows, some cats are bigger than some dogs.

shutterstock.com · 1046217481

Because of this we need to introduce another measurement, we will call this the 'floppiness' of the image. We can then take a large 'training' set of classified images of cats and dogs and plot them according to size and floppiness. The result might look something like this on the left.



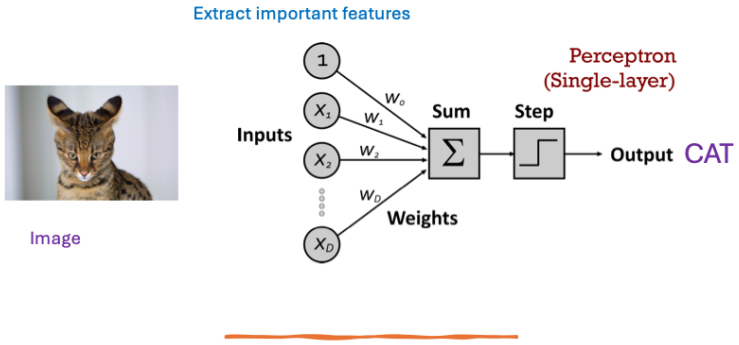
We can now try to separate cats from dogs by drawing a line through the training data so that any image above it is a dog, and every image below is a dog. We have done this on the right. The blue dashed line does a poor job at separating the data, whereas the red

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.

Mathematically, 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

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

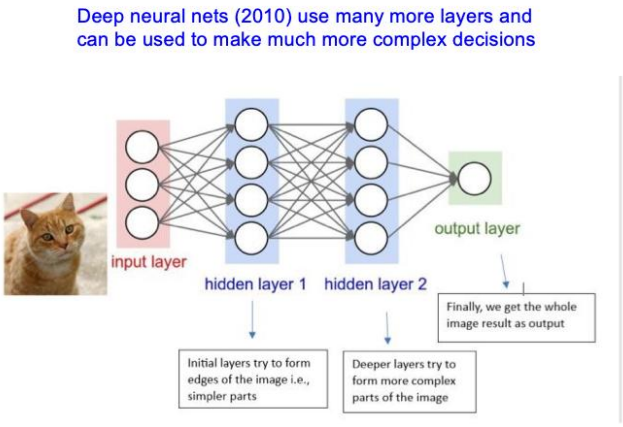
Images above the line 'dogs' have $w_1 x_1 + w_2 x_2 + w_0 > 0$ and images below 'cats' have $w_1 x_1 + w_2 x_2 + w_0 < 0$. The coefficients w_1,w_2, and w_0 have to be 'trained' so that these inequalities hold for each of the labelled training images. Once we have done this, then for any other image we can tell whether it is a cat or a dog by evaluating this expression. This is the action of the 'perceptron' which was introduced in 1963. More generally, if we introduce the 'activation function' $\sigma(z)$ which in this case is the Heavyside (or step) function which satisfies the condition $\sigma(z) = 1$ if z > 0 and $\sigma(z) = 0$ if z < 0, then 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. 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.

By increasing the width D we can increase the ability of the neural net to classify images. However a much more efficient was to do this is to use a *deep neural network* where the output of one layer is fed into another layer as illustrated below



and each layer has its own set of weights and biases. The number L of layers is called the depth of the network. The deeper the network, the more expressive it is, but also the harder it is to train. The 'activation function' σ 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 during training, and are often chosen by experiment. The overall structure is called the 'architecture' of the neural network. The above is a fully connected architecture, and we will see presently how different architectures are used in other applications. The set θ of all of the weights and biases are the free parameters of the network, which are determined by training

Using a deep network it is possible to classify many more types of image, or indeed any other form of data. An important example of this is the use of such architectures by doctors to diagnose different types of clinical condition given a medical image such as an X-Ray, see Budd et. al.

1.3 Training a ML architecture

Training a ML system is the process of finding the parameters θ so that it does a pre assigned task. Training 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 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 ML network requires the following ingredients.

(i) A training set x and a validation set y

(ii) A loss function L(x;θ) which you want to be close to zero on the training/validation set if the network is performing correctly (for example is it correctly identifying a dog as a dog)

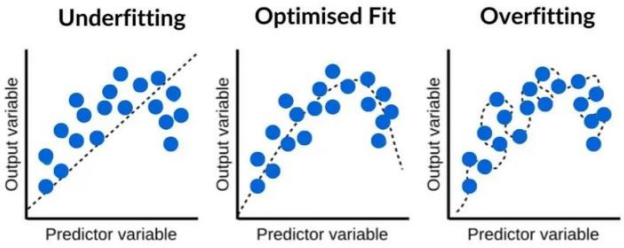
(iii) A measure of how L changes as you change θ. This is called back-propagation and is achieved in (say) PyTorch by automatic differentiation

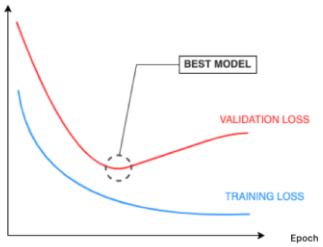
(iv) An optimisation procedure, together with a learning rate (which is generally found by experiment), which will determine the optimal parameters θ^* which minimise the loss function L over all elements of the training set x

(v) A validation step in which the loss function is evaluated over y.

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

Training is not easy, for a number of reasons. Firstly, the loss function 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 ML procedures, where the network does not perform as required. The more information we can provide about the problem (for example by building in the physics) the better we can construct the loss function. 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 over fit to the training data x, so that we fit to the noise in the data rather than to the signal. This is illustrated below on the left.





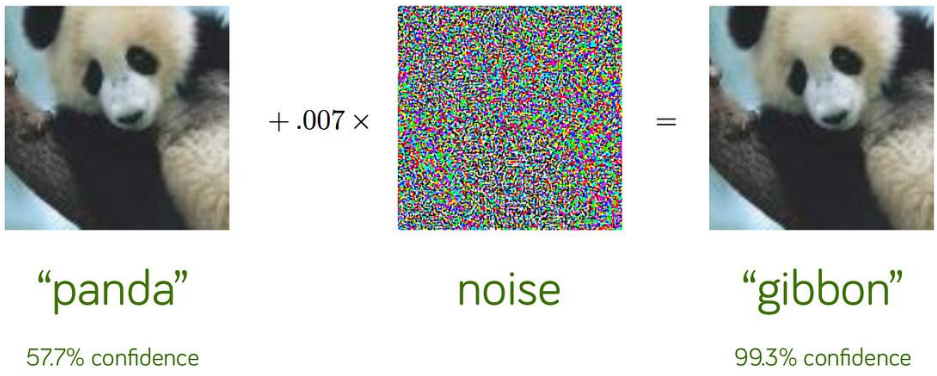
The impact of over fitting is reduced by using the validation set y which should be chosen to be significantly different from the training set x. As the optimiser works through the process of minimising L (by looking at epochs of typically random samples of the training data) the loss on the validation set is also monitored. When this starts to rise, as illustrated above on the right, then the network is over fitting to the training set, and the optimisation process is terminated at the best model (illustrated)

1.4 Some advantages and disadvantages of this approach.

Once trained a ML network can be very effective as an image classifier, or for many other tasks. As we will see they are often very good for many forms of image classification, forecasting the weather, and solving certain differential equations. However there are certainly problems associated with this approach.

Bias: One is that the training can be influenced by bias in the data set. An example is shown above of an image of a stop sign that was mis identified as a 45mph sign after training. The reason being that the training images for 45mph signs all had blue backgrounds, so 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 ML systems for grading job applicants, 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.

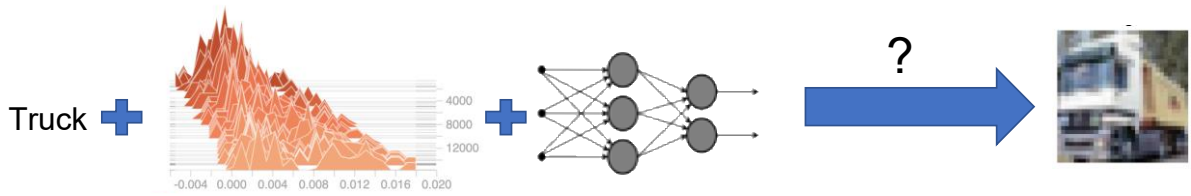
Adversarial Attack: When the ML architecture is trained to recognise an image, it is acting as a black box, and it is often not clear how it makes its decisions. One impact of this is it can be subject to 'adversarial attack'. In this a small, but carefully selected, perturbation can be made to the image so that it is incorrectly classified. (This is an ill-conditioning problem in numerical analysis). As an example, we can take a picture identified as a panda, add a small amount of noise (specifically designed to 'trick' the ML architecture) and get a picture then identified as a gibbon.



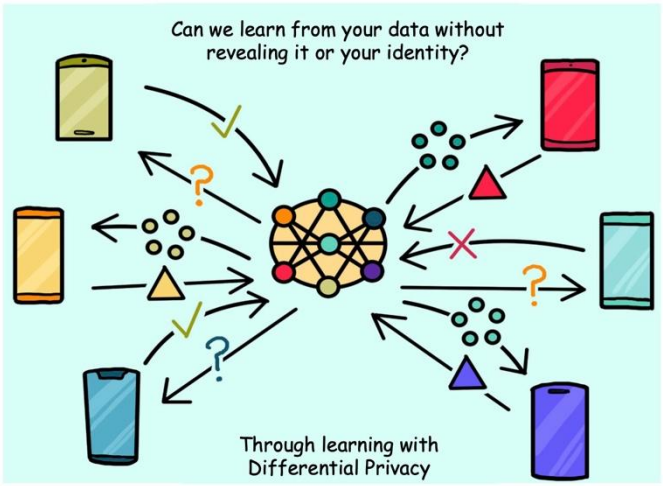
It is important that image classifiers should not be vulnerable to such form of attack, as this greatly degrades their ability to make reliable decisions. See Higham et. al [4] for a very good account of this problem.

Moving away from the training data (stable system bias): We will see presently, that ML methods (specifically Neural Operators) can be very effective in forecasting the weather. *But* .. can a weather forecast trained on Californian data predict a storm in the UK? The answer is no! And this gets to the heart of ML methods. Essentially 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 the following sections.

Data security or not: Image classification models can memorise their training images through the training updates (i.e. the gradients used in the optimisation process).



This can cause a problem if the training data is sensitive, for example images from patient records, or data from a census. In principle it may be able to reverse engineer a ML architecture to see what it has been trained on. The methods of *differential privacy* (effectively producing synthetic data to train the ML method on based on the true data have been developed to keep your training data secure, see [5].

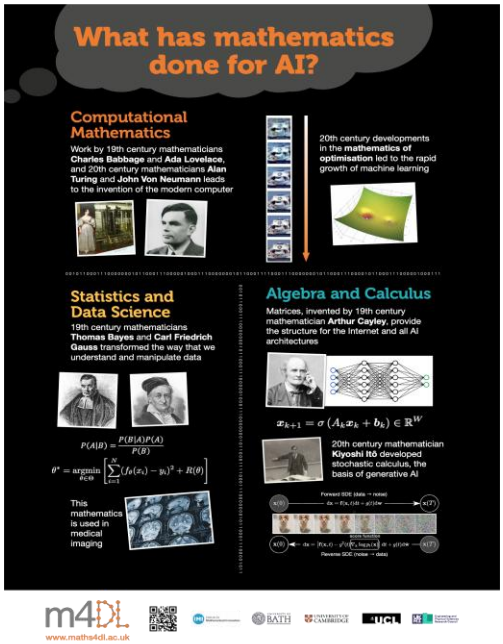


In contrast we might want to be able to detect whether a ML architecture has been trained on illegal data, such as pornographic images, or images with copyright. At the moment this is possible, but difficult, using machine learning forensics.

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. For a much more comprehensive account see Grohs [6] and Drori [7].

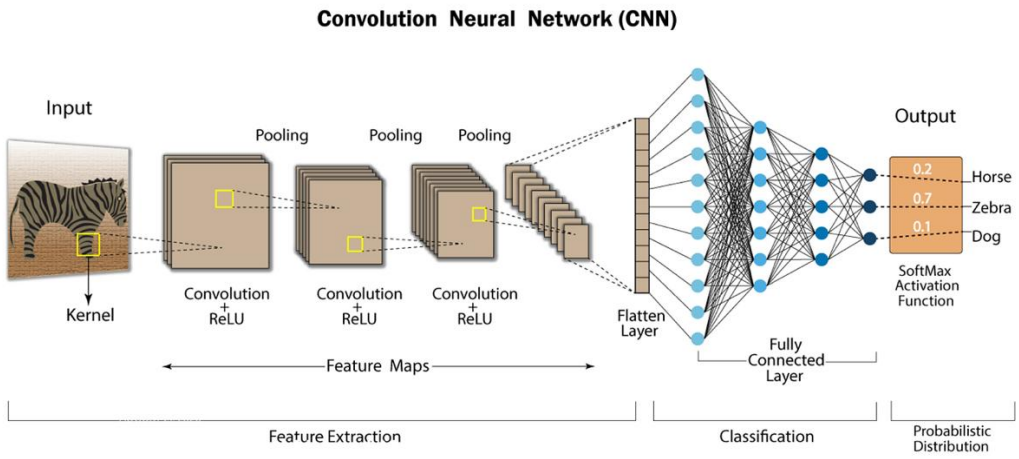
2.1 The MLP

The basic architecture described in the Introduction is usually called a (fully connected) multi-layer perceptron or MLP. Such architectures are general purpose and 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 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 aproximation of functions (especially in higher dimensions), the solution of differential equations using PINNS or variational methods (see Section 3). Key to the success of the MLP is its 'expressivity', which is roughly its ability to represent a function in terms of its trainable parameters. A lot of work has gone into studying this problem, see Gutyniok [6], and we have a good understanding of it. Roughly speaking, if an MLP has width D and depth L and we use a continuous activation function such as ReLU, then it is (in theory) possible to find a set of parameters θ so that the error in approximating a function f 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 at comparatively little extra cost. The catch behind this, is that training the MLP architecture to achieve this optimal result may be very difficult, if not impossible. This is why different (more trainable) architectures have been designed to deal with specific tasks. See [6] for more details.

2.2 The Convolutional Neural Network (CNN)

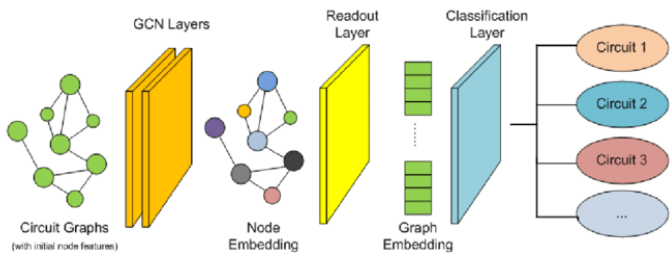
The CNN is a neural network designed for image processing. Conventional image processing relies on applying linear filters to an image to extract features such as edges, and the reduce the effects of noise. If the image is thought of as being a high dimensional vector **x** (with the terms of each such 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 A is a convolution operator with a 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. 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. The later layers can then have an MLP form to learn information from these features. (We saw this in action in the cat and dog example in the introduction). There are two advantages to this structure over an MLP. Firstly, it uses tried and trusted image processing technology which is well understood and predictable. Secondly 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)

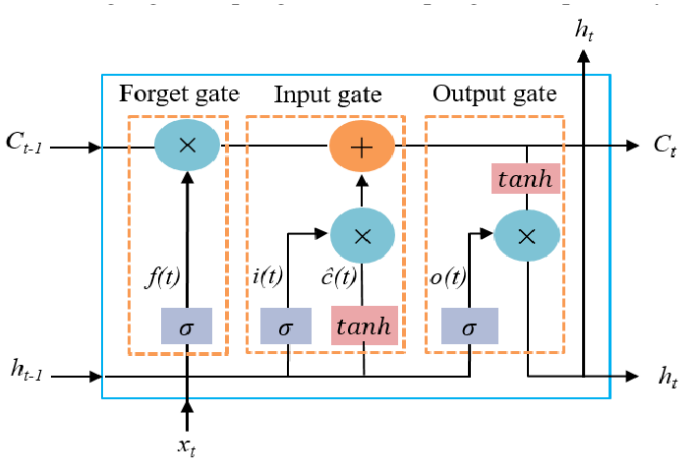
The CNN is effective because it treats nearby pixels as being related. In a GNN architecture this observation is taken further to look at data with terms related through a the edges of a graph, of which they are nodes.



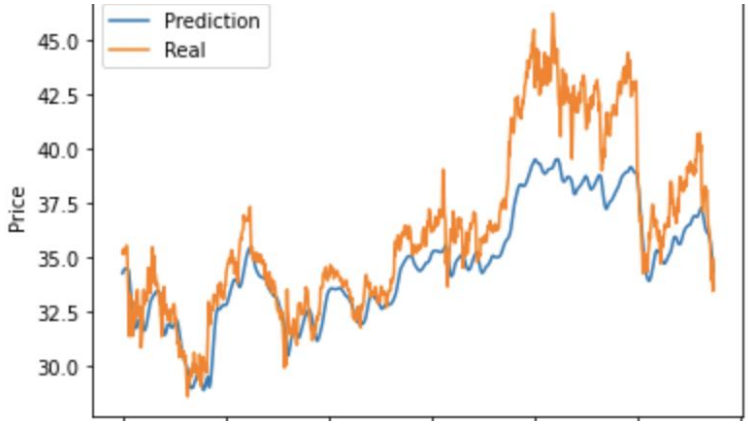
As well as in image processing they are used across various other fields, including: Molecular drug design: Predicting the efficacy of molecules based on their atomic structure. Social network analysis: Identifying communities and influential users. Physics and engineering: Modelling unstructured grid data and optimizing mesh relocation in finite element methods (FEMs). Healthcare: Identifying progression pathways for diseases like Alzheimer's from patient record.

2.4 The LSTM and recurrent neural networks

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 speech, and the need to use technology for speech recognition. Given any such time series and important question is then, given such a sequence for i=1 .. N, can we *predict* future values of x_i for i = N+1 .. N+m. Such questions are of importance in, for example in the power generation and the financial industries. 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 tools to bear on this problem which exploit the sequential nature of the input data. The primary tool for doing this is the RNN or 'recurrent neural network'. These are a type of NN which take information from previous inputs which then influences the current input and output, so they feed back into themselves with an internal memory. They are used for prediction, translation, composition, speech recognition, and natural language processing. An RNN processes input sequences (x_i) via hidden units (h_i) to form outputs (y_i) by 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 through the network



The LSTM (Long and short term memory) architecture (see above) was introduced by Hochreiter and Schmidhuber in 1997 to address the problem of infomation fading. Currently LSTMs are proving very effective in prediction, and are used, for example, by the power industry and by Uber. For more details of how these are used see Lynch [1] and Korstanje [8]. Below is an example of using an LSTM to predict the price of a stock.



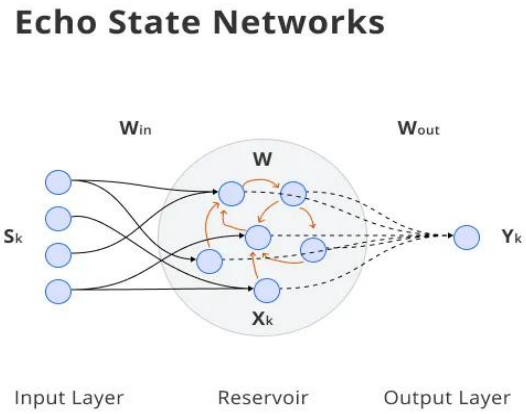
2.5 Other architectures

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. See Drori [7] for a comprehensive account of all of these. The most recent 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' Vaswami et al [9], and stacks of encoders passing information on to each other, usually performing computations in parallel. These have state of the art performance for many tasks, starting with natural language processing (hence the name LLM or large language models), but extending to computer vision and audio processing as well as many other tasks.

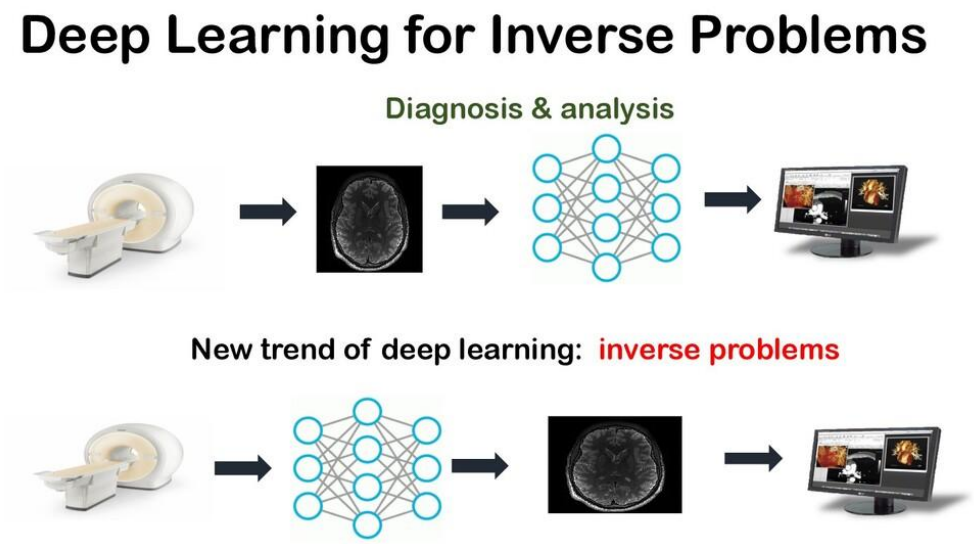
Echo State/Reservoir Networks

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 simple 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/predicting the dynamics of chaotic dynamical systems.



2.6 Neural network based methods for inverse problems

Inverse problems are an are in which scientific machine learning is making an important contribution. An example of a typical inverse problem is medical imaging such as MRI or a CAT scan, where measurements are made, with noise, 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 extreme sensitivity to the input data), and require some form of *regularisation* to work.



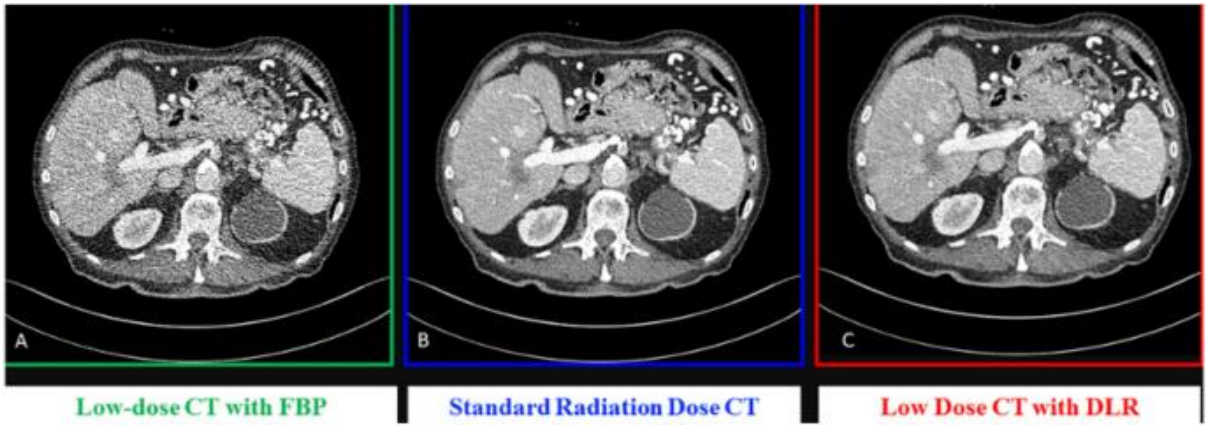
As an important example of this is the linear problem to find x given y where

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

And A is an ill-conditioned matrix. Thus can be ‘solved’ by finding x where

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

Here R(x) is the Regulariser and the choice of this is critical. Examples which do not use machine learning are (standard but of mixed effectiveness) Tikhonov regularisation where R(x) = $||x||^2$, and (the very effective but complex to apply) sparsity preserving Total Variation(TV) regularisation $R(x) = \| \ |\nabla u| \ \|$. However none of these regularisers take into account the nature of the solution. As an example a doctor looking at an image would use their experience (say that they are expecting to see an image of a knee) to select a good solution. This information can be included into a *learned regulariser* so that R(x) can be trained as before to select a good image. Learned regularisers are proving very effective not only in medical imaging but other areas such as art restoration. See Hertrich et al. [10] for a review of this methodology. The figure below shows three images constructed using different regularisers, with the results from the learned regulariser shown on the left.



2.7 Generative AI

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

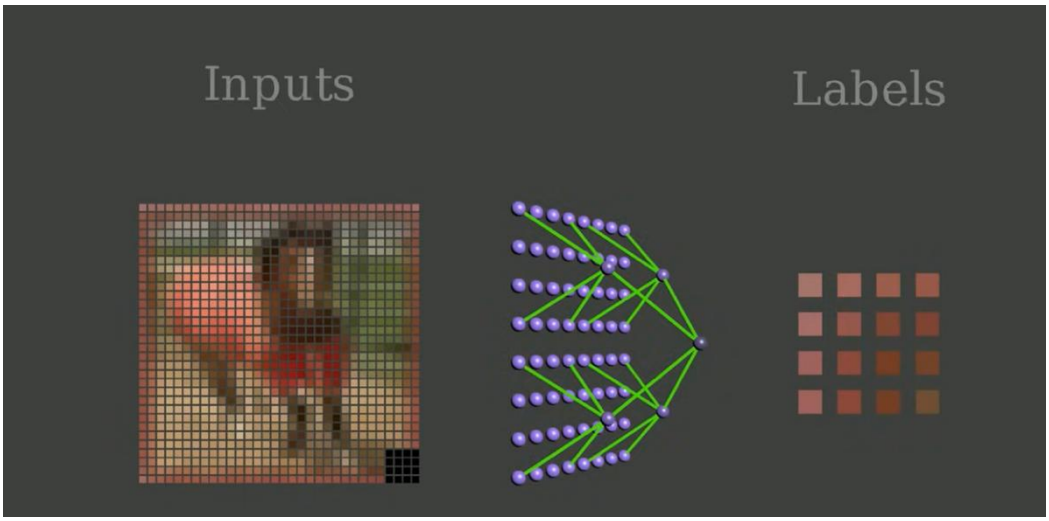
In the problem in the introduction, we showed that we could train an ML system to classify an image given a set of training labels and images. Generative AI performs the opposite task. Given a (large) set of images which are assumed to be samples of a large distribution of similar images, Generative AI attempts to find the image from the that distribution which most closely matches the given label. Of course we can do much more than that. We can, for example, ask a generative platform to (say) compose a poem about ML in the style of Brahms, or write an essay for us.

The original methods used by Generative AI use auto-regression to predict the next most likely part of a sequence, given a lot of training examples. Exactly 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 given *The quick brown fox jumped over the* .. Then we fill in with *Lazy Dog* etc.

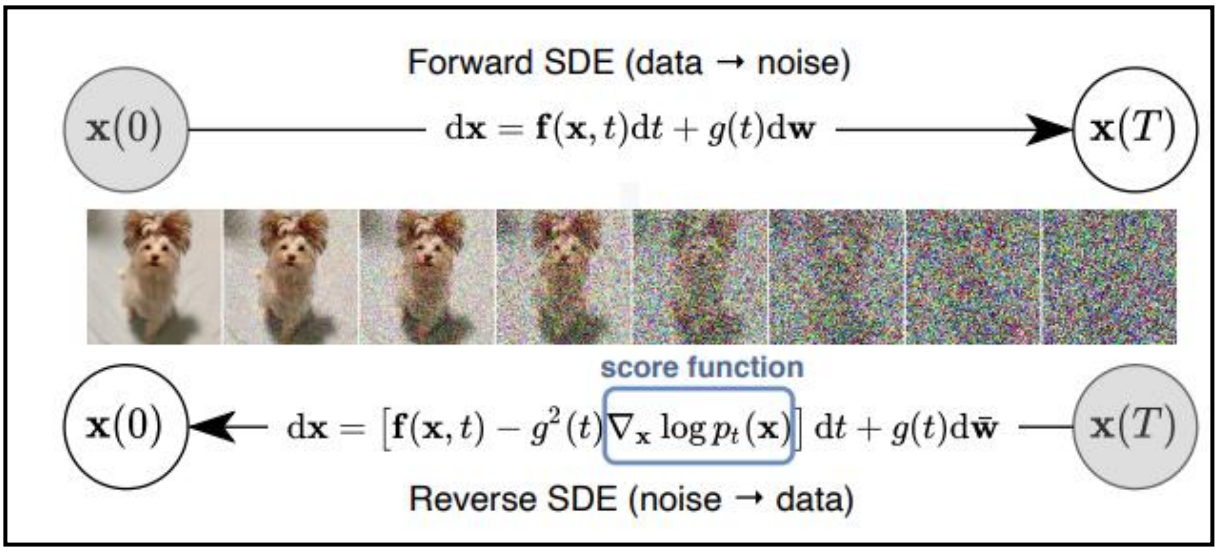
Similarly, given an image with some pixels missing, we can train an algorithm to fill in the next most likely pixels:



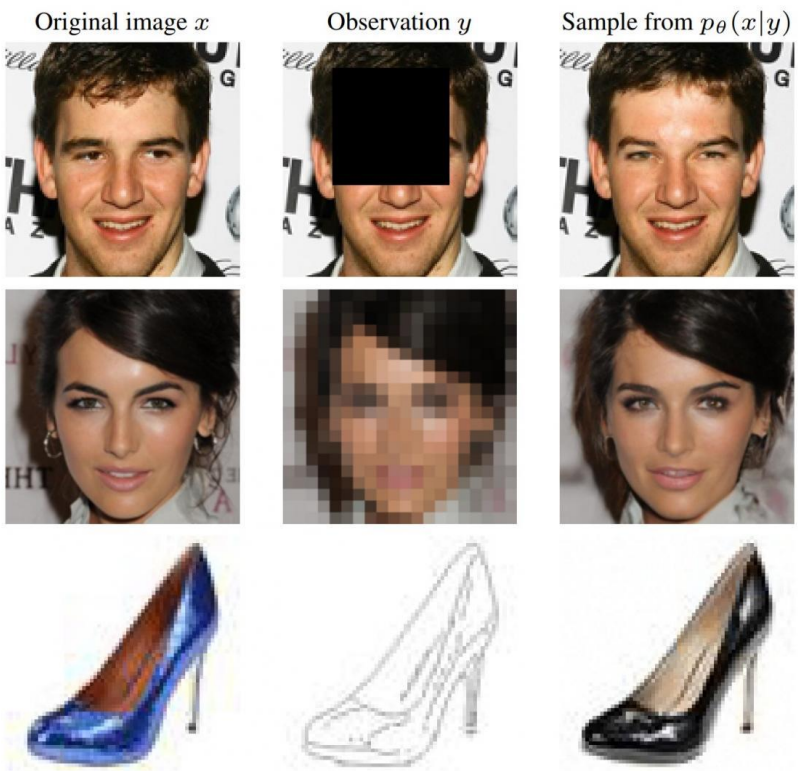
This can be repeated successively with more and more pixels removed, guided by the labels.

Auto-regressive methods give good answers, and are used in large language model such as Cat-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 is 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.



See Batzolis et. al. [11] for a full description of how this methodology works. Below are some examples of it working (taken from [11]) in which we see the original image, a distortion of that image, and then the use of diffusion based generative AI to recover the original image from the distortion. Similar diffusion based methods can be used to (for example) add in extra detail to a forecast of rainfall.



Stable diffusion methods are fast, and produce high quality photo-realistic images from random seeds as shown below (none of these are real people).

This 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, ODE's and PDE's 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 questions 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:

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

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

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

We first introduce the Finite Difference Method (FDM) which approximates the derivative as the difference between neighbouring grid points. By picking a finite number of points across our domain, $x_i, \ i \in [1, \dots, N]$, we can approximate the function at these points $u_i \approx u(x_i)$. This allows us to make approximations of derivates of the function such as

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

Applying this logic to the Heat equation given by $\frac{\partial u}{\partial t} = \Delta u$ we need to discretise in time and space which gives us the approximation $u_i^n = u(x_i, t_n)$ which in turn would allow us to discretise the whole problem as

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

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 of FDM can be bounded and depends on the spatial and temporal step sizes together with a problem-dependent constant

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

This can be improved (i.e. reducing $C(u)$ and increasing p and 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.

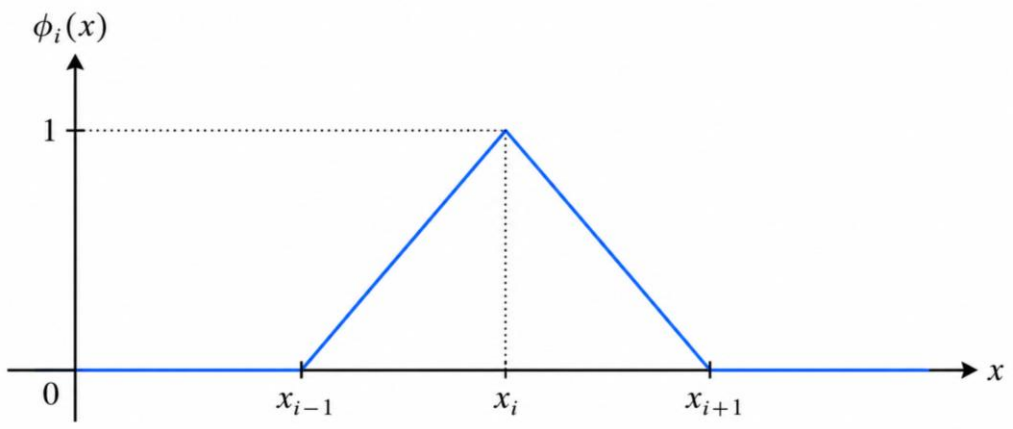
Extending to higher dimensions can however cause issues. As we have seen, FDM requires a structured grid - typically a uniformly spaced rectangular mesh. In higher dimensions this gives many points in the domain to solve for, and this can be further exacerbated by irregular or complicated geometries.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

3.1.2 The Finite Element Method

Our next method will 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. Finite element methods (FEM) aim to build a solution as a linear sum of simple basis function where each basis function acts over a subset of the domain and is zero elsewhere (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 these coefficients we consider an energy and aim to minimise it$F(u) = \int_\Omega f(u, \nabla u, x)dx$

We can 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 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 [12] 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

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

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

Our final classical method is the collocation method. As with the finite element method, 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 , $L(u) = 0$ we therefore impose

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

This generates a system of equations for the unknown coefficients $U_i(t)$. 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. 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 can often be bounded by

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

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 limited and dimensional problems.

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, as 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 new, machine learning (ML) based numerical solver proposed in 2019 [13]**.** 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 aim to solve differential equations of the form $Du = f$ where D is a differential operator acting on u and f is data function. If we consider the residual of this differential equation and take the norm of it

$$\big||Du - f|\big|$$

we observe that this is minimised when u is a solution to our original differential equation. Like with the classical methods we saw earlier we now want to approximate the solution to our differential equation with a different type of function. In the case of PINNs we will use a neural network, like we saw in the previous chapters, which we shall denote u_θ. The idea behind this is that there exists a multitude of theories (first and foremost the universal approximation theory [14]) which very roughly says that there exist neural networks that can well approximate any function we like.

To train our neural network we need a loss function which is where our differential equation residual comes in. Ideally we would like to plug our neural network into our

residual norm making use of automatic differentiation to evaluate the differential operator on our and try to minimise it:

$$\tilde{L}(\theta) = \arg\min_{\theta \in \Theta} \left| |Du_\theta - f| \right|$$

However, like other numerical methods we cannot evaluate our chosen function everywhere in the domain. Instead we evaluate the residual at a finite set of points within the domain and minimise the resulting loss 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. 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 a loss function that looks like

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

We can treat boundary and initial conditions in the same way by considering the residual of an appropriate operator in those subdomains to obtain a full loss function of the form

$$L(\theta) = \sum_{\{j=1}^{N} \left| Du_\theta(x_j) - f(x_j) \right|^2 + \beta \sum_{i=1}^{M} |Bu_\theta(x_i) - f(x_i)|^2$$

Where we have N sampled collocation points within our domain and M collocation points on the boundary. For a 1 dimensional problem this would simplify down to $M = 2$. 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.

3.2.2 Examples

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

Example 1: Two-Point Boundary Value Problem

Consider the regular two-point boundary value problem

$$-u_{xx} = f(x, u, u_x), \qquad u(0) = a, \qquad u(1) = b \qquad x \in [0,1].$$

In a PINN approach, we approximate the solution using a neural network $u_\theta(x)$, where θ denotes the trainable network parameters. 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 a loss function containing both the equation residual and the boundary errors:

$$L(\theta) = \frac{1}{N_r}\sum_{i=1}^{N_r} |r(X_i^r;\theta)|^2 + \frac{\beta}{2}(|u_\theta(0)-a|^2 + |u_\theta(1)-b|^2)$$

Where X_i^r are collocation points in (0,1). 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 PDE's with PINNs

Consider the semilinear parabolic PDE

$$u_t = u_{xx} + f(x,u,u_x), \qquad u(0,t) = a, \qquad u(1,t) = b \qquad u(x,0) = u_0(x) \qquad x \in [0,1]$$

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

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

$L(\theta) = \frac{1}{N_r}\sum_{i=1}^{N_r}|r(X_i^r;\theta)|^2 + \frac{1}{2}(|u_\theta(0)-a|^2 + |u_\theta(1)-b|^2)$,

where X_i^r 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 full PINN, where the neural network directly represents the solution as a function of both space and time, $u_\theta(x,t)$. In this case the residual

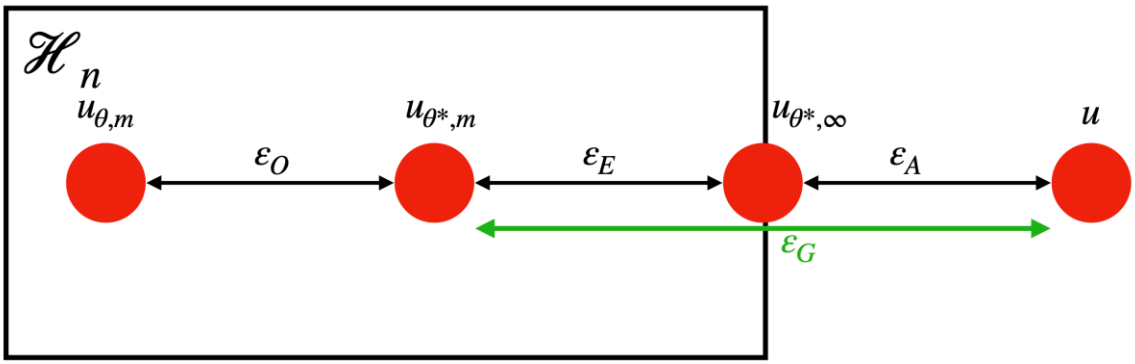
$$r(x,t) = \partial_t u_\theta - \partial_{xx}u_\theta - f(x, u_\theta, \partial_x u_\theta)$$

is evaluated at collocation points in the space–time domain and the loss function incorporates both the PDE residual and the initial and boundary conditions. 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 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 of functions H_n that can be represented by a neural network with n degrees of freedom. In practice, after training the network using a finite set of collocation points, we obtain an approximation $u_{\theta,m}$. However, this approximation is influenced by several limitations in the learning process.

First, the training procedure 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 perfect optimisation using the same finite set of collocation points. The difference between these two functions corresponds to the optimisation error.

Even if perfect optimisation were achieved, the use of a finite number of collocation points introduces another source of error. 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 arising from the finite sampling of collocation points.

Finally, the neural network itself may not be able to represent the true solution exactly within the chosen function class H_n. The difference between $u_{\theta^*,\infty}$ and the true solution u is known as the approximation error, which reflects the expressive power of the neural network. The total error that we observe in practice can therefore be decomposed as

$|u_{\theta,m} - u|$.

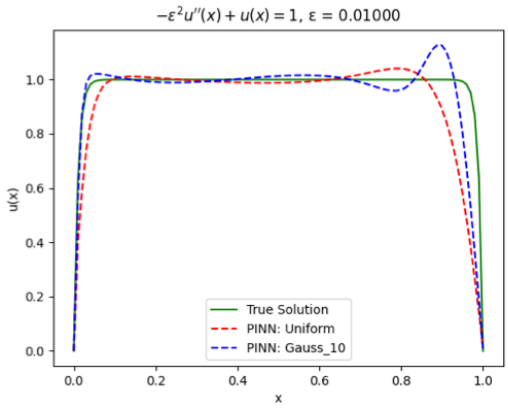
Theoretical work for PINNs has bounded both the approximation error and estimation error (often coupled together as the generalisation error) however, there does not exist a bound on the optimisation error which often dominates the total error observed in practice.

3.4 Why do PINNs sometimes fail?

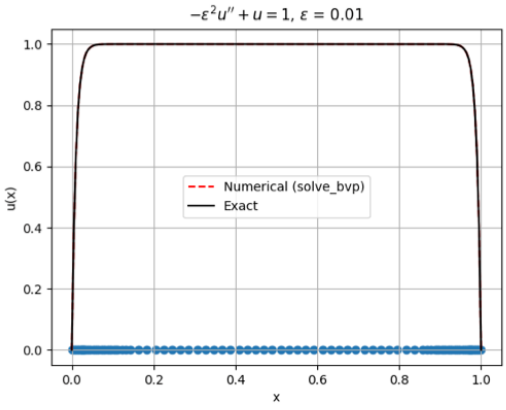
Although PINNs provide a flexible framework for solving differential equations, their practical performance can differ significantly from the theoretical picture outlined in the previous section. In particular, the *optimisation error* introduced during training can dominate the overall error and prevent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

To illustrate this, consider the singularly perturbed boundary value problem

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



PINN



Collocation

This problem exhibits boundary layers near x=0 and x=1 of width $O(\varepsilon)$. Below 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 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 can therefore dominate the overall error. Another issue we are seeing here is one of spectral bias, that is the higher frequency terms (parts of the solution that change more rapidly) are not learnt as well. This issue is not specific to PINNs and can occur with other numerical methods however a direct resolution of this problem for PINNs requires an extra bit of problem knowledge built into our solver.

Recent research has identified several factors that contribute to these training difficulties alongside like for like comparisons of PINNs against classical methods such as FEM. 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 landscape associated with PINNs is highly non-convex, meaning that optimisation algorithms may become trapped in poor local minima or flat regions of parameter space.

One cause of this non-convexity is due to weights and biases in the network performing vastly different roles as part of our approximate solution. Adding in the activation function to the network then creates a very non-linear relationship 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, and methods based on neural tangent kernels.

3.5 The Deep Ritz Method

We now introduce an alternative neural-network based approach for solving partial differential equations is the Deep Ritz Method (DRM), introduced by Weinan E and Bing Yu (2017) [15]. One benefit of the method is that we implicitly deal with spectral bias which was one of the issues we 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 Poisson equation $-\Delta u = f$ with suitable boundary conditions corresponds to minimising the functional

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

The Deep Ritz Method approximates the solution using a neural network $u_\theta(x)$ and seeks

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

Where the integrals are numerically approximated through a set of quadrature points.

A key motivation for this formulation is that the optimisation is often better behaved than the PINN residual loss. 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

Looking back to section 3.1.2 you may notice that the Deep Ritz Method is closely related to the 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, 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 compared with classical discretisation's, it inherits many of the optimisation difficulties associated 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 as a PINN or a DRM to find the pointwise solution of a differential equation using an unsupervised learning approach. The *neural operator method* takes a different approach. We describe this 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:

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

The function $u_T(x)$ is then a (possibly nonlinear and infinite dimensional) function $F(u_0(x))$ of the input function $u_0(x)$.

We now assume that we have a (large) number of solutions $u_T^i(x)$ to the differential equation at time T for a large number of different initial conditions u_0^i for I = 1... N giving a 'solution pair' Here the solutions 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. Then given a new initial condition u_0 as an input to the neural network it can find the future value u_T 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. We will explore this later in this section, as well as showing how the use of additional physics can increase the accuracy of this method.

To be precise, suppose that we have a neural network $N(u_0, \theta)$, with input (some finite dimensional representation of) the function u_0. Then we find the neural operator by training the parameters θ to minimise the loss function L given by

$$L = \sum_i \| \; u_T^i - N\left(u_0^i, \theta\right) \; \|$$

or similar. There are a number of ways of doing this which we will explore in this section.

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 (we will meet this ODE later). 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

$$u_T = B\,u_0 = \; e^{AT}u_0.$$

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.

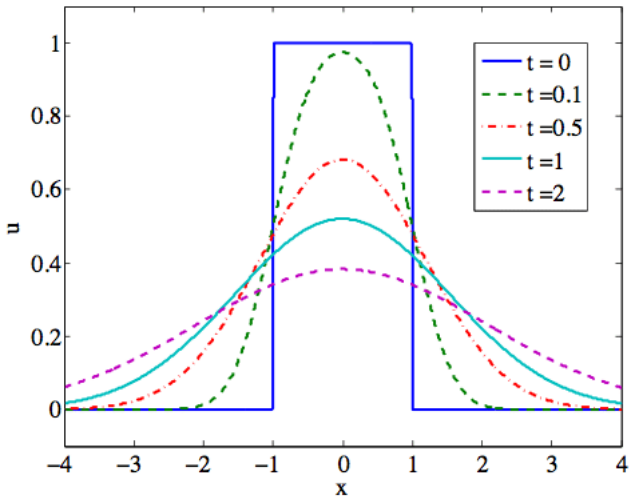
Important note A neural network is a *nonlinear function of the input*. Thus it will approximate the linear operator B by a nonlinear operator. This may not be optimal in practice. If you know in advance that the operator is linear then it is possible to use a more direct method, such as the Moore-Penrose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

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

A typical solution for a top-hat initial function is given below.



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 fact the error function) with the initial data, so that

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

Again, once the function G has been 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:

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

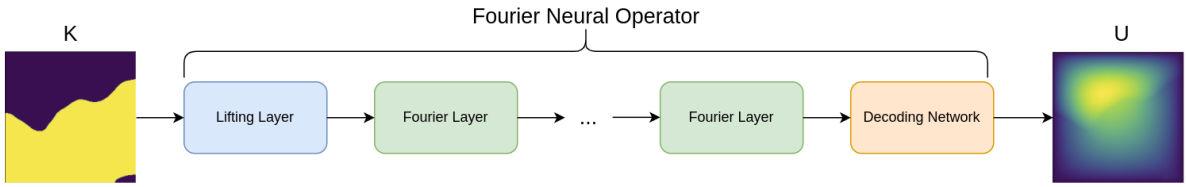
Provided that we can find the Fourier coefficients of u_0 then we can easily learn this linear map from the training data. This observation motivates the FNO algorithm of the next section

(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 N(k) of the input data k(**x**). As above, we can learn this function using a neural operator method such as the FNO described next.



4.3 The FNO method

There are now many method for computing a Neural Operator. One of the first of these was the Deep-O-Net method [16] which projects the input u_0 down 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 [17], developed at CalTech, 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.

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' Ψ acting on u_0 parametrised by θ, which is a deep NN of depth L, constructed as follows:

$$\Psi(u_0,\theta). \equiv Q\circ L_L\circ\ldots\circ L_2\circ L_1\circ P(u_0) \approx u_T = F(u_0),$$

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 this operation the FFT is restricted to M modes. Nonlinearity and higher order modes are introduced via the nonlinear activation function σ, 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 which will not be detected if the Neural Operator is only trained on data with 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:

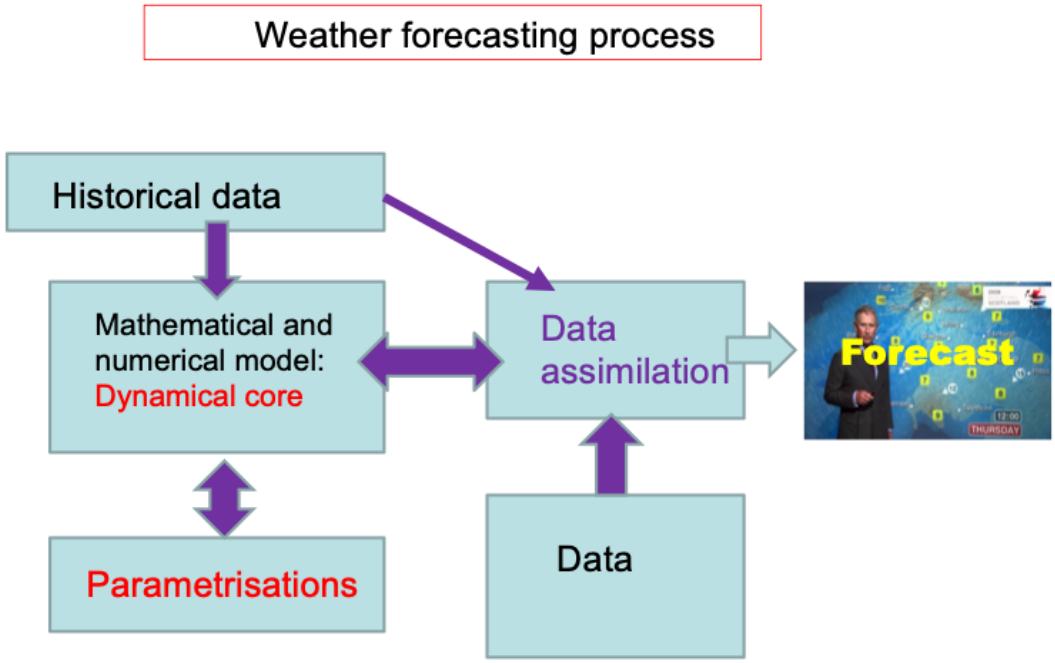
1. A set of PDEs describing the oceans and atmosphere etc. is constructed using physical reasoning based on historic data. This is called the 'dynamic core'. It is solved using a suitable numerical discretisation.

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

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 [18] model 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 [19].

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 use 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 in one 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). The ODE can then be solved by a numerical metho such as RK4 etc.

This ODE can then be trained to find appropriate x(T) given input x(0). 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 .

The Neural ODE methods were introduced by Chen et. al. [20] and the following figure illustrating them is taken from their paper:

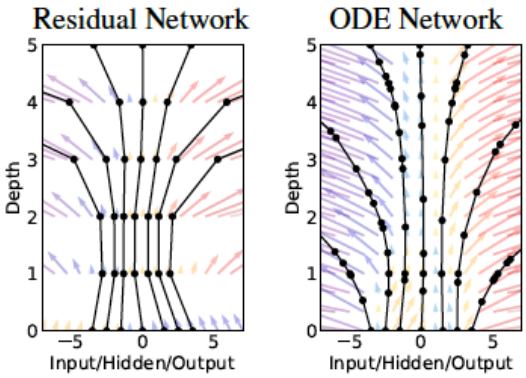


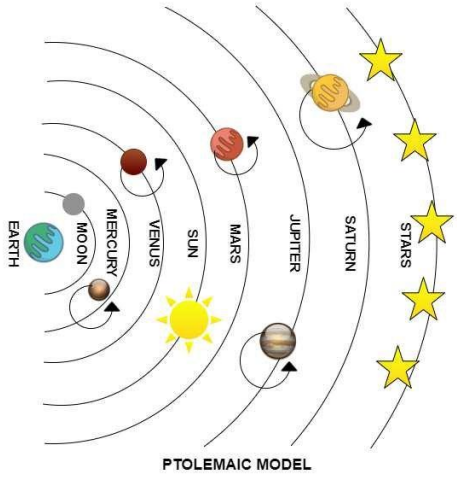
Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

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 adjoin methods.

5.2 Reconstructing dynamics: Neural ODES II

Suppose that we have some physical system, with an output/state x_i the time t_i and we record the measurements $y_i = H(x_i) + e_i$ where e_i is a noisy error. A key question in science, is: 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, or Newton's law of gravitation) of what was really going on.

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

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

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 $F(y;\theta)$ is a MLP or similar. This system can be solved using a standard numerical method and $z(t_i) = z_i$ evaluated.

Given the set of data points as above, we can then train (**) 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 from the Matlab website on Neural ODES [21]

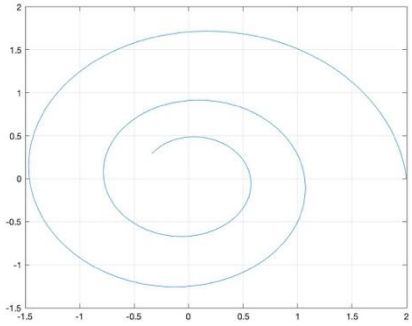
Let x satisfy the linear ODE

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

with

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

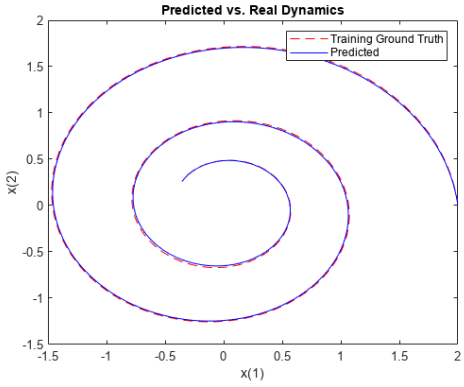
This has solutions which spiral into the origin for all initial data. An orbit for which x(0) = [2,0] is given below, computed using RK4 over the interval t ∈ [0,15] with 2000 data points



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

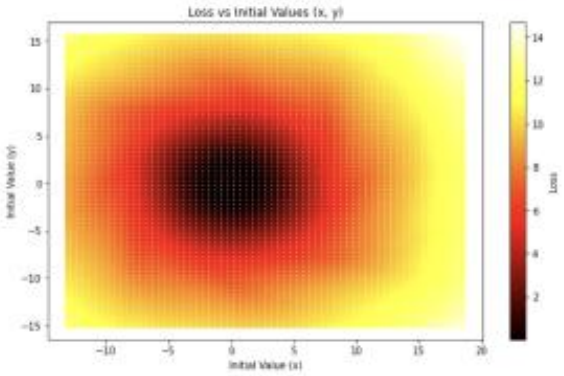
$$f(\mathbf{z}, \theta) = W_2 \ \tanh\left(W_1 \ \mathbf{z} + \mathbf{b}_1\right) + \mathbf{b}_2.$$

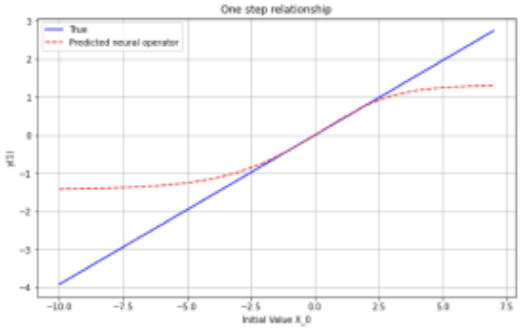
This is then trained by solving the ODE numerically and using the first 400 data points. If this is done then the trained neural net finds the rest of the orbit with ease as seen below.



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 as illustrated below (left). We would not be able to use the neural ode in the regions coloured yellow for example.





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, it rapidly saturates. On the right we see the approximation of F(x) (in one-dimension) compared to that of the neural ode. The impact of the 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.

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) Brunton et. al. [22] 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

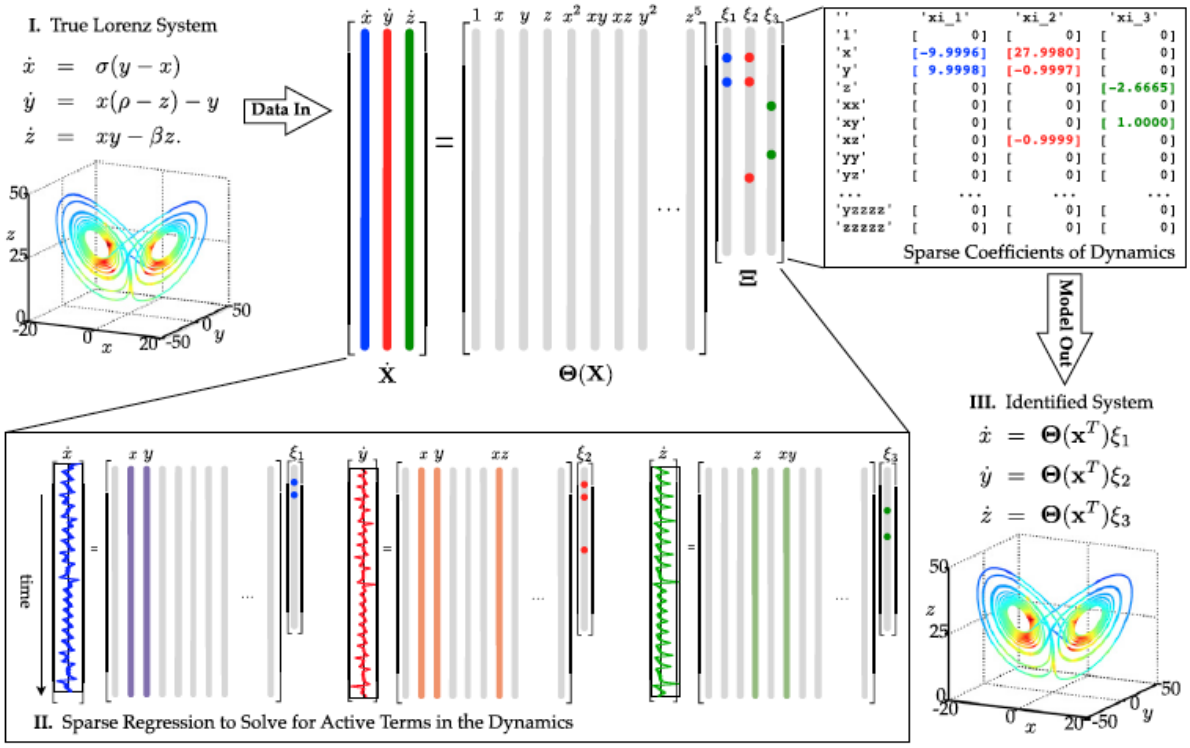
$$\Xi = \left[\xi_1\ \xi_2\ \cdots\ \xi_n\right]$$

then the approximating ODE becomes

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

As in the case of the Neural ODE this 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 [22] for the Lorenz equations see below:



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 effective for

- Describing the flow through a NN
- Learning dynamics from data
- They inherently non smooth (or close to non smooth)
- They work best when informed by the physics

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 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

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

This is a collection of articles on adversarial attack.

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

Section 2: Architectures

[6] 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.

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

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

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

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

A true classic introducing the transformer architecture!

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

[11] 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

[12] P. G. Ciarlet, The Finite Element Method for Elliptic Problems, (1978), North-Holland.

[13] 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

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

[15] Weinan E, 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

[16] 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

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

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

This paper introduces the GraphCast algorithm for weather forecasting

[19] 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

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

[21] Matlab tutorial on neural ODEs

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

[22] 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