Polynomial approximation numerical integration

A parallel approach to Initial Value Problems through optimization

v0.9

Yiorgos Panagiotopoulos

May 2024

Abstract

We present a short introduction to neural ordinary differential equations and how they emerge as continuous time generalizations of residual neural networks. One of the main bottlenecks of neural ODEs has been their speed both on training and inference. The numerical solver used for the forward and backward pass of the network can greatly impact its performance both in terms of speed and accuracy. We develop a new numerical ODE solver based on refining a polynomial approximation of the solution and explore its impact on this framework. At the end we showcase how our method compares to some commonly used numerical solvers.

Chapter 1

Introduction

The machine learning scene in 2024 is dominated by gargantuan models with billions of parameters. Even though they have demonstrated impressive results the compute and energy requirements of such models are a concern. It's the author's opinion that we ought to transition to analog or hybrid models of computation to reduce energy usage and increase speed. Even though digital computers are not going away any time soon studying alternative learning frameworks could ease the transition.

Neural differential equations re-emerged in the recent years after [1]. It's considered that they have great potential especially in:

- modelling physical phenomena governed by differential equations
- irregularly sampled time series
- generative models including continuous normalising flows and stochastic differential equations

More importantly, in out opinion, they highlight a connection between deep learning and dynamical systems. Even thought we still operate in the discrete domain to use them in our current systems they offer an opportunity to close the gap between continuous time models and machine learning.

We provide a quick overview of Neural ODES and propose a novel numerical integration method to tackle some of them limitations.

Chapter 2

Residual Networks

In order to understand the intuition behind neural ordinary differential equations one should have a basic knowledge of residual networks. Residual networks, also known as ResNets, were introduced by He and others in their seminal paper [2], to address the problem of *degradation* in very deep architectures. It had become apparent by state of the art models of that time like [3] that depth plays a very important role in vision tasks, including classification. Stacking layers (depth) allows for the integration of low/mid/high level features in the learning process. The immediate obstacle in deep learning is the notorious vanishing gradient problem, but this has been largely solved through batch/group normalisation [4].

It had been noticed though [5], that while stacking more layers, accuracy is initially saturated, unsurprisingly, but then it degrades rapidly. This degradation phenomenon is not caused by overfitting as the training error increases with the number of layers. ResNets solved this problem by utilising a residual learning framework.

2.0.1 Residual Learning

One can prove, by construction, that a deep architecture can be as accurate as a corresponding shallower one. This is achieved by adding identity layers in-between the layers of the shallower model. In more detail, consider a network that converges with some accuracy. Adding more layers to it would achieve the exact same accuracy if the extra layers learned to map their inputs straight to their outputs effectively making them identity mappings. The existence of this artificially deep network suggests that deep networks shouldn't produce higher error that the shallower ones. But the empirical evidence show that optimisers can't find this solution, at least in sensible time limits.

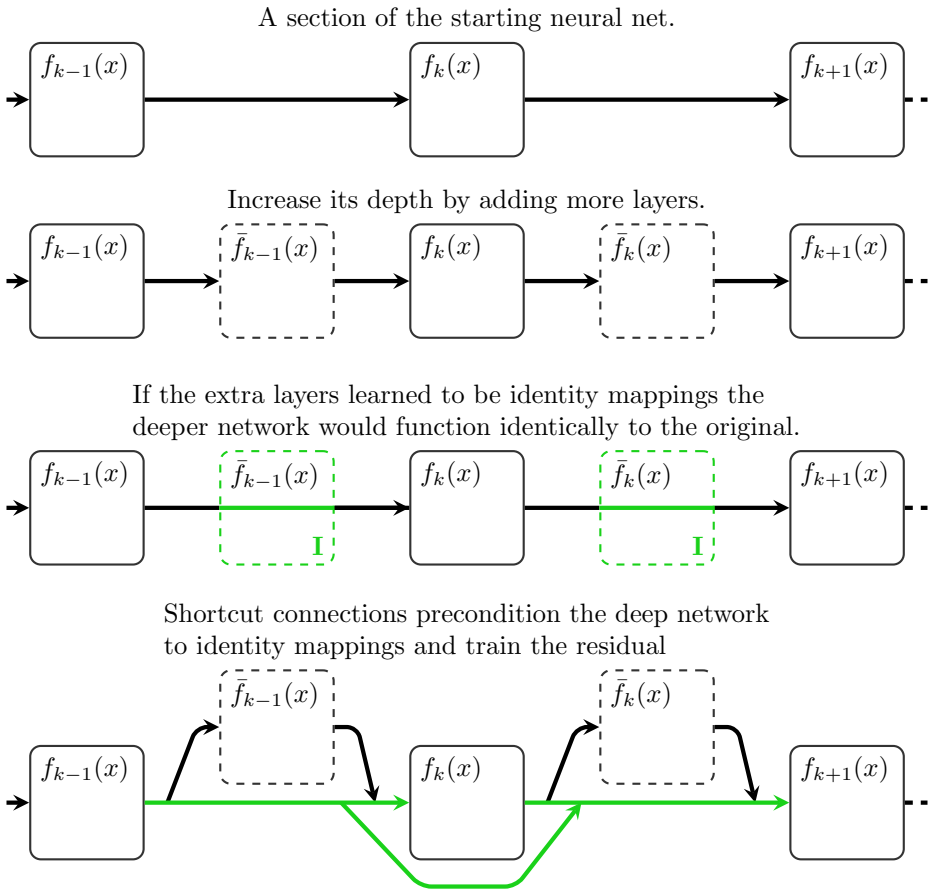

Figure 2.1: A visual explenation of residual learninig. Information flows through the shortcut connections as if they were identity mappings. The network learns the residual instead of a direct mapping. Residual neural networks use this property to achieve very large depth.

In order to incorporate this observation into the network, so called *shortcut connections* are introduced to the layers. Basically, the input to a block of stacked layers is added back to it's output. This way the network instead of learning the direct mapping $\mathcal{H}(x) = \mathcal{F}(x)$, it learns the residual mapping $\mathcal{H}(x) = \mathcal{R}(x) + x$. The optimiser can then push the residual mappings to 0 leading to identity connections. Even though identity mappings are unlikely to be optimum, experiments showed that the residual connections have generally small responses, meaning identities are good pre-conditioners for deep architectures.

Since their conception Residual Networks have revolutionised deep learning allowing

for much deeper architectures than previously used. Furthermore, the idea of residual learning has found vast appeal in many other architectures. Transformers for example, employ residual connections, allowing many such layers to be stacked creating large expressive models [6].

Chapter 3

Neural ODEs

There exist examples in the literature of combining neural networks and dynamical systems, even since the 1990s [7]. Interest began to increase after the publication of ResNet [8]. But the field of Neural Differential Equations really became prominent with [1].

3.0.1 A continuous time network

Let's revisit Residual Networks and how they work. The input-output relationship of the k-th discrete ResNet block is given by:

$$\boldsymbol{y}_{k+1} = \boldsymbol{y}_k + f(\boldsymbol{y}_k; \boldsymbol{\theta}_k) \tag{3.1}$$

where $\boldsymbol{y}_{k+1} \in \mathbb{R}^{N_y}$ is the output of the block, $\boldsymbol{y}^k \in \mathbb{R}^{N_y}$ is its input and f is a fully connected layer with parameters $\boldsymbol{\theta}_k \in \mathbb{R}^{N_\theta}$. Notice that we refer to ResNet blocks, not layers, since residual connections are generally not between the input and output of a single linear layer but a stack of them; from now on we will use the terms interchangeably.

The update in the *hidden state* $\boldsymbol{y}$ in (1) resembles the formula of the forward Euler iteration for solving ordinary differential equations. This observation led some researchers [8]–[10] to make a connection between neural network architectures and differential equations. Suppose that we could keep increasing the depth of the ResNet in (1) while the time of the forward pass remained bounded, meaning each layer would take less and less time. In the limit we get a differential equation governed by the dynamics defined by a neural network f:

$$\frac{\boldsymbol{y}(t)}{dt} = f(\boldsymbol{y}(t); \boldsymbol{\theta}) \tag{3.2}$$

Instead of a discrete sequence of hidden states, $\boldsymbol{y}(t)$ is a continuous flow $\boldsymbol{y} : [0, T] \to \mathbb{R}^{N_y}$ defined be the vector field parameterised by $\boldsymbol{\theta}$.

One important issue we haven't addressed is that the network's weights, $\boldsymbol{\theta}_k$, in (1) are different for each k, but at (2) we consider them to be constant in time. For notation's sake we could define f as $f(\boldsymbol{y}(t), t, \boldsymbol{\theta})$ meaning at certain "depths" different sections of the weights vector $\boldsymbol{\theta}$ are used, maybe in a piecewise constant fashion [11]. Alternatively, the weights could be time dependant which complicates the model but leads to *Galerkin* Neural ODEs, a more suitable continuous time equivalent of ResNet [12]. For our purposes we consider the weights constant for the rest of this paper.

3.0.2 Inference

Under this new machine learning framework depth is replaced by time. Inference is performed by solving an initial value problem from initial time $t = 0$ to terminal time $t = T$, i.e.:

$$\boldsymbol{y}(T) = \boldsymbol{y}(0) + \int_0^T f(\boldsymbol{y}(\tau); \boldsymbol{\theta})\, d\tau \tag{3.3}$$

Obviously, Eq. (3) describes the solution of a continuous time differential equation, while, as we know, our computers work in discrete time *(for now)*. In order to overcome this obstacle we employ a numerical method for solving *Initial Value Problems*, . If we were to use forward Euler, an explicit method, we would get a recurrence relation of the form

$$\boldsymbol{y}(t_k + h) = \boldsymbol{y}(t_k) + h \left.\frac{d\boldsymbol{y}(t)}{dt}\right|_{t=t_k} \tag{3.4}$$

By considering that $\frac{d\boldsymbol{y}(t)}{dt} = f(\boldsymbol{y}(t); \boldsymbol{\theta})$ Eq. (4) can be written as:

$$\boldsymbol{y}(t_k + h) = \boldsymbol{y}(t_k) + h f(\boldsymbol{y}(t_k); \boldsymbol{\theta}) \tag{3.5}$$

which coincides with the ResNet formula. Many other neural networks of deep architectures can be interpreted as solving a neural ODE using different methods [13]. The field of numerical differential equation solvers is quite vast meaning one could choose one that fits his specifications. A notable case is that of adaptive solvers like Runge-Kutta-Fehlberg, that use varying step sizes. This way to obtain $\boldsymbol{y}(T)$ the solver may require different number of f evaluations depending on the input and the complexity of the vector field at that input. For simple dynamics the solver can "decide" to use large step sizes meaning fewer function evaluations. Otherwise when the vector field is more complex, a smaller step size will be used to reduce error thus leading to more function evaluations. We could interpret this behaviour as a network of "variable depth".

It's important to clarify here that there are effectively two networks at play. Firstly, there is $f(\cdot)$, inherited from a block or layer of the original formulation of the problem. It is a neural network in the classical sense comprised of matrix vector multiplications and non-linearities. Secondly, there is the outlining model, the neural differential equation, which receives input $\boldsymbol{y}_0$ and produced output $\boldsymbol{y}(T)$; $f(\cdot)$ is to neural ODE what a layer

is to ResNet. Internally, depending on the solver, $f(\cdot)$ will be evaluated for many values of $\boldsymbol{y}(t)$. As disguised above, in contrast to ResNet $f(\cdot)$ contains all the learnable parameters of the model, which in out case are independent of depth. In conclusion the Neural ODE consists of: a neural network $f(\cdot)$ -that defines a *learnable* vector field- parameterised by $\boldsymbol{\theta}$, some input y_0 and a numerical solver that applies $f(\cdot)$ until it reaches $\boldsymbol{y}(T)$.

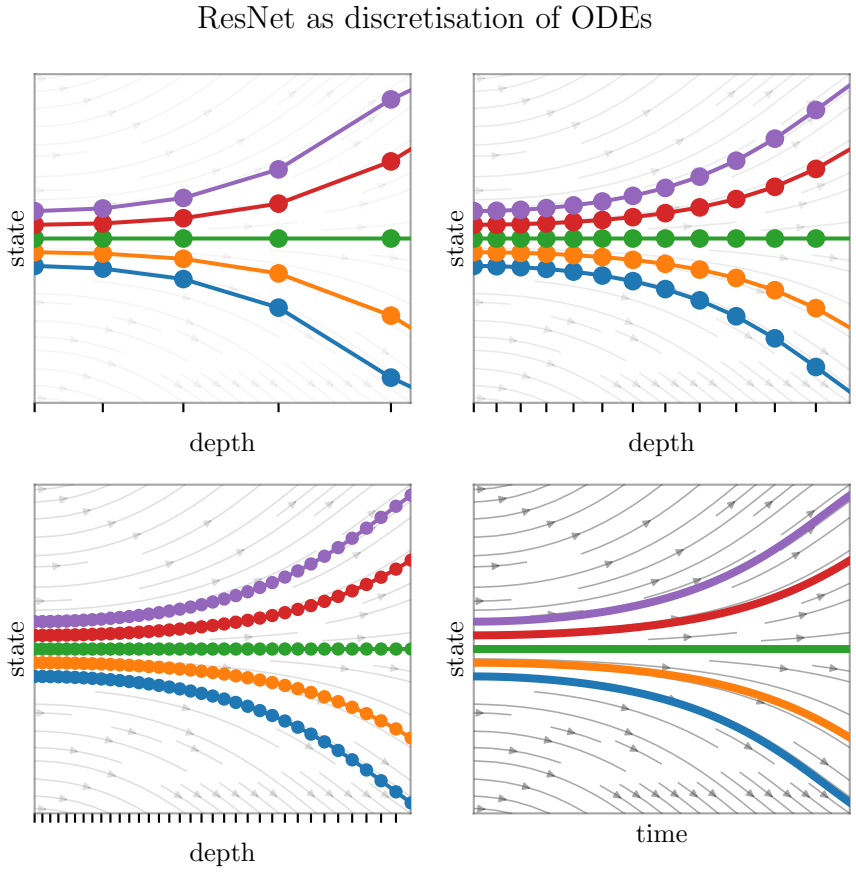

Figure 3.1: ResNets can be considered as discretizations of an ODE solution. Under this assumption Resnet defines a continuous time vector field.

3.0.3 Training

Optimising any model through gradient descent, in a supervised learning setting, requires acquiring the gradient of some loss function $\mathcal{L}(\boldsymbol{\theta})$ with respect to (wrt.) the

model's parameters $\boldsymbol{\theta}$. Usually, the loss for each sample in the dataset is calculated using a cost, or distance, function between the output of the network and the desired or ideal output $\boldsymbol{y}^*$. Sometimes, the loss function may be dependent on the state on multiple times. But for simplicity's sake we will examine the case where it only depends on the state at time $t = T$, which as is is apparent, depends indirectly on parameters θ.

$$L(\boldsymbol{y}(T), \boldsymbol{y}^*) = L\left(\boldsymbol{y}(0) + \int_0^T f(\boldsymbol{y}(\tau), \tau, \boldsymbol{\theta})\, d\tau\ , \boldsymbol{y}^*\right) \tag{3.6}$$

One could naively solve the initial value problem using any numerical ODE solver and try to backpropagate the gradients through the steps of the solver. This approach may be possible with the help of modern automatic differentiation software but its highly memory inefficient **matsubara2021symplectic**. In order to overcome this problem we are going to follow a different approach. To this end let's first define the gradient of L wrt the hidden state at some time t, i.e.:

$$\boldsymbol{a}(t) = \frac{\partial L}{\partial \boldsymbol{y}(t)} \tag{3.7}$$

The above quantity is called the *adjoint state*. It can be proven that for its derivative [14], the following equation holds:

$$\left.\frac{d\boldsymbol{a}(t)}{dt}\right|_{t=t_k} = -\boldsymbol{a}(t)^T \left.\frac{\partial f(\boldsymbol{y}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{y}(t)}\right|_{t=t_k} \tag{3.8}$$

Notice that we can easily compute $\boldsymbol{a}(T)$ as we know the form of the loss function eg. mean square error loss. Having this initial condition $\boldsymbol{a}(T)$ we can solve the differential equation (8) backwards in time to find $\boldsymbol{a}(t)$ for any t. Solving the equation, values of $\boldsymbol{y}(t)$ may be required. We can get them by solving the initial differential equation (3) starting from $\boldsymbol{y}(t)$ and again moving from time T to time 0.

Our goal is to calculate $\nabla_{\boldsymbol{\theta}} L$ which can be done by solving another differential equation or equivalently by evaluating the following integral:

$$\nabla_{\boldsymbol{\theta}} L = \int_T^0 \boldsymbol{a}(t)^T \frac{\partial f(\boldsymbol{y}(\tau), \boldsymbol{\theta})}{\partial \theta}\, d\tau \tag{3.9}$$

All these results stem from optimal control theory and more specifically consist specialised cases of Pontryagin's Maximum Principle. A discussion on how to prove (8) and (9) in a simpler way is presented on Appendix B.

3.0.4 Applications, benefits and limitations

There are examples in the literature where researchers have used Neural ODEs to simulate physical problems as the continuous time dynamics fit such problems well. Recently

the authors of [15] showed Neural CDEs, a variant of neural ODEs, lend themselves well to video modelling tasks, with performance comparable to classical neural nets. Other applications include modelling for irregularly sampled time series and continuous normalising flows [1], [11]. More generally Neural ODEs could be seen as a drop in replacement for ResNets.

Despite the scientific interest in the last years continuous time models have mostly remained in the lab. One major cause for this is the speed of inference and training. Neural ODEs typically require more function evaluations than classical models. Furthermore, one subtle limitation is that as the vector fields becomes more and more complex to fit the data the harder and more expensive it gets to solve them numerically. For those reasons Neural ODEs tend to become computationally intractable for large datasets. Researches have tried to combat these problems using methods like hypersolvers [16] and algebraically reversible solvers [17], [18] . Another way to speed up continuous time networks is to speed up the numerical solver through parallelism **gander201550**.

Chapter 4

Numerical Solvers

Ordinary differential equations are heavily featured in many scientific and engineering disciplines. We usually wish to solve problems where the dynamics of some system f along with some initial state y_0 are known, at some initial time t_0 and we are tasked to find the value of the state vector at some time t_1. This problem can be formulated as an initial value problem or IVP .

$$\frac{dy}{dt} = f(t, y), \quad y(0) = y_0 \tag{4.1}$$

The exact solutions to these ODEs is usually hard to calculate or intractable. To this end, many numerical methods have been developed over the years.

4.0.1 Introduction

In this chapter we will make a short reference to some basic theoretical prerequisites of numerical ODE solver. Then we will showcase some important methods. For simplicity we will focus on scalars but all the theorems are easily extensible to vectors. Moreover, we can consider higher dimensional IVPs as a (coupled) collection of scalar ODEs, so the same principles as those presented here apply.

In general even if f is continuous it is not guaranteed that the there exists a unique solution to IVP (10). Fortunately, under some mild conditions on f the existence and uniqueness of a solution is ensured through the following theorem.

Theorem 1 (Picard-Lindelöf theorem) *Suppose that $f(\cdot, \cdot)$ is a continuous function of its arguments in a region U of the (t, y) plane which contains the rectangle*

$$\boldsymbol{R} = \{(t, y) : t_0 \leq t \leq T, |y - y_0| \leq Y\},$$

where $T > t_0$ and $Y > 0$ are constants. Suppose also, that there exists a positive constant L such that

$$|f(t, y_\alpha) - f(t, y_\beta)| \leq L|y_m - y_\beta| \tag{4.2}$$

holds whenever (t, y_α) *and* (t, y_β) *lie in the rectangle* $\boldsymbol{R}$*. Finally, letting*

$$M = \max\{|f(t,y)| : (t,y) \in \boldsymbol{R}\},$$

suppose that $M(T - t_0) \leq Y$*. Then there exists a unique continusously differentiable function* $t \mapsto y(t)$*, defined on the closed interval* $[t_0, T_M]$*, which satisfies* (10)*.*

The condition (11) is called **Lipschitz condition** and L is called the **Lipschitz constant** for f. For most applications we are interested in, we can assume that f is Lipschitz. There also exists a an extension of Picard's theorem to multivariate functions $f : \mathbb{R}^m \to \mathbb{R}^m$. The theorem for multiple dimensions is analogous to the scalar case except that the absolute function above is replaced with the Euclidean norm on vectors.

An important property in the study of differential equations and numerical solvers is that of stability.

Definition 4.0.1 *A solution* $y = v(t)$ *to* (3) *is said to be* ***stable*** *on the interval* $[t_0, T]$ *if for every* $\epsilon > 0$ *there exists* $\delta > 0$ *such that for all* z *satisfying* $|v(t_0) - z| < \delta$ *the solution* $y = w(t)$ *to the differential equation* $\frac{dy}{dt}y = f(t, y)$ *satisfying the initial condition* $w(t_0) = z$ *is defined for all* $t \in [t_0, T]$ *and satisfies* $|v(t) - w(t)| \leq \epsilon$ *for all* t *in* $[t_0, T]$*. A solution which is stable on* $[t_0, \infty)$ *(i.e. stable on* $[t_0, Y]$ *for all* Y *and with* δ *independent of* T*) is said to be* ***stable in the Lyapunov sense****. Moreover, if*

$$\lim_{x\to\infty} |v(x) - w(x)| = 0,$$

then the solution $y = v(t)$ *is called* ***asymptotically stable****.*

One can prove [19] that if f at (10) is Lipschitz continuous the solution to the IVP is stable on $[t_0, T]$.

4.0.2 Forward Euler

The Euler method, also known as forward Euler, is one of first developed and hence one of the most basic numerical ODE solvers. It works by partitioning the interval on which we wish to solve the initial value problem to a mesh of M discrete points $t_n : k = 0, 1, \ldots M - 1$ with distances h_k between them, without loss of generality less assume the points are equidistant and $h = t_{k+1} - t_k$. Supposing we know the value of the function y at some time t_0, Euler's method allows us to iteratively calculate an approximation y_n for the value of $y(t_n)$ at the mesh points. It can be derived directly by the forward difference approximation of the derivative.

$$\left.\frac{dy}{dt}\right|_{t=t_n} = \lim_{h\to 0} \frac{y(t_n + h) - y(t_n)}{h} \tag{4.3}$$

$$\approx \frac{y(t_n + h) - y(t_n)}{h} \quad \text{for small } h \tag{4.4}$$

Rearranging terms in the above equation and substituting the known function for the derivative we arrive at the Euler iteration.

$$y_{k+1} = y_n + hf(t_n, y_n))$$

The local truncation error is the error of the approximation compared to the true solution for one step of the algorithm supposing $y(t_n) = y_n$ We can intuitively understand that as the *step size* at (4.2) gets smaller the lower the truncation error becomes. This can be shown formally considering the Taylor expansion of y around t_n

$$\begin{aligned} y(t_n + h) &= y(t_n) + hy'(t_n) + \frac{1}{2}h^2 y''(t_n) + \mathcal{O}(h^3) \\ y(t_n + h) &= y_{n+1} + \frac{1}{2}h^2 y''(t_n) + \mathcal{O}(h^3) \\ y(t_n + h) - y_{n+1} &= \frac{1}{2}h^2 y''(t_n) + \mathcal{O}(h^3) \\ \text{local error} &= \mathcal{O}(h^2) \end{aligned}$$

So the error scales inversely with the square of the step size, as we will see later higher order solver sacrifice some of the speed of Euler's method, which require only one evaluation of f per iteration, to achieve error that scales inversely with higher powers of the step size. The above analysis refers to the error of one step, as the algorithm continues the error from previous iterations accumulates. Under some loose conditions, namely f is Lipschitz continuous on y and y has a bounded second derivative it can be proven that the global truncation error is roughly proportional to h [20]. The exact upper limit of the global truncation error has a more complex but not of significant importance for our case. This property, the global truncation error being $O(h)$, makes forward Euler a *first order* method.

Implicit Euler

An alternative derivation of the Euler method is found by integrating the ODE between two consecutive mesh points t_n and t_{n+1}

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(\tau, f(y(\tau)))t\tau$$

and applying a numerical integration technique known as the rectangle rule

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + hf(t_n, f(y(t_n))) \\ y_{n+1} &= y_n + hf(t_n, y_n) \end{aligned}$$

Different numerical integration methods, called quadratures, give rise to different numerical solvers, for example the trapezium rule results in the following formula

$$y_{n+1} = y_n + \frac{1}{2}h\left[f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right]$$

Notice that in the above equation y_{n+1} appears on both left and right hand sides. This is called an explicit Euler method and requires the solution of this implicit equation for each n.

4.0.3 General explicit one-step method

A general explicit one-step method expands the class of Euler solver and it can be written in the form:

$$y_{n+1} = y_n + h\Phi(t_n, y_n; h), \quad n = 0, 1, \ldots, M-1$$

Where Φ is a continuous function on its variables, in the case of forward Euler $\Phi(t_n, y_n; h) = f(t_n, y_n)$.

The global error for methods of the form is given by:

$$e_n = y(t_n) - y_n$$

The truncation error T_n is given by:

$$T_n = \frac{y(t_{n+1}) - y_{n+1}}{h} = \frac{y(t_{n+1}) - y(t_n)}{h} - \Phi(t_n, y(t_n); h) \tag{4.5}$$

The next theorem provides a bound on the global error in terms of the truncation error. It can be shown that if Φ is Lipschitz continuous in its second argument the explicit one-step method's global error is bounded in terms of the local truncation error. This suggests that if as truncation error goes to zero as $h \to 0$ then the global error also converges to zero (as long as $|e_0| \to 0$ when $h \to 0$). This observation motivates the following definition.

Definition 4.0.2 *A general explicit one-step numerical method is said to be* ***consistent*** *with the IVP* (10) *if the truncation error defined in Eq. 14 is such that for any* ϵ *there exists a positive* $h(\epsilon)$ *for which* $T_n < \epsilon$ *for* $0 < h < h\epsilon$ *and any pair of points* $(t_n, y(t_n)), (t_{n+1}, y(t_{n+1}))$ *on any solution curve in* $\boldsymbol{R}$.

Besides consistent, we wish our solver to be accurate, to quantify the accuracy of a solver we define its order of accuracy.

Definition 4.0.3 *A general explicit one-step method is said to have* ***order of accuracy*** p*, if* p *is the largest positive integer such that, for any sufficiently smooth solution curve* $(t, y(t))$ *in* $\boldsymbol{R}$ *of the IVP* (10)*, there exists constants* k *and* h_0 *such that*

$$|T_n| \leq Kh^p \quad for\ 0 < h \leq h_0$$

for any pair of points $(t_n, y(t_n)), (t_{n+1}, y(t_{n+1}))$ *on the solution curve.*

Having introduced the concepts of consistency and the order of accuracy we can move to more involved and more accurate methods called Runge-Kutta methods.

4.0.4 Runge-Kutta methods

Runge-Kutta methods, introduced by Runge with significant contributions by Heun and Kutta, aim to improve Euler's method accuracy by introducing intermediate evaluations of f between t_n and t_{n+1}.

In their most general formulation of a Runge-Kutta method is given by:

$$y_{n+1} = y_n + \sum_{i=1}^{s} b_i k_i, \tag{4.6}$$

where

$$k_i = f\left(t_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j\right), \quad i = 1, \ldots, s$$

The above formula is quite terse, and actually includes both explicit and implicit Runge-Kutta methods, we will start from explicit methods and move to implicit ones.

Explicit Runge-Kutta methods

It is helpful to start from a specific example, namely the second order Runge Kutta method and generalize to higher orders. We aim to solve the same initial value as above again on some grid of points $t_n, k = 1, \ldots, M-1$. For a single time step we aim to find $y_{k+1} = y(t_n + h)$ knowing y_n and f. Ideally we would solve the ODE by integrating

$$y(t_n + h) = y_n + \int_{t_n}^{t_n+h} f(\tau, y(\tau)) d\tau$$

But calculating this integral is often hard or even impossible, so we approximate it using some quadrature rule which in general can be written as:

$$y(t_n + h) \approx y_n + h \sum_{i=1}^{N} \omega_i f(t_n + c_i h, y(t_n + c_i h))$$

The specific quadrature used generates and a different order Runge Kutta method. If we were for example to use $N = 4$ and the 3 point Simpson's quadrature rule we would derive the fourth order Runge-Kutta method commonly referred to as "RK4". But our goal here is to derive a general second order method so we use $N = 2$, and we also choose $c_1 = 0$.

$$y(t_n + h) \approx y(t_n) + h b_1 f(t_n, y_n)) + h b_2 f(t_n + c_2 h, y(t_n + c_2 h))$$

The first term we can easily calculate since we know all the relative quantities and we denote $k_1 = h\, f(t_n, y_n)$. The second term contains $y(t_n + c_2 h)$ which is unknown, but we

can calculate its Euler approximation and introduce $k_2 = h\, f(t_n + \alpha h, y_n + \beta k_1)$ where $a = c_2, b = c_2 - t_n$. Finally, we have the following (note that a third order method would also have k_3 term that would include k_1 and k_2, a fourth k_4 etc)

$$y(t_n + h) = b_1 k_1 + b_2 k_2$$

To find out the unknown variables b_1, b_2, α, β we replace the left hand side with its Taylor expansion around t_n:

$$y(t_n) + hy'(t_n) + \frac{h^2}{2!} y''(t_n) + \mathcal{O}\left(h^3\right) = y(t_n) + b_1 k_1 + b_2 k_2. \tag{4.7}$$

In the above equation $y'(t_n)$ is obviously $f(t_n, y(t_n))$ and $y''(t_n) = \frac{dy}{dt} f(t, y(t))\Big|_{t=t_n}$ since f has both direct and indirect dependence of t it is expanded as $\frac{d}{dt} f(t, y(t)) = f_t + f f_y$ from the multivariate chain rule, where f_x is $\frac{\partial f}{\partial x}(t, y(t))$, we omit f's arguments omitted for convenience. Substituting k_1, k_2 in (4.4.1) we get:

$$hf + \frac{h^2}{2}(f_t + f f_y) + \mathcal{O}\left(h^3\right) = b_1 hf + b_2 hf(t + \alpha h, y + \beta k_1). \tag{4.8}$$

We now apply the 2-dimensional Taylor expansion on the right-hand side of (4.4.1):

$$hf + \frac{h^2}{2}(f_t + f f_x) + \mathcal{O}\left(h^3\right) = b_1 hf + b_2(hf + \alpha h^2 f_t + \beta h^2 f f_x) + \mathcal{O}\left(h^3\right)$$
$$hf + \frac{h^2}{2}(f_t + f f_x) + \mathcal{O}\left(h^3\right) = (b_1 + b_2)hf + h^2(\alpha b_1 f_t + \beta b_2 f f_x) + \mathcal{O}\left(h^3\right)$$

In order for the Runge-Kutta method to agree with the Taylor expansion of the true value of y at $t_n + h$ up to $\mathcal{O}(h^3)$ accuracy we need to choose:

$$\begin{aligned} b_1 + b_2 &= 1, \\ \alpha b_1 &= 1/2, \\ \beta b_2 &= 1/2 \end{aligned}$$

Choosing different parameters that satisfy these constraints results to different second order Runge-Kutta methods.

Generally explicit methods of order N are of the form:

$$y_{n+1} = y_n + h \sum_{i=1}^{N} b_i k_i$$

where

$$k_1 = f(t_n, y_n), \tag{4.9}$$
$$k_2 = f(t_n + c_2 h, y_n + (a_{21} k_1)h), \tag{4.10}$$
$$k_3 = f(t_n + c_3 h, y_n + (a_{31} k_1 + a_{32 k_2})h), \tag{4.11}$$
$$\vdots \tag{4.12}$$
$$k_N = f(t_n + c_s h, y_n + (a_{s1} k_1 + a_{s2} k2 + \cdots + a_{s,s-1} k_{s-1})h) \tag{4.13}$$

Similarly to the second order case to specify a N-th order method we need the coefficient a_{ij} called the Runge-Kutta matrix, the weights b_i and the nodes c_i. It is also true in the general case that for a Runge-Kutta method to be consistent it is a necessary and sufficient condition for b_i to hold that.

$$\sum_{i=1}^{s} b_i = 1$$

A convenient representation for the equations at (15) is the Butcher tableau, it contains the coefficients matrix, the weights and the nodes in a grid like structure. For an explicit method the Butcher tableau is written as:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s_1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline & & & & & \\ & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}$$

As we mentioned above the "RK4" method is the most well known explicit Rugne-Kutta method, its Butcher tableau looks like this:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & & & & \\ & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

We started this section with the simplest numerical ODE method forward Euler, it can also be expressed as a first order explicit Runge Kutta method whose Butcher tableau is:

$$\begin{array}{c|c} 0 & \\ \hline & \\ & 1 \end{array}$$

Chapter 5

Parallelism in Time

It is apparent that Moore's law is crumbling. In the last decades focus has shifted to parallel computation to increase performance rather than increasing single threaded frequencies [CITATION?]. In the field of deep learning, massive parallelisation of training and inference workloads have led to major breakthroughs [6]. Differential equations solvers have intrinsic serial characteristics. Each iteration of a time stepping algorithm depends on the previous steps. Nevertheless, after the seminal work presented in [21], several methods have been proposed for parallel in time, numerical integration. Recent advances include the parareal algorithm [22] and PFASST [23] algorithm. Massaroli et al. [24] used principles of multiple shooting layers, a subcategory of parallel in time methods, and root finding algorithms, to accelerate neural ODES. In the next subsection we introduce a new parallel in time algorithm which is based on the use of a polynomial approximation of the solution of the IVP.

5.0.1 Polynomial approximation

Let us consider the following initial value problem that we wish to solve in some time interval $t \in [0, T]$ with $\boldsymbol{y}(t_0) = \boldsymbol{y}_0$. We will examine the scalar case first where $y : \mathbb{R} \to \mathbb{R}$:

$$\frac{dy(t)}{dt} = f(y(t)) \tag{5.1}$$

We define the N-th order polynomial approximation $\hat{y}(t) = \sum_{n=0}^{N-1} b_n \phi_n(t)$ where $\phi_n(t)$ are basis functions for a vector space of polynomials and b_n are some coefficients we wish to find. We can also express $\hat{y}$ as the dot product $\hat{y}(t) = \boldsymbol{b}^t \boldsymbol{\phi}(t)$ where $\boldsymbol{\phi} : [0, T] \to \mathbb{R}^N$ and $\boldsymbol{b} \in \mathbb{R}^N$. By substituting $\hat{y}$ in (21) we get:

$$\frac{d}{dt}\boldsymbol{b}^T \boldsymbol{\phi}(t) \approx f(\boldsymbol{b}^T \boldsymbol{\phi}(t)) \tag{5.2}$$

$$\boldsymbol{b}^T \dot{\boldsymbol{\phi}}(t) \approx f(\boldsymbol{b}^T \boldsymbol{\phi}(t)) \tag{5.3}$$

We define an approximation error (in the mean square error sense) that we wish to minimise in the interval of interest.

$$e(\boldsymbol{b}) = \int_0^T \left(\boldsymbol{b}^T \dot{\boldsymbol{\phi}}(\tau) - f(\boldsymbol{b}^T \boldsymbol{\phi}(\tau))\right)^2 d\tau \tag{5.4}$$

Depending on the choice of basis functions, the derivative of $\boldsymbol{\phi}(t)$ with respect to time can be trivial to obtain analytically. We opt to use Chebyshev polynomials as the basis functions of our approximation which are known for their properties in approximation theory [25]. They form an orthonormal basis for functions in $[-1, 1]$ but we can use them in our desired interval through a simple change of variables while being mindful of the differential in $\frac{d}{dt}\boldsymbol{\phi}(t)$. More specifically, through the linear transformation $\hat{t} = 2 * \frac{t-\alpha}{\beta-\alpha} - 1$ we can move the integration interval from $[a, b]$ to $[-1, 1]$. By substituting $t = \hat{t}$ in (22) we get:

$$\begin{aligned}
\frac{d}{dt}\boldsymbol{b}^T\boldsymbol{\phi}(t) &= f(\boldsymbol{b}^T\boldsymbol{\phi}(t)) \\
\frac{d}{d\hat{t}}\frac{d\hat{t}}{dt}\,(\boldsymbol{b}^T\boldsymbol{\phi}(\hat{t})) &= f(\boldsymbol{b}^T\boldsymbol{\phi}(\hat{t})) \\
\frac{2}{b-a}\boldsymbol{b}^T\dot{\boldsymbol{\phi}}(\hat{t}) &= f(\boldsymbol{b}^T\boldsymbol{\phi}(\hat{t}))
\end{aligned}$$

For the sake of convenience for the rest of the section we omit the hat from t and the constant in front of $\dot{\boldsymbol{\phi}}(t)$ but keep in mind that the limits of integration are $[-1, 1]$

Chebyshev polynomials of the first kind $T_n(t)$ can be defined by:

$$T_n(\cos\theta) = \cos(n\theta) \tag{5.5}$$

While Chebyshev polynomials of the second kind can be defined by:

$$U_n(\cos\theta)\sin\theta = \sin((n+1)\theta) \tag{5.6}$$

Alternatively, they can be obtained through the recurrence relationships:

$$\begin{aligned}
&T_0(t) = 1 && U_0(t) = 1 \\
&T_1(t) = t && U_1(t) = 2t \\
&T_n(t) = 2tT_{n-1}(t) - T_{n-1}(t) && U_n(t) = 2tU_{n-1}(t) - U_{n-1}(t)
\end{aligned} \tag{5.7}$$

It can also be shown that for the derivative of T_n it holds that:

$$\frac{dT_n}{dt} = nU_{n-1} \tag{5.8}$$

Generalising to a vector valued $\boldsymbol{y} : [t_0 = -1, t_1 = 1] \to \mathbb{R}^L$ the collection of coefficients becomes a matrix $B \in \mathbb{R}^{N \times L}$. Another way of thinking about this is stacking L coefficient vectors, one for each dimension next to each other, forming the coefficient matrix B. The error function is then reformulated as

$$e(B) = \int_{t_0}^{t_1} \|B^T \dot{\boldsymbol{\phi}}(\tau) - f(B^T \boldsymbol{\phi}(\tau))\|^2 \, d\tau \tag{5.9}$$

Since we operate in discrete time we define a grid of M points $\boldsymbol{t} = [t_0 \; t_1 \; \ldots \; t_{M-1}]^T$ and replace the integral with a Riemann sum. The points are usually equidistant but other sets of points between t_0 and t_1 could be used. For our calculations we use the Chebyshev node of the second kind $t_m = \cos\left(\frac{m\pi}{M-1}\right), \; m = 0, \ldots, M-1$.

$$e(B) = \sum_m \|B^T \dot{\boldsymbol{\phi}}(t_m) - B^T f(\boldsymbol{\phi}(t_m))\|^2 \Delta t_m \tag{5.10}$$

We therefore seek to find coefficients B^* that minimise the above error. To find the minimum we set the derivative wrt. B equal to zero and solve for B

$$\nabla_B e(B) = 0 \tag{5.11}$$

Depending on the form of f it can be hard -or even impossible- to find a closed form solution to this problem. We therefore resort to an iterative process for finding B, starting from some B_0 we form the solution as $B_{k+1} = B_k + \eta \Delta B_k$, η being a positive scalar learning rate. Depending on the order of the optimisation method used ΔB_k takes different forms, most commonly (under the gradient descent schema) ΔB_k is defined as $\nabla_{B_k} e$.

A zero order optimisation algorithm

Let's substitute B at $(k+1)$-th iteration in 31:

$$\nabla_{B_{k+1}} \sum_m \|B_{k+1}^T \dot{\boldsymbol{\phi}}(t_m) - f((B_{k+1}^T \boldsymbol{\phi}(t_m))\|_2^2 \Delta t_m = 0 \tag{5.12}$$

$$\nabla_{B_{k+1}} \sum_m \|B_{k+1}^T \dot{\boldsymbol{\phi}}(t_m) - f\big((B_k + \eta \Delta B_k)^T \boldsymbol{\phi}(t_m)\big)\|_2^2 \Delta t_m = 0 \tag{5.13}$$

We substitute the function $g(B, t) = f(B^T \boldsymbol{\phi}(t))$ inside the sum with its zero-th order approximation around B_k, meaning the constant (on B) function $f(B_k^T \boldsymbol{\phi}(t))$. Exchanging the nabla and applying some matrix calculus we get:

$$\sum_m \Big(B_{k+1}^T \dot{\boldsymbol{\phi}}(t_m) - f\big((B_k^T \boldsymbol{\phi}(t_m)\big)\Big) \dot{\boldsymbol{\phi}}^T(t_m) \Delta t_m = 0 \tag{5.14}$$

$$\sum_m B_{k+1}^T \dot{\boldsymbol{\phi}}(t_m) \dot{\boldsymbol{\phi}}^T(t_m) \Delta t_m - f\big((B_k^T \boldsymbol{\phi}(t_m)\big) \dot{\boldsymbol{\phi}}^T(t_m) \Delta t_m = 0 \tag{5.15}$$

We notice in the above relationship that the sums of outer products can be rewritten as matrix multiplications, to that goal lets define the following quantities.

$$\Phi = \boldsymbol{\phi}(\boldsymbol{t}^t) = \begin{bmatrix} T_0(t_0) & T_0(t_1) & \dots & T_0(t_{M-1}) \\ T_1(t_0) & T_1(t_1) & \dots & T_1(t_{M-1}) \\ & & \vdots & \\ T_{N-1}(t_0) & T_{N-1}(t_1) & \dots & T_{N-1}(t_{M-1}) \end{bmatrix}$$

From (28) we can define the time derivatives of $\boldsymbol{\phi}$ as:

$$\dot{\Phi} = \dot{\boldsymbol{\phi}}(\boldsymbol{t}^t) = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 1 \cdot U_0(t_0) & 1 \cdot U_0(t_1) & \dots & 1 \cdot U_0(t_{M-1}) \\ & & \vdots & \\ (N-1) \cdot U_{N-2}(t_0) & (N-1) \cdot U_{N-2}(t_1) & \dots & (N-1) \cdot U_{N-2}(t_{M-1}) \end{bmatrix}$$

Using these matrices we can rewrite Eq. 35 in terms of matrix products as follows:

$$0 = B_{k+1}^T \dot{\Phi}(\dot{\Phi}^T \odot \Delta\boldsymbol{t} \cdot \mathbf{1}^T) - f\big(B_k^T \Phi\big)(\dot{\Phi}^T \odot \Delta\boldsymbol{t} \cdot \mathbf{1}^T) \tag{5.16}$$

$$B_{k+1}^T = f\big(B_k^T \Phi\big)(\dot{\Phi}^T \odot \Delta\boldsymbol{t} \cdot \mathbf{1}^T)(\dot{\Phi}(\dot{\Phi}^T \odot \Delta\boldsymbol{t} \cdot \mathbf{1}^T))^{-1} \tag{5.17}$$

Eq. 37 outlines the iteration scheme for this zero order optimisation algorithm. Omitting the partitions' lengths terms $\Delta\boldsymbol{t}$ temporarily for clarity:

$$B_{k+1}^T = f\big(B_k^T \Phi\big)\dot{\Phi}^T(\dot{\Phi}\dot{\Phi}^T)^{-1}$$

It becomes apparent that the inverted matrix is the same for all iterations meaning it only needs to be inverted once at the beginning. Furthermore, f is applied to each row of its input matrix in parallel, which provides a parallel speed-up to the algorithm.

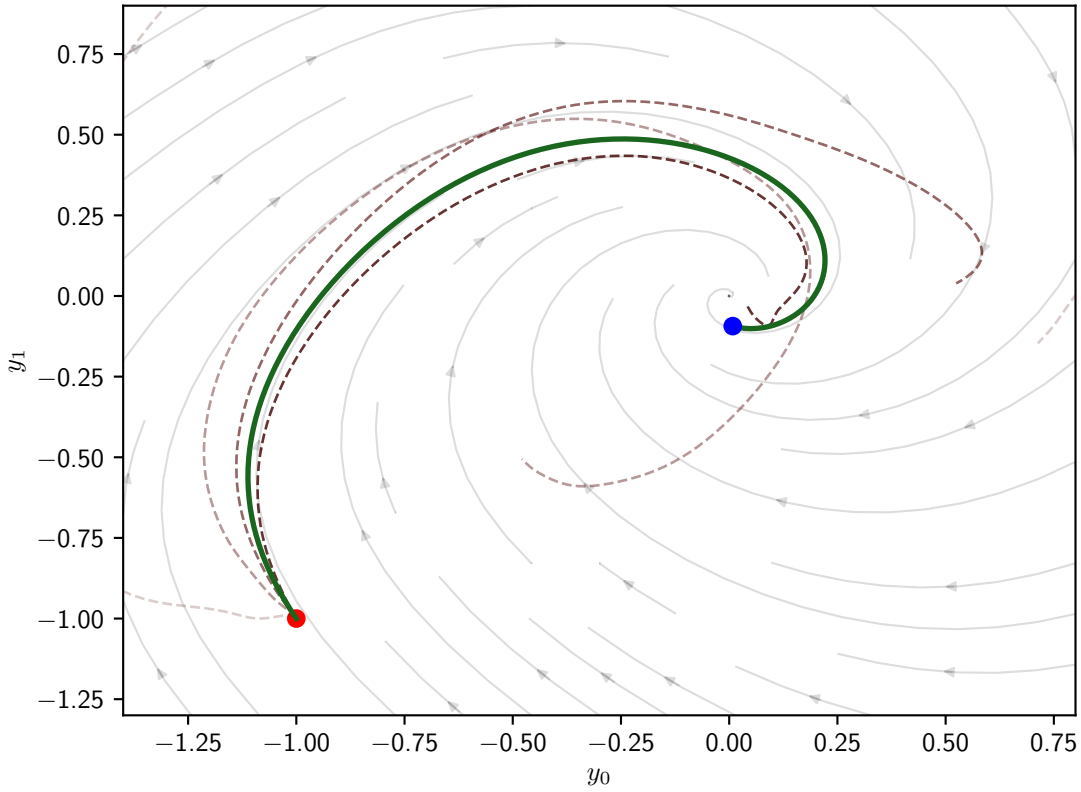

Figure 5.1: Regression of the approximate solutions to the true solution. As the coefficients go to the optimum and the error function approaches the minimum the trajectories are getting closer to the true solution.

Higher order minimization

It's apparent that the accuracy of this iterative solution depends on the residual of the zero order approximation on Eq. (35). The zero order algorithm converges rather fast but can't achieve high accuracy. Experimental results show that it oscillates around the optimum.

To combat that, we can use the solution matrix $\bar{B}$ as the starting value of a higher order method to increase accuracy. A simple choice is a fist degree optimisation method such as gradient descent or one of each variants.

Another choice is the Newton-Raphson algorithm with Hessian modification, a second order method. A version of the algorithm 5 is showcased in Appendix D.1, where line-search is used for the learning rate or step size of the update.

Combining polynomial approximation with the Newton algorithm we arrive at algorithm 14.

Algorithm 1 Polynomial approximation numerical integration

choose ϵ_{ls}, ϵ_n, B_0, K_max
$\boldsymbol{t} \leftarrow [t_0\ t_1\ t_2\ \dots\ t_{M-1}]^T$
$k \leftarrow 1$
calculate $\Phi, \dot{\Phi}$
repeat
  calculate $e(B_k]$
  if $e(B^k) < \epsilon_n$ **then**
    return B_k
  end if
  $\boldsymbol{d}^{[k]} \leftarrow -\left[\nabla^2 e(B^{[k]})\right]^{-1} \nabla e(B^{[k]})$
  $\alpha^{[k]} \leftarrow \text{lineseach}(e, B^{[k]}, \boldsymbol{d}^{[k]})$
  $B^{[k+1]} \leftarrow B_k + \alpha \boldsymbol{d}^{[k]}$
  $k \leftarrow k + 1$
until $k > k_max$

5.0.2 Realisation details

Imposing initial conditions

Since the value of $y(t)$ and $f(y(t))$ are known at $t = 0$ we can calculate a part of B analytically which allows to both: impose our initial conditions and to reduce the computational cost. Starting with initial state:

$$\boldsymbol{y}(t_0) = B^T \boldsymbol{\phi}(t_0) = B_0 \phi_0(t_0) + B_1 \phi_1(t_0) + B_{2:}^T \boldsymbol{\phi}_{2:}(t_0) \tag{5.18}$$

We can write a similar equation for $f(\boldsymbol{y}_0)$.

$$f(\boldsymbol{y}(t_0)) = B^T \dot{\boldsymbol{\phi}}(t_0) = B_0 \dot{\phi}_0(t_0) + B_1 \dot{\phi}_1(t_0) + B_{2:}^T \dot{\boldsymbol{\phi}}_{2:}(0) \tag{5.19}$$

We can group this two equation in system:

$$\begin{bmatrix}\boldsymbol{y}_0 & \boldsymbol{f}_0\end{bmatrix} = B_0 \begin{bmatrix}\phi_0(t_0) & \dot{\phi}_0(t_0)\end{bmatrix} + B_1 \begin{bmatrix}\phi_1(t_0) & \dot{\phi}_1(t_0)\end{bmatrix} + B_{2:}^T \begin{bmatrix}\boldsymbol{\phi}_{2:}(t_0) & \dot{\boldsymbol{\phi}}_{2:}(t_0)\end{bmatrix} \tag{5.20}$$

Which can be written as:

$$\begin{bmatrix}\boldsymbol{y}_0 & \boldsymbol{f}_0\end{bmatrix} = \begin{bmatrix}B_0 & B_1\end{bmatrix} \begin{bmatrix}\phi_0(t_0) & \dot{\phi}_0(t_0) \\ \phi_1(t_0) & \dot{\phi}_1(t_0)\end{bmatrix} + B_{2:}^T \begin{bmatrix}\boldsymbol{\phi}_{2:}(t_0) & \dot{\boldsymbol{\phi}}_{2:}(t_0)\end{bmatrix} \tag{5.21}$$

Solving for $[B_0\ B_1]$ we have:

$$\begin{bmatrix} B_0 & B_1 \end{bmatrix} = \left(\begin{bmatrix} \boldsymbol{y}_0 & \boldsymbol{f}_0 \end{bmatrix} - B_{2:}^T \begin{bmatrix} \boldsymbol{\phi}_{2:}(0) & \dot{\boldsymbol{\phi}}_{2:}(0) \end{bmatrix} \right) \begin{bmatrix} \phi_0(t_0) & \dot{\phi}_0(t_0) \\ \phi_1(t_0) & \dot{\phi}_1(t_0) \end{bmatrix}^{-1} \tag{5.22}$$

The lower triangular structure of $\phi(t)$ ensures that this 2×2 matrix is always invertible. This can be show easily since $\phi_0(t_0) = T_0(-1)$ is always 1, $\phi_1(t_0) = T_1(-1) = -1$, the derivative of T_0 is obviously 0 and $\dot{\phi}_1(t_0)$ is something non-zero.

Equation (5.2.1) allows us to express the first two rows of the coefficients matrix B as a linear combination of the rest $(N-1)-2$ rows. This means we don't need to "learn" them though optimisation, we instead minimize the error subject to the remaining rows.

The initialisation of B is in the users choice, it could be random matrix. Alternatively, one could use a coarse ODE solver, likely a forward Euler iteration with relatively large step size, to get a rough estimate $\boldsymbol{y}_c(t)$. Then by solving the least squares problem $\boldsymbol{y}_c(t) - \Phi B_0$ get an initialisation that's closer to the optimum, B^*, for faster convergence. One has to figure out a good balance between the number of steps of the coarse solver -which are innately serial- and how far the initial B is from the optimum, which will increase the number of iterations.

The Hessian

The primary concern computationally is the number of function evaluations or NFE of f, the neural networks. The number of coefficients in B is orders of magnitude smaller that the number of parameters in the network, hence storing the full Hessian shouldn't be a limiting factor.

Instead of directly inverting the Hessian to obtain the next B, we use a modified Cholesky decomposition (specifically of LDU form) as described in [26]. We use this technique in order to project the Hessian to a space of positive definite matrices in case it is not, which is a necessary condition for the Newton algorithm to converge. This way we can increase the definiteness of the Hessian while also solving the linear system $\boldsymbol{d}^{[k]} \leftarrow -\left[\nabla^2 e(B^{[k]})\right]^{-1} \nabla e(B^{[k]})$ as an LDU system (two triangular and one diagonal systems). To prevent confusion around the dimensionality of the Hessian $\nabla_B e$, the columns of B are stacked as a vector so that the hessian is a rectangular matrix instead of a $4D$ tensor.

5.0.3 Comparison with classical and other parellel-in-time methods

First of all, the parallelisation potential of such a method comes from the fact that f can be evaluated for all input points $\hat{y}(t)$ simultaneously at a single iteration. More specifically, during the fist, zero-th order part of the algorithm, a vector of points is passed into f to update the coeffecients as can be seen in (**??**). In reality, f is applied on each element of this vector but since the calculations are independent all elements

are evaluated in parallel. Similarly, during the Newton iteration the Hessian can, up to some degree, be computed in parallel by exploiting vectorization [CITE PYTORCH vmap?]. This is in contrast to classical ODE solvers where the solution, or the trajectory in state space, is calculated from initial time to terminal time, step by step; in our case the whole trajectory (a continuous time approximation of it rather) is known from the start. This is what allows to calculate the values -and the derivatives- along the entire trajectory in a parallel manner.

Obviously, more than one iteration is needed to obtain a solution to some specified accuracy. But if there are sufficiently many computing nodes available, each iteration should be at least comparable to a single step of many common ODE solvers. The critical point of out method is that it takes less parallel/batch evaluations of f to converge to a solution than steps of a time stepping solver.

For the second part of the algotithm the Jacobian and Hessian matrices of e also have to be calculated. Because of how automatic differentiation works the Jacobian is a byproduct of calculating the Hessian. Calculating the full Hessian is actually not of trivial cost and we will address this later. Additionally, the line search algorithm -for finding step sizes along the path of the Newton direction requires some evaluations of e and its Jacobian. Decreasing the number of function evaluations is crucial for further speeding up this algorithm. To this goal the cheaper recursive least squares solution is found first at significally less number of function evaluations and the Newton algorithm is used for refining the solution to highter accuracy.

It's important to note that traditional ODE numerical solver also take more that 1 (serial) function evaluations at each step. For example RK45, a widely used algorithm from the family of Runge-Kutta methods, involves six sequential function evaluations (NFE) [CITE?] at each step. Likewise, Dormand-Prince, the default method in many software packages [CITE?], sometimes referred as DOPRI5 also has an NFE of 6 for a single step.

Comparing popular parallel in time ODE solving methods, like PFASST and parareal, to our approach showcases similarities but also differences. One major difference is simplicity which allows easier implementation. Polynomial Approximation Numerical integration only builds upon simple polynomial approximation and a numerical optimiser. Other methods, like PFASST [23], rely on more complicated *spectral deferred correction* methods, a well established but more involved numerical method. Moreover, other parallel in time techniques, like parareal, split the initial value problem into N smaller boundary value problems using a combination of coarse and fine solvers [22].

One advantage of the method presented here is that it evaluates f on multiple inputs simultaneously, to minimise some error function. This *synchronised* parallelisation scheme resembles how multiples input samples are processed in batches during training of a neural network. This means that one can exploit GPUs' architecture for mass parallelisation using existing frameworks.

5.0.4 Shortcomings and improvements

As mentioned above the critical factor for the performance of neural ODEs is the number of function evaluations that have to be performed sequentially. While our experiments show that polynomial approximation integration can reduce NFE there are hidden costs involved with the calculating of the full Hessian and during line search.

Firstly, because the parameters B should be significantly less than the parameters of the network; calculating the full Hessian doesn't present problems as far as memory is concerned. But since Newton optimisation requires the inverse of the Hessian it is not possible to perform a fast Hessian-vector operation exploiting automatic differentiation properties. We instead calculate the full Hessian first, row by row, and then invert it.

Secondly, the line search algorithm involves computing the objective function and it's derivative at multiple states ($\boldsymbol{y}(t)$). While we hope that it shouldn't check more that two or three candidate values for the learning rate we can expect at least 4 more function evaluations that will be performed serialy [27].

These limitations of Newton optimisation with line search call for more numerical methods to be tested like Quasi-Newton, Conjugate Gradients and BFGS.

5.0.5 Experiments

We compare the performance of our method to popular time stepping methods in terms of accuracy and absolute error. Most implementations of variable step length methods like RK45 allow for absolute (atol) and relative (rtol) tolerances to be specified which keep the local error estimates bellow $atol + rtol|y|$. For our case, we define tolerances for both steps of the algorithm, least squares and Newton iteration in the following ways. For the first part, the algorithm stops when there is no sufficient change in the parameters: $\|B^{[k]} - B^{[k-1]}\| <$ tol_ls. While for the Newton part the iteration stops when the gradient is sufficiently close to zero: $\|\partial_k e\| <$ tol_newton. Adjusting these two *hyperparameters* as well as the number of approximation coefficients and the number of points where the error is calculated; we can change the convergence properties of the algorithm.

Spiral

First, we will examine a toy example of a nonlinear spiral in $\mathbb{R}^2$. We assume that the true solution is provided by the RK45 method with step size three orders of magnitude smaller than the interval. We assume this approximation to be good enough for out purpose and compare the norm of the error at terminal time with other alongside our approach.

$$\frac{d\boldsymbol{y}}{dt} = \tanh(\begin{bmatrix} a & 1 \\ -1 & -a \end{bmatrix} \boldsymbol{y})$$

We solve the ODE in the interval $[0, 10]$ for $\boldsymbol{y}(0) = [2,\ 1]^T$. We use the same tolerances for the time stepping methods and for ours (PAN) even thought there is not direct equivalence. This shouldn't matter as we are ultimately interested only in the comparison between accuracy and number of function evaluations (NFE). For the other solvers the relative error is also set to a small number so that the absolute error is unaffected.

Table 5.1: Metrics for the solution of an IVP with different methods. For PAN params indicates the number of approximation coefficients per dimention, in this case there are 2 dimensions. NFE for PAN is show as NFE for least squares plus (+) NFE for Newton.

Method(params)	tolerance	error(t)	NFE
RK45	1.0e-06	4.14e-07	212
Radau	1.0e-06	6.97e-08	599
BDF	1.0e-06	4.56e-06	250
LSODA	1.0e-06	4.34e-06	171
PAN(50)	1e-06/1e-06	1.86e-10	37 + 5
PAN(100)	1e-06/1e-06	4.17e-14	36 + 5
RK45	1.0e-09	6.28e-10	716
Radau	1.0e-09	1.05e-10	2994
BDF	1.0e-09	1.66e-08	656
LSODA	1.0e-09	4.48e-10	361
PAN(50)	1e-09/1e-09	1.86e-10	42 + 5
PAN(100)	1e-09/1e-09	2.25e-15	55 + 5

We remind that for PAN, during optimisation f is evaluated on multiple points, but since this calculations can be performed in parallel we assume the wall clock time is similar to one evaluation plus parallelisation overhead. As we can see in table 1 for this simple problem our method achieves better accuracy while taking significantly less number of function evaluation.

Lorenz Attractor

The Lorenz system is a system of ordinary differential equations known for its chaotic behaviour.

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x)\\ \frac{dy}{dt} &= x(\rho - z) - y\\ \frac{dz}{dt} &= xy - \beta z\end{aligned}$$

We again solve the equation using a variety of solvers and compare them with our method. Though, this time we use "step" for our method, meaning we apply the algorithm in smaller sub-intervals setting the endpoint of each sub-solution as the initial conditions for the next interval. The problem parameters are set to: $\rho = 28,\ \sigma = 10,\ \beta = \frac{8}{3}$, the interval is $[0, 15]$ and $\boldsymbol{y}(0) = [2, 1, 1]^T$.

In table, 2 we see that our method achieves better accuracy for less amount of function evaluation on this system of ODEs. Although, we have to note that accuracy seems to become saturated and decreasing tolerances or increasing number of coefficients doesn't improve results. It becomes apparent that finding the optimal hyperparameters is crucial for the most efficient use of the algorithm.

Table 5.2: Accuracy and NFE for Lorenz system. For PAN the interval has been split in subintervals of length 0.3 and the algorithm has been applied sequencially on each interval.

Method(params)	tolerance	error(t)	NFE
RK45	1.0e-06	5.75e-02	5174
Radau	1.0e-06	2.17e-02	16476
BDF	1.0e-06	2.09e-01	4794
LSODA	1.0e-06	4.86e-02	2803
PAN(50)	1e-06/1e-06	1.60e-07	1180 + 250
PAN(100)	1e-06/1e-06	1.60e-07	1231 + 250
RK45	1.0e-09	6.58e-05	19532
Radau	1.0e-09	4.19e-06	89999
BDF	1.0e-09	5.65e-04	15184
LSODA	1.0e-09	5.45e-06	4735
PAN(50)	1e-09/1e-09	1.61e-07	1399 + 262
PAN(100)	1e-09/1e-09	1.61e-07	1450 + 390

Bibliography

[1] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pmlr, 2015, pp. 448–456.

[5] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *arXiv preprint arXiv:1505.00387*, 2015.

[6] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[7] R. Rico-Martinez, K. Krischer, I. Kevrekidis, M. Kube, and J. Hudson, "Discrete-vs. continuous-time nonlinear signal processing of cu electrodissolution data," *Chemical Engineering Communications*, vol. 118, no. 1, pp. 25–48, 1992.

[8] E. Weinan, J. Han, and Q. Li, "A mean-field optimal control formulation of deep learning," *arXiv preprint arXiv:1807.01083*, 2018.

[9] Y. Lu, A. Zhong, Q. Li, and B. Dong, "Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations," in *International Conference on Machine Learning*, PMLR, 2018, pp. 3276–3285.

[10] L. Ruthotto and E. Haber, "Deep neural networks motivated by partial differential equations," *Journal of Mathematical Imaging and Vision*, vol. 62, no. 3, pp. 352–364, 2020.

[11] P. Kidger, "On neural differential equations," *arXiv preprint arXiv:2202.02435*, 2022.

[12] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama, "Dissecting neural odes," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3952–3963, 2020.

[13] X. Chen, "Ordinary differential equations for deep learning," *arXiv preprint arXiv:1911.00502*, 2019.

[14] L. S. Pontryagin, *Mathematical theory of optimal processes*. Routledge, 2018.

[15] J. Chiu, S. Duffield, M. Hunter-Gordon, K. Donatella, M. Aifer, and A. Gu, "Exploiting inductive biases in video modeling through neural cdes," *arXiv preprint arXiv:2311.04986*, 2023.

[16] M. Poli, S. Massaroli, A. Yamashita, H. Asama, and J. Park, "Hypersolvers: Toward fast continuous-depth models," *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 105–21 117, 2020.

[17] P. Kidger, J. Foster, X. C. Li, and T. Lyons, "Efficient and accurate gradients for neural sdes," *Advances in Neural Information Processing Systems*, vol. 34, pp. 18 747–18 761, 2021.

[18] J. Zhuang, N. C. Dvornek, S. Tatikonda, and J. S. Duncan, *Mali: A memory efficient and reverse accurate integrator for neural odes*, 2021. arXiv: `2102.04668 [cs.LG]`.

[19] E. Süli, "Numerical solution of ordinary differential equations," *Mathematical Institute, University of Oxford*, 2010.

[20] J. C. Butcher, *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[21] J. Nievergelt, "Parallel methods for integrating ordinary differential equations," *Communications of the ACM*, vol. 7, no. 12, pp. 731–733, 1964.

[22] Y. Maday and G. Turinici, "A parareal in time procedure for the control of partial differential equations," *Comptes Rendus Mathematique*, vol. 335, no. 4, pp. 387–392, 2002.

[23] M. Emmett and M. Minion, "Toward an efficient parallel in time method for partial differential equations," *Communications in Applied Mathematics and Computational Science*, vol. 7, no. 1, pp. 105–132, 2012.

[24] S. Massaroli, M. Poli, S. Sonoda, *et al.*, "Differentiable multiple shooting layers," *Advances in Neural Information Processing Systems*, vol. 34, pp. 16 532–16 544, 2021.

[25] L. N. Trefethen, "Finite difference and spectral methods for ordinary and partial differential equations," 1996.

[26] P. E. Gill, W. Murray, and M. H. Wright, *Practical optimization*. SIAM, 2019.

[27] S. J. Wright, *Numerical optimization*. 2006.

[28] S. H. Cheng and N. J. Higham, "A modified cholesky algorithm based on a symmetric indefinite factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 19, no. 4, pp. 1097–1110, 1998.

[29] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, "Deep double descent: Where bigger models and more data hurt. arxiv," *arXiv preprint arXiv:1912.02292*, 2019.

[30] K. C. Kiwiel, "Convergence and efficiency of subgradient methods for quasiconvex minimization," *Mathematical programming*, vol. 90, pp. 1–25, 2001.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[32] R. Kashyap, "A survey of deep learning optimizers–first and second order methods," *arXiv preprint arXiv:2211.15596*, 2022.

[33] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, 2011.

.1 The backpropagation equations

In order to contrast and highlight the differences between classical neural networks, residual neural networks and neural ODEs we showcase how the gradients of the loss wrt. the learnable parameters is calculated in each architecture. This quantity is necessary in order to minimise the loss of the network using any -gradient descent based- optimisation algorithm.

.1.1 Vanilla Neural Networks

Classical artificial neural networks are comprised of linear transformations followed by non-linear activation functions. The equation that describes the input-output relationship of the n-th layer is:

$$\boldsymbol{y}_n = \phi(W_n \boldsymbol{y}_{n-1}) \tag{23}$$

Where ϕ is a non-linear function, W_n is the layer's weights, $\boldsymbol{y}_{n-1}$ the output of the previous layer, we omit the biases vector to simplify the equations but the logic is the same. Unfolding this equation for the whole network of depth N we get:

$$\boldsymbol{y}_{out} = \phi(W_N \cdot \phi(W_{N-1} \cdot \phi(W_{N-2}(\dots)))) \tag{24}$$

The output of the network is passed in a (scalar) cost function $C(\boldsymbol{y}_{out}, \boldsymbol{y}^*)$. In order to apply gradient descent based optimisation algorithms it is necessary to calculate the gradient wrt. all the weights:

$$\frac{\partial C}{\partial W_n}, \quad \text{for } n = 1, 2, \ldots, l \tag{25}$$

Focusing on a single layer, lets use $\mathbf{z}_n$ to denote $W_n \cdot \mathbf{y}_{n-1}$ or whats goes in the non-linear function at each layer. By applying the chain rule we get:

$$\frac{\partial C}{\partial \mathbf{y}_{n-1}} = \left[\frac{\partial \mathbf{y}_n}{\partial \mathbf{y}_{n-1}}\right]^T \frac{\partial C}{\partial \mathbf{y}_n} = \left[\frac{\partial}{\partial \mathbf{y}_{n-1}} \phi(W_n \mathbf{y}_{n-1})\right]^T \frac{\partial C}{\partial \mathbf{y}_n} \tag{26}$$

$$= \left[W_n^T \phi^{(1)}(\boldsymbol{z}_n)\right]^T \frac{\partial C}{\partial \mathbf{y}_n} \tag{27}$$

$$= \left[\phi^{(1)}(\boldsymbol{z}_n)\right]^T W_n \frac{\partial C}{\partial \mathbf{y}_n} \tag{28}$$

The non-linearity ϕ is applied on each element of the input vector independently, so its partial derivative wrt. the input its a diagonal matrix and its transpose is the same as itself.

$$\frac{\partial C}{\partial \mathbf{y}_{n-1}} = \phi^{(1)}(\boldsymbol{z}_n) W_n \frac{\partial C}{\partial \mathbf{y}_n} \tag{29}$$

Equation (48) allows to calculate all such derivatives recursively starting from the last layer where $\frac{\partial C}{\partial \mathbf{y}_{out}}$ has a closed form solution. We can now calculate the desired gradients with respect to the weights. Below W_n^i is the i-th column of the weights matrix of the n-th layer.

$$\begin{aligned}
\frac{\partial C}{\partial W_n^{[i]}} = \left[\frac{\partial \boldsymbol{y}_n}{\partial W_n^i}\right]^T \frac{\partial C}{\partial \boldsymbol{y}_n} &= \left[\frac{\partial}{\partial W_n^{[i]}} \phi(W_n \boldsymbol{y}_{n-1})\right]^T \frac{\partial C}{\partial \boldsymbol{y}_n} \\
&= \left[\frac{\partial}{\partial W_n^{[i]}} \phi(W_n^{[1]} \boldsymbol{y}_{n-1}^{[1]} + \cdots + W_n^{[i]} \boldsymbol{y}_{n-1}^{[i]} + \ldots)\right]^T \frac{\partial C}{\partial \boldsymbol{y}_n} \\
&= \left[\left[\boldsymbol{y}_{n-1}^{[i]}\right]^T \phi^{(1)}(\boldsymbol{z}_n)\right]^T \frac{\partial C}{\partial \boldsymbol{y}_n} \\
&= \phi^{(1)}(\boldsymbol{z}_n)\, \boldsymbol{y}_{n-1}^{[i]} \frac{\partial C}{\partial \boldsymbol{y}_n}
\end{aligned}$$

Stacking all the columns together we get:

$$\frac{\partial C}{\partial W_n} = \phi^{(1)}(\boldsymbol{z}_n) \frac{\partial C}{\partial \boldsymbol{y}_n} \boldsymbol{y}_{n-1} \tag{30}$$

Algorithm 2 Backpropagation

Do the forward pass, save in memory $W_n, \mathbf{z}_n, \mathbf{y}_n$
Calculate $\frac{\partial C}{\partial \mathbf{y}_{out}}$ which is trivial
for $n = N-1, \ldots, 1$ **do**
  Using (48) and $\frac{\partial C}{\partial \mathbf{y}_{n+1}}$ find $\frac{\partial C}{\partial \mathbf{y}_n}$
  Using (49) and $\frac{\partial C}{\partial \mathbf{y}_n}$ find $\frac{\partial C}{\partial W_n}$
end for

Using (48) and (49) together we can find all the necessary gradients for backpropagation; as expressed in algorithm 2

The backpropagation algorithm is quite fast in modern hardware as it consists of mostly matrix vector multiplications. Although, it is memory intensive since it requires saving all the weights and activations from the forward pass and accessing them during the backward pass.

.1.2 Residual Networks

Residual neural networks operate quire similarly to classical ones in terms of training.

$$\boldsymbol{y}_n = \phi(W_n \boldsymbol{y}_{n-1}) + \boldsymbol{y}_{n-1} \tag{31}$$

$$\frac{\partial C}{\partial \mathbf{y}_{n-1}} = \left[\frac{\partial \mathbf{y}_n}{\partial \mathbf{y}_{n-1}}\right]^T \frac{\partial C}{\partial \mathbf{y}_n} = \left\{\frac{\partial}{\partial \mathbf{y}_{n-1}}\left[\phi(W_n \mathbf{y}_{n-1}) + \mathbf{y}_{n-1}\right]\right\}^T \frac{\partial C}{\partial \mathbf{y}_n} \tag{32}$$

$$= \left[W_n^T \phi^{(1)}(\boldsymbol{z}_n) + I\right]^T \frac{\partial C}{\partial \mathbf{y}_n} \tag{33}$$

Equation (51) is again a recursive relation that can be used to find all intermediate "states " starting from the output layer. The gradients wrt. weights becomes:

$$\frac{\partial C}{\partial W_n^{[i]}} = \left[\frac{\partial \boldsymbol{y}_n}{\partial W_n^i}\right]^T \frac{\partial C}{\partial \boldsymbol{y}_n} = \left\{\frac{\partial}{\partial W_n^{[i]}}\left[\phi(W_n \boldsymbol{y}_{n-1}) + \mathbf{y}_{n-1}\right]\right\}^T \frac{\partial C}{\partial \boldsymbol{y}_n} \tag{34}$$

Since the output of layer $n-1$ doesn't depend on the weights of layer n, (53) becomes identical to (49) and the same backpropagation algorithm is used to train the network.

> Backpropagation can be seen as a specific of Automatic Differentiation, a very powerful framework to acquire gradients numerically. The governing principle is the same, propagating the gradients through functions with closed form derivative, utilising the chain rule.

.1.3 Neural ODEs

Calculating $\nabla_\theta L$ in neural ODEs can be achieved by differentiating through the solver steps but as we will demonstrate there is a much more efficient way, using tools from optimal control theory, namely adjoint sensitivities. The governing equations for neural ODEs are (3), (8), (9) we repeat them for convenience:

$$\boldsymbol{y}(T) = \boldsymbol{y}(0) + \int_0^T f(\boldsymbol{y}(\tau), \tau, \boldsymbol{\theta})\, d\tau, \quad \boldsymbol{y}(0) = \boldsymbol{y}_0$$

$$\frac{d\boldsymbol{a}(t)}{dt} = -\boldsymbol{a}(t)^T \frac{\partial f(\boldsymbol{y}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{y}}, \quad \boldsymbol{a}(T) = \frac{\partial L}{\partial \boldsymbol{y}(T)}$$

$$\nabla_{\boldsymbol{\theta}} L = -\int_T^0 \boldsymbol{a}(t)^T \frac{\partial f(\boldsymbol{y}(\tau), \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\, d\tau$$

Calculating the last integral numerically requires the values of $\boldsymbol{a}(t)$ at all the points in time the solver chooses. In the case of a fixed step size solver with step h those would be $t_m = T - mh,\ m = 0, 1, \ldots$. In the case of an adaptive step size t_m could be any point between T and 0. In both cases we can calculate those quantities by solving the second ODE starting from time $t = T$ -where $a(T)$ has a closed form solution- and moving backwards in time. Next we need the value of $\boldsymbol{y}$ at the same t_ms. Again we can calculate them starting from $\boldsymbol{y}(T)$ and solving the original IVP again backwards in time. Notice that we don't need to save any intermediate state or activations from the forward pass since we can re-calculate them moving backwards. In this sense, neural ODEs are reversible (up to numerical tolerances).

In algorithm 3 we reformulate the integral equation for the gradient of the loss wrt. weights as a differential equation: $\frac{d}{dt}\boldsymbol{a}_\theta(t) = -\boldsymbol{a}(t)^T \frac{\partial f}{\partial \boldsymbol{\theta}}$. We know that $\boldsymbol{a}_\theta(T) = 0$ and we search for $\boldsymbol{a}_\theta(0) = \nabla_\theta L$.

.2 An alternative derivation for $\frac{dL}{d\boldsymbol{\theta}}$

There are already many ways in the literature for finding this quantity without delving into control theory and reverse sensitivities. Intuitively in can be thought as a continuous analogous to classical backpropagation.

$$\text{classical} \rightarrow \frac{dL}{d\boldsymbol{\theta}} = \sum_k \frac{\partial L}{\partial \boldsymbol{y}_k} \frac{\partial f(\boldsymbol{y}_k, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{35}$$

$$\text{adjoint sensitivities} \rightarrow \frac{dL}{d\boldsymbol{\theta}} = \int_T^0 \frac{\partial L}{\partial \boldsymbol{y}(\tau)} \frac{\partial f(\boldsymbol{y}(\tau), \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\, d\tau \tag{36}$$

Algorithm 3 Adjoint

Choose ODE solver hyperparameters (h, ...)
Do the forward pass, find $\boldsymbol{y}(T)$
$\boldsymbol{a}(T) \leftarrow \frac{\partial L}{\partial \boldsymbol{y}(T)}$ (closed form)
$\boldsymbol{a}_\theta(T) \leftarrow 0$
$t \leftarrow T$
while $t > 0$ **do**
  $\boldsymbol{y}(t - \Delta t) \leftarrow \text{IntStep}(\boldsymbol{y}(t), f)$
  $\boldsymbol{a}(t - \Delta t) \leftarrow \text{IntStep}(\boldsymbol{a}(t), -\boldsymbol{a}(t)\frac{\partial f}{\partial \boldsymbol{y}})$
  $\boldsymbol{a}_\theta(t - \Delta t) \leftarrow \text{IntStep}(\boldsymbol{a}_\theta(t), -\boldsymbol{a}(t)\frac{\partial f}{\partial \boldsymbol{\theta}})$
  $t \leftarrow t - \Delta t$
end while
return $\boldsymbol{a}_\theta(0) = \nabla_\theta L$

In the original neural ODEs paper [1] the authors prove (8) and use an augmented state to prove (9) while others like [11] provide alternative proofs. We present another, in our opinion much simpler, derivation for (9). Consider the more general case where the loss is dependant on intermediate state on a point $s \in (0, T)$. This is equivalent to the usual case where the loss depends only on the output state with $\mathcal{L}(s) = L(\boldsymbol{y}(s)) = 0$ for $s \neq T$. Moreover, the state $\boldsymbol{y}$ is a also dependant on the weights even though it's not explicitly written.

$$
\begin{aligned}
\frac{dL(\boldsymbol{y}(s, \boldsymbol{\theta}))}{d\boldsymbol{\theta}} &= \int_0^s \frac{d}{dt}\left(\frac{dL(\boldsymbol{y}(s, \boldsymbol{\theta})}{d\boldsymbol{\theta}}\right) dt \\
&= \int_0^s \frac{d}{dt}\left(\frac{dL(\boldsymbol{y}(t, \boldsymbol{\theta})}{d\boldsymbol{y}(t)}\frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}}\right) dt \\
&= \int_0^s \frac{d}{dt}\frac{dL(\boldsymbol{y}(t, \boldsymbol{\theta})}{d\boldsymbol{y}(t)} \cdot \frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}} + \frac{dL(\boldsymbol{y}(t, \boldsymbol{\theta})}{d\boldsymbol{y}(t)} \cdot \frac{d}{dt}\frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}} dt \\
&= \int_0^s \dot{\boldsymbol{a}}(t)\frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}} + \boldsymbol{a}(t) \cdot \frac{d}{d\boldsymbol{\theta}}\frac{d\boldsymbol{y}(t)}{dt} dt \\
&= \int_0^s \dot{\boldsymbol{a}}(t)\frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}} + \boldsymbol{a}(t) \cdot \frac{d\boldsymbol{f}}{d\theta} dt \\
&= \int_0^s \dot{\boldsymbol{a}}(t)\frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}} + \boldsymbol{a}(t)\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}(t)}\frac{\partial \boldsymbol{y}(t)}{\partial \boldsymbol{\theta}}\right) dt \\
&= \int_0^s \frac{d\boldsymbol{y}(t)}{d\boldsymbol{\theta}}\left(\dot{\boldsymbol{a}}(t) + a\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}(t)}\right) + \boldsymbol{a}(t)\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}} dt
\end{aligned}
$$

From (8) the sum in the parenthesis is 0:

$$\frac{dL(\boldsymbol{y}(s,\boldsymbol{\theta}))}{d\boldsymbol{\theta}} = \int_0^s \boldsymbol{a}(t)\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}dt \tag{37}$$

Setting $s = T$ we arrive at the desired formula.

.3 Vector-Jacobian products

Modern software packages utilise automatic differentiation methods to calculate derivatives of some arbitrary function f, usually with respect to some parameters or weights θ.

$$\frac{\partial f(x;\theta)}{\partial \theta} \tag{38}$$

.4 Optimisation

In the field of numerical optimisation many methods have been developed for minimising a *scalar* objective function f. One category of those, *line search* methods, are iterative algorithms that seek to update the current iterate x_k to a new value closer to the minimum. They work by choosing a direction p_k -and step size α- along which f is decreased. Alternatively the problem can be restated as:

$$\min_{a>0} f(x_k + \alpha p_k) \tag{39}$$

While ideally we would solve (58) exactly, in practice it is computationally expensive and practically unnecessary. Line search implementations usually generate some trial step sizes until they find one that satisfies certain termination conditions we will examine later.

A *descent direction* is a direction that causes f to decrease along it given sufficiently small $\alpha > 0$: $f(x_k + \alpha p_k) < f(x_k)$.We can show that if p_k is a descent direction then $p_k^T \nabla f_k < 0$. From the first order Taylor series expansion we have:

$$f(x_k + ap_k) = f(x_k) + ap_k^T \nabla f_k + O(a^2) < f(x_k)$$

Ignoring quadratic term since α is small.

$$ap_k^T \nabla f_k < 0$$
$$p_k^T \nabla f_k < 0$$

It can be proven though Zoutendijk's theorem that: as long as the search direction is a descent direction and the step size fulfils some conditions the algorithm is globally convergent.

The defining characteristic of a line search method is the way in which we obtain the search direction p_k. An obvious choice is to use the direction of *steepest descent*, mathematically obtained as the negative of the gradient at the current iterate.

$$x_{k+1} = x_k - \alpha \nabla f_k \tag{40}$$

In machine learning literature steepest descent, or as they are more commonly refereed, *gradient descent* methods are extremely common. Almost all neural network models are trained using some derivative of classic gradient descent. These methods are not categorised as line searches since they do not seek to find an appropriate step length α on each iteration. They instead define either a fixed or dynamically updated *learning rate*. Usually they employ some notions of momentum and stochasticity to move along a direction dictated by an approximation of the local gradient. Even so they remain first order methods since they only utilise information about the first derivative. Some notable gradient descent methods as well as how the minimisation problem is formulated in the context of machine learning are discussed later.

.4.1 The Newton-Raphson algorithm with Hessian modification

In more general minimisation problems a prevalent search direction for line search optimisers is the *Newton direction*, methods using this direction are called Newton methods. This direction is derived from the second-order Taylor series approximation of the objective function, near the current iterate x_k. Consider a continuous, twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}^n$, its second order Taylor series approximation is:

$$f(x_k + p) = f_k + p^t \nabla f_k + \frac{1}{2} p^t \nabla^2 f_k p + R(p) \tag{41}$$

with $R(p)$ being of order $O(\|p\|^3)$. For small values of p, the residual term diminishes and we have a pretty good quadratic model of f. Seeking to minimise this model we set the derivative wrt. p equal to 0:

$$\nabla f_k + \nabla^2 f_k p_t = 0 \tag{42}$$

$$p = -\left[\nabla^2 f_k\right]^{-1} \nabla f_k \tag{43}$$

Equation (62) gives the definition of the newton direction. As mentioned before, in order for the algorithm to be globally convergent the search direction has to be a descent direction. By multiplying (61) from the left with p^t we get:

$$p^t \nabla f_k + p_t \nabla^2 f_k p_t = 0 \tag{44}$$

$$p^t \nabla^2 f_k p_t = - p^t \nabla f_k < 0 \tag{45}$$

$$p^t \nabla^2 f_k p_t > 0 \tag{46}$$

From (65) it is apparent that the Hessian matrix $H_k = \nabla^2 f_k$ has to be positive definite which is generally true if x_k is near the minimum but not necessarily true away from it. If the Hessian is not positive definite there is no guarantee that (62) gives a descent direction or even that the Hessian is non-singular. In order to address this issue a positive definite approximation of the Hessian is used. The approximation can be obtained in several ways, some involve adding a multiple of the identity or some correction matrix ΔH, others modify the eigenvalues of the true Hessian directly.

For example in [28] is shown that, if H_k has spectral decomposition $H_k = Q\Lambda Q^T$ then the correction matrix of minimum Frobenius norm that ensures that the smallest eigenvalue of $H_k + \Delta H$ is larger or equal to δ is given by:

$$\Delta H = Q(diag)\tau_i Q^T, \quad \text{with} \quad \tau_i = \begin{cases} 0, & \lambda_i \geq \delta, \\ \delta - \lambda_i, & \lambda_i < \delta, \end{cases}$$

Another technique, the one used here, is to perform (or try to perform) a Cholesky decomposition, more specifically an LDL decomposition of the Hessian. Since the Hessian is not always positive definite the factorisation $H = LDL^T$ may not exist or if even if it does the algorithm used to compute it is numerically unstable. The core idea of modified Cholesky decomposition is to modify the values of the diagonal matrix D to be sufficiently positive while the factorisation is computed.

More specifically, [27] provide an algorithm for computing the LDL^T factorisation a positive definite approximation of a matrix. It accepts two additional parameters δ, β so that the following bounds are satisfied.

$$d_j \geq \delta, \quad |m_{ij}| \leq \beta, \quad i = j+1, j_2, \ldots, n \tag{47}$$

The algorithm calculates the elements of the diagonal D and the unitriangular matrix L column by column. It is a simpler version of the algorithm presented by [26] that additionally introduces symmetric row-column interchanges resulting in lower $\|H_k - \hat{H}_k\|$, $\hat{H}_k$ being the approximation.

After addressing the definiteness of the the Hessian let's describe how an algorithm would decide on the step size. As noted earlier Zoutendijk's theorem requires the search direction to be a descent direction as well as the step size to fulfil a certain set of conditions. One can prove Zoutedijk's theorem for multiple sets of conditions, namely the Wolfe, strong Wolfe and Goldstein conditions. We will focus on the strong Wolfe conditions.

Assume we have chose direction p_k at step k of the algorithm, our second objective is to find step size a to proceed to the next step without minimising $\phi(\alpha) = f(x_k + \alpha p_k)$ explicitly. The first strong Wolfe condition states that our guess for a should give *sufficient decrease* in the objective function in the following way:

$$f(x_k + ap_k) \leq f(x_k) + c_1 a \nabla f_k^T p_k \tag{48}$$

Algorithm 4 Modified Cholesky

for $i = j+1, \ldots, n$ **do**
  $c_{jj} \leftarrow a_{jj} - \sum_{s=1}^{j-1} d_s l_{js}$
  $\theta_j \leftarrow \max_{j<i\leq n}(|c_{ij}|)$
  $d_j \leftarrow \max\left(|c_{jj}|, \left(\frac{\theta_j}{\beta}\right)^2, \delta\right)$
  for $i = j+1, \ldots, n$ **do**
    $c_{ij} \leftarrow a_{ij} - \sum_{s=1}^{j-1} d_s l_{is} l_{js}$
    $l_{ij} \leftarrow c_{ij}/d_j$
  end for
end for

for some $c_1 \in (0,1)$ usually chosen to be quite small ($\approx 10^{-4}$). This ensures the reduction in f is proportional to the step size as well as the directional derivative $\nabla f^T p_k$. The sufficient decrease condition is not enough to make reasonable progress since it's satisfied for values of α close to 0. To prevent this, the second strong Wolfe condition or *curvature condition* states that α should also satisfy:

$$|\nabla f(x_k + \alpha p_k)^T p_k| \leq |c_2 \nabla f_k^T p_k| \tag{49}$$

for some constant $c_2 \in (c_1, 1)$ usually set around 0.9 for Newton direction. Notice that the left hand side if the derivative wrt. α of $\phi(a)$ at a_k and the right hand size at 0. Practically the curvature condition requires the slope of ϕ at the new point to be greater that at the start times c_2. Remember that p_k is a descent direction so $\phi(0)$ is strictly negative. Larger slope would mean we get closer to a stationary point of zero gradient. The absolute ensures that the slope doesn't become too positive and we don't overshoot away from the stationary point.

Utilising those conditions we construct the following line search algorithm parameterised by c_1, c_2, α_{max}

The procedure uses the knowledge that the interval (a_{i-1}, a_i) contains step lengths satisfying the strong Wolfe conditions if one of the following three conditions is satisfied:

1. a_i violates the sufficient decreasee condition
2. $\phi(a_i) \geq \phi(a_{i-1})$
3. $\phi'(a_i) \geq 0$

The last step of the algorithm performs extrapolation to find the next trial value a_{i+1}. We can use some more involved extrapolation technique like the ones mentioned previously in the section, or we can simply set a_{i+1} to some constant multiple of a_i. Whichever strategy we use, it is important that the successive steps increase quickly enough to reach the upper limit $a_{\max}$ in a finite number of iterations.

Algorithm 5 Line search

Set $a_0 \leftarrow 0$, choose $a_1 \in (0, a_{max})$
$i \leftarrow 1$
repeat
  if $\phi(a_i) > \phi(0) + c_1 a_i \phi'(0)$ or $[\phi(a_i) \geq \phi(a_{i-1})$ and $i > 1]$ **then**
    return **zoom**(a_{i-1}, a_i)
  end if
  if $|\phi'(a_i)| \leq -c_2 \phi'(0)$ **then**
    return a_i
  end if
  if $\phi'(a_i) \geq 0$ **then**
    return **zoom**(a_i, a_{i-1})
  end if
  $a_{i+1} \leftarrow \min(a_m ax, 2 * a_i)$
  $i \leftarrow i + 1$
until

We know specify the function zoom, which requires a little explanation. The order of its input is such that each call has the form: **zoom**(a_{lo}, a_{hi}), where

1. the interval bounded by a_{lo} and a_{hi} contains step lengths that satisfy the strong Wolfe conditions
2. a_{lo} is, among all step lengths generated so far and satisfying the sufficient decrease conditions, the one giving the smallest value;
3. $a_h i$ is chosen so that $\phi a_{lo}(a_{hi} - a_{lo}) < 0$

Each iteration of zoom generates and iterate a_j between a_{lo} and a_{hi}, and then replaces one of these endpoints by a_j in such a way that the above properties continue to hold.

If the new estimate a_j happens to satisfy the strong Wolfe conditions, then zoom has served its purpose of identifying such a point, so it terminates with $a^* = a_j$. Otherwise, if a_j satisfies the sufficient decrease condition and has a lower function value than a_{lo} , then we set $a_{lo} \leftarrow a_j$ to maintain condition (2). If by doing so we end up violating condition (3), we set a_{hi} to the old value of a_{lo} ensuring condition (3) holds.

Several implementations of the Newton algorithm with line search exist in scientific software packages. We have written an exact Newton routine in **PyTorch** utilising automatic differentiation and GPU parallelism.

.4.2 Gradient Descent Methods

For the sake of completeness we will present some fist order optimisation methods. Gradient descent and its derivative are by far the most used optimizers in machine

Algorithm 6 zoom

repeat
  Interpolate to find a trial step length a_j between a_{lo} and a_{hi}
  Evaluate $\phi(a_j)$
  if $\phi(a_j) > \phi(0) + c_1 a_j \phi'(0)$ or $\phi(a_j) \geq \phi(a_{lo}]$ **then**
    $a_{hi} \leftarrow a_j$
  else
    Evaluate $\phi'(a_j)$
    if $|\phi'(a_j)| \leq -c_2 \phi'(0)$ **then**
      Set $a^* \leftarrow a_j$ and **stop**
    end if
    if $\phi'(a_j)(a_{hi} - a_{lo}) \geq 0$ **then**
      $a_{hi} \leftarrow a_{lo}$
    end if
    $a_{lo} \leftarrow a_{hi}$
  end if
until

learning. For really high dimensional problems, such as those we are concerned with in modern deep learning settings, second order methods introduce quadratically scaled (on the number of parameters) computation and memory cost. This cost is related to the computation and storage of the Hessian (or an approximation of it) which is of size $num_parameters^2$. The tradeoff between number of iterations and cost per iteration leans in favor of gradient descent optimizers as the number of parameters increases, especially considering the fact that they lend themselves well to automatic differentiation and distributing training. The loss function for example All these in combination with the higher implementation complexity have lead to the dominance of first order optimisation techniques in machine learning.

Stochastic Gradient Descent

Optimising a deep learning model in the strict sense means minimising the expected generalization error $\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\text{data}}} L(f(\boldsymbol{x},\boldsymbol{\theta}),\boldsymbol{y})$ where p_{data} is the joint distribution of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$, $\boldsymbol{\theta}$ are the models parameters and L some per sample cost function. Since the distribution of data is largely unknown we create an empirical estimate from a set of training examples $\boldsymbol{x}_i, \boldsymbol{y}_i{}_{i=1}^N$

$$\tilde{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(f(\boldsymbol{x}_i;\theta),\boldsymbol{y}_i)$$

Even though this approach is theoretically prone to overfitting, since models with high number of parameters and resultantly high capacity can simply memorise all training examples, without the ability to generalize outside the training set; empirical evidence show that models with large number of parameters achieve state-of-the-art performance in many tasks [29]. Minimising such a loss function though traditional gradient descent would require to evaluate the cost for all samples in the training set on each iteration to obtain the loss.

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} + \eta \nabla_{\theta_k} L(\boldsymbol{\theta}_k)$$

To alleviate some computational cost on each iteration Stochastic Gradient Descent (SGD) is used which instead of calculating the loss for all samples in the training set it samples a subset of them and performs the weights updated based on that. This approach introduces some stochasticity to the gradient approximation which leads to noise increasing the convergence rate but can lower the overall computational cost. To keep the the algorithm to diverging due to noise a lower learning rate is required to maintain stability [30]. Furthermore, the stochastic nature of SGD can allow it to escape from local minima in the non convex high-dimensional loss space that a classic gradient descent scheme would be trapped in.

To accelerate Stochastic Gradient Descent a number of methods have been proposed some of them in the recent years due to the popularity of deep neural networks.

Momentum

Based on the idea of physical momentum in real world objects the momentum method was proposed in the context of machine learning by [31] borrowing the idea from previous work by Polyak. This algorithm considers previous values of the gradient in the update step in a exponentially decaying manner. The update along dimensions that keep moving in the same direction keeps getting bigger and for dimensions that change direction the update gets smaller. Even though it can be applied to both stochastic and classical gradient descent, SGD with a momentum term has been the gold standard for many years for non-convex optimisation problems, and presents very good results. It converges faster that classical GD due to accumulated velocity while also dampening oscillations in regions of high curvatures by combining gradients of opposite signs. SGD Momentum can be expressed with the following equation

$$\begin{aligned} \Delta\boldsymbol{\theta}_{k+1} &\leftarrow \alpha \Delta\boldsymbol{\theta}_k - \eta \nabla_{\theta_k} L(\boldsymbol{\theta}_k) \\ \boldsymbol{\theta}_{k+1} &\leftarrow \boldsymbol{\theta}_k + \Delta\boldsymbol{\theta}_{k+1} \end{aligned}$$

where α is an exponential decay factor between 0 and 1 that determines the amount of contribution previous gradients have to the weights update.

Nesterov Momentum

A major disadvantage of classical momentum (CM) method is its tendency to overshoot as is approaches the minimum. Nesterov momentum seeks to solve this problem drawing from Nesterov's accelerated gradient method for convex functions. The equations for Nesterov's momemtum are:

$$\Delta\boldsymbol{\theta}_{k+1} \leftarrow \alpha\Delta\boldsymbol{\theta}_k - \eta\nabla_{\theta_k}L(\boldsymbol{\theta}_k + \alpha\Delta\boldsymbol{\theta}k)$$
$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \Delta\boldsymbol{\theta}_{k+1}$$

This formula resembles CM but instead of calculating the gradient of loss at the current $\boldsymbol{\theta}$ it is calculated after the previous velocity is added. This method is based on a corrector-predictor scheme where you first make a big step on the direction of the accumulated gradient and then make a correction based on the gradient on the point you land. This behaviour helps the method avoid oscillations, such as those that appear in CM around high-curvature regions, by pointing the gradients back to $\boldsymbol{\theta}_k$ more effectively than classical momentum [32].

Adagrad

The Adagrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the the sum of the historical squared values of the gradient.

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \frac{\eta}{\sqrt{G_k} + \epsilon}\nabla_{\theta_k}L(\boldsymbol{\theta}_k)$$

G is a matrix containing the squared sum of the past gradients with regards to all $\boldsymbol{\theta}$ along its diagonal. ϵ is a small quantity to avoid dividing by zero. For weight parameter with the largest gradient has the largest decrease in its learning rate, while the parameter with the smallest gradient has the smallest decrease in its learning rate. This results in faster progress in the more gentlysloped regions of parameter space and works well when the gradient is sparse [33].

RMSProp

The RMSProp algorithm is famous for being an unpublished but well known optimisation algorithm. RMSprop modifies Adagrad so that instead of accumulating the square of the gradients for each parameter, which can cause the learning rate to diminish; uses a exponentially decaying moving average of the squared gradients to discard the past history.

$$G_{k+1} \leftarrow \beta G_k + (1-\beta)(\nabla_{\theta_k}L(\boldsymbol{\theta}_k))^2$$
$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \frac{\eta}{\sqrt{G_k} + \epsilon}\nabla_{\theta_k}L(\boldsymbol{\theta}_k)$$

Adam

One of the most widely used optimisation algorithms used in deep learning today is Adam (adaptive moments). It combines elements of momentum and RMSprop with some modifications. Firstly, we incorporate momentum by computing the first order moment of the gradient as exponentially decaying moving average. Secondly, we calculate the second order moment as the gradient, again as an exponentially decaying average. The parameter update rule is given as:

$$\begin{aligned}
\boldsymbol{v}_k &\leftarrow \alpha \boldsymbol{v}_{k-1} + (1-\alpha)\nabla_{\theta_k} L(\boldsymbol{\theta}_k) \\
\boldsymbol{s}_k &\leftarrow \beta \boldsymbol{s}_{k-1} + (1-\beta)\nabla_{\theta_k} L(\boldsymbol{\theta}_k) \\
\boldsymbol{\theta}_{k+1} &\leftarrow \boldsymbol{\theta}_k - \eta \frac{\hat{\boldsymbol{v}}_k}{\sqrt{\hat{\boldsymbol{s}}_k + \epsilon}} \nabla_{\theta_k} L(\boldsymbol{\theta}_k)
\end{aligned}$$

In the above equations $\hat{\boldsymbol{v}}_k$, $\hat{\boldsymbol{v}}_k$ are the bias corrected estimates of the moments to account for their initialization. Since RMSProp lacks the correction factor, it has a high-bias in the early stages of training, while Adam is robust to the choice of hyperparameters.