Uppsala University

Department of Mathematics

UPPSALA
UNIVERSITET

master's thesis

March 4, 2021

Physics Informed Machine Learning of Nonlinear Partial Differential Equations

Jonas Mack

Supervisor: Gunnar Staff, Cognite AS

Abstract

Finding approximate solutions to nonlinear partial differential equations given some initial and boundary conditions is a well studied task within the field of numerical analysis. Nevertheless, numerical methods face several limitations with respect to the complexity of the underlying problem and the related computational effort.
In this work we aim to investigate whether methods from another discipline, namely machine learning, can solve such tasks and overcome the limitations of the numerical approaches. Specifically we consider physics informed neural networks, a recently discovered method that allows the encoding of the underlying partial differential equation directly into the loss function of a neural network by applying automatic differentiation with respect to the network's inputs. Our investigations suggest that these networks are indeed able to detect the imposed dynamics and overcome numerical issues related to stability and discontinuities.
The main result of this work is the discovery of a strong dependence between the overall performance of the method and the performance on the initial condition. We encountered that this dependence makes physics informed neural networks prone to produce wrong solutions if a small change in the initial condition can lead to a substantially different evolution of the system. This observation suggests that machine learning solvers are not yet ready to fully replace numerical approaches. Nevertheless, the achieved performance is very impressive and gives rise to the hope that this is just the beginning of an entirely new branch in scientific computing, i.e. solving partial differential equations using machine learning.

Acknowledgements

Just as everything in life, this work would have not been possible without the help of others, friends and family.
First and foremost I would like to thank my supervisor Gunnar Staff. Your continuous support during the entire process is greatly appreciated and the numerous conversations we had will be kept in very good memory. It has always been a lot of fun working with you. Your sharp mind together with your passion are truly inspiring.
Furthermore, I am incredibly thankful for all the software, papers, books and ideas that are available due to an increased Open Source mindset.
Finally, I want to express my outmost gratitude to my friends and family who supported me during this at times challenging period. Special thanks go to Adam, Julia and Marcel whose feedback on this work was of fundamental help. Most importantly, I wish to thank my parents Beate and Peter, my sister Julia and my brother Nico. Knowing that I have your unconditional support has always been an essential pillar in my life.

Contents

1. Introduction

The enormous applicability of *machine learning* methods to a wide variety of problem domains, including fields like image recognition, natural language processing and medical diagnosis just to mention a few, is simultaneously a cause and consequence of the increased attention paid to this field during the past decade. Despite all such advancements, machine learning methods have so far barely been used to tackle problems within the field of scientific computing including numerical analysis. While both fields involve algorithmic structures to find approximate solutions to complex problems, and therefore benefited from the recent growth in available computation power, they take quite different approaches.

The goal of this work is to study to what extent one of the main problems considered within the field of numerical analysis, namely the approximate solution to a given *partial differential equation* (PDE) together with some defined set of initial and boundary conditions, can be solved by using the tools developed within the field of machine learning.

Such problems, often referred to as initial and boundary value problems, arise frequently in a wide range of applications, including physics and engineering but also social sciences and finance. Thanks to their large number of applications, researchers have been trying to solve these kind of problems using numerical methods even since long before computers were available, which led to a vast and nowadays very well studied theory. Nevertheless, numerical approaches often require a profound understanding of the underlying problem. Moreover, if the dynamics imposed by the underlying PDE become very complex, frequently originating from nonlinear terms in the governing equation, the computational effort to solve the problem accurately might become unaffordably large.

On the contrary, there has only been very little research on how to solve such problems using a machine learning approach. In their recent work on *Physics Informed Deep Learning* [1] the authors introduced the concept of *physics informed neural networks* which aim to predict the solution to a PDE given some initial and boundary conditions by directly encoding the underlying PDE into the loss function using automatic differentiation. The model is then trained by generating the training data from the initial and boundary conditions.

Considering the impressive performance machine learning methods and in particular neural networks have been shown to have on such a wide range of applications, the concept of physics informed neural networks sounds promising. During the course of this work we aim to understand how these novel

networks compare to the classical approaches. While numerical approaches are deterministic, machine learning solutions often rely on some introduced randomness. Furthermore, machine learning solutions usually depend on a large set of parameters making it at times harder to interpret the results. Nevertheless, we will try to bring some light to the training process of physics informed neural networks.

We will start by defining an example problem that is considered during the entire course of this work, namely the one-dimensional viscous Burgers' equation with Dirichlet boundary and sinus-shaped initial conditions. The diffusion term is chosen to be very low such that the solution develops a sharp internal layer that looks like a discontinuity.
In the third chapter we will consider several numerical approaches to tackle this problem, which are all based on the method of *finite differences*. Due to the discontinuity in the example problem we expect some numerical difficulties that require a deeper understanding of the underlying problem, thus motivating us to consider a machine learning approach.
Afterwards, we will turn our attention to the aforementioned physics informed neural network method. The first part of the fourth chapter introduces the underlying mathematical concepts, i.e. neural networks and automatic differentiation. Since the novelty in the considered approach lies in the way automatic differentiation is applied, we will put a special focus on this topic. In the second part of the fourth chapter we will study how the method of physics informed neural networks performs on the example problem. By taking a closer look at the prediction of the initial condition and by considering another example, namely the Allen-Cahn equation, we will try to better understand the learning process.
We will conclude on the overall findings of this work by comparing the results found in the third and fourth chapter. Furthermore, we will discuss some future research options that have the potential to advance this recent field of finding an approximate solution to a PDE using machine learning.

All methods presented in this work are implemented in Python. The code is publicly available on GitHub.[1]

[1] `https://github.com/sch0ngut/machine-learning-of-pdes`

2. Burgers' Equation as a Nonlinear Example Problem

As an example problem for solving nonlinear PDEs we will consider the one-dimensional viscous Burgers' equation, given as

$$u_t + uu_x = \nu u_{xx}, \quad x \in [-1, 1], \quad t \in [0, 1] \tag{2.1}$$

together with the initial condition

$$u(x, 0) = -\sin(\pi x), \quad x \in [-1, 1] \tag{2.2}$$

and the Dirichlet boundary conditions

$$u(1, t) = u(-1, t) = 0, \quad t \in [0, 1]. \tag{2.3}$$

Historically, Burgers' equation was first studied by Bateman [2] and only later considered by Burgers [3] as part of his studies concerning the mathematical modelling of turbulences. Hence Burgers' equation is widely used to address problems in fluid dynamics but is also found in many other areas of applied mathematics.

During the course of this work we will refer to the problem defined by the equations (2.1)–(2.3) as the *example problem* and consider the viscosity parameter ν to be given by

$$\nu = \frac{1}{100\pi} \tag{2.4}$$

since for small values of ν the solution to the problem develops a sharp internal layer which looks like a discontinuity as shown in Figure 1 where the exact solution to the problem is presented visually. We can see that for $x = 0$ at around $T > 0.4$ the solution does not seem smooth anymore.

As we will see in the following section, this sharp internal layer, also called shock formation, can be hard to resolve for classical numerical methods which makes it an interesting example for our study. One of the questions we will try to answer is whether a machine learning approach is able to overcome this difficulty.

Besides the reasons mentioned so far, studying the example problem has the additional practical advantage that an exact solution can be computed analytically as shown in [4], which allows to easily study and compare the performance of the considered numerical and machine learning solvers.

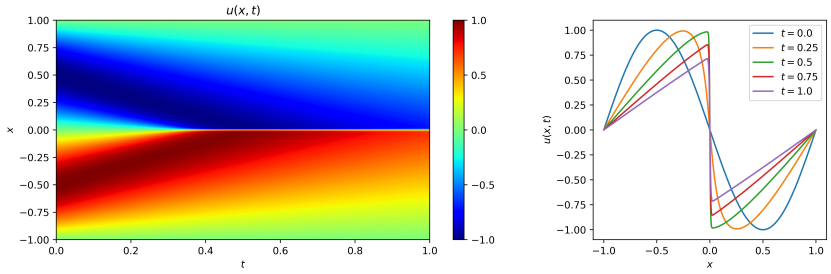


Figure 1: Exact solution to the example problem. *Left:* Contour plot on the entire (x, t)-domain. *Right:* Snapshots at selected time points.

3. Finite Difference Solutions

Before studying how the example problem can be solved using a machine learning approach, we will take a look at how classical (i.e. numerical) methods can be applied. Throughout this section we will consider several numerical methods that are all based on the method of finite differences using a uniform mesh.

We begin by discretising the entire spatio-temporal domain denoted by

$$\Omega = [a, b] \times [0, T] = [-1, 1] \times [0, 1]$$

using $H + 1$ equidistant points in space and $K + 1$ equidistant points in time. Further, let $h = (b - a)/H$ denote the mesh size and and $k = T/K$ the step size such that we can write the discrete points in time $\{t^n\}_{n=0}^K$ and in space $\{x_j\}_{j=0}^H$, respectively, as

$$\begin{aligned} x_j &= a + j \cdot h, & j &= 0 \dots H \\ t^n &= n \cdot k, & n &= 0 \dots K. \end{aligned}$$

Finally, we will use the notation u_j^n for the numerical approximation of $u(x_j, t^n)$, i.e.

$$u_j^n \approx u(x_j, t^n).$$

Note that for readability reasons lower indices on x and u are used to indicate the spatial discretisation and upper indices on t and u to indicate the temporal discretisation.

After discretising the spatio-temporal domain Ω we need to find discrete approximations for the derivatives occurring in the Burgers' equation (2.1). In the following sections we will present different ways to do so and discuss the expected behaviour and possible problems.

3.1. A Simple FTCS Approach

One of the most obvious schemes to represent the derivatives occurring in (2.1) is the so called *Forward Time Centered Space* (FTCS) discretisation. Hereby, the spatial derivatives are approximated using an explicit second order central difference stencil. This means that we will approximate the convection term in the point (x_j, t^n) as

$$uu_x = u(x_j, t^n)u_x(x_j, t^n) \approx u_j^n \frac{u_{j+1}^n - u_{j-1}^n}{2h} \tag{3.1}$$

and similarly the diffusion term as

$$\nu u_{xx} = \nu u_{xx}(x_j, t^n) \approx \nu \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}. \tag{3.2}$$

Finally, the time derivative u_t is approximated using the first order explicit Euler, or forward time difference scheme, i.e.

$$u_t = u_t(x_j, t^n) \approx \frac{u_j^{n+1} - u_j^n}{k}. \tag{3.3}$$

Plugging the approximations (3.1), (3.2), (3.3) into the Burgers' equation (2.1) yields the following FTCS approximation:

$$u_j^{n+1} = u_j^n + k \cdot \nu \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2} - k \cdot u_j^n \frac{u_{j+1}^n - u_{j-1}^n}{2h}.$$

Combining this with the discrete versions of the initial and boundary conditions (2.2), (2.3), i.e.

$$\begin{aligned} u_j^0 &= -\sin(\pi x_j), & j &= 0 \dots H \\ u_0^n = u_H^n &= 0, & n &= 0 \dots K \end{aligned}$$

allows us compute a discrete, approximate solution to the problem on the entire spatio-temporal domain Ω.

Using Taylor expansion we can calculate the local truncation error as

$$\begin{aligned} \tau(x_j, t^n) &= |u(x_j, t^n) - u_j^n| \\ &= \left| \frac{k}{2}[u_{tt}]_j^n + u_j^n \frac{h^2}{6}[u_{xxxx}]_j^n - \nu \frac{h^2}{12}[u_{xxxx}]_j^n + \mathcal{O}(k^2) + \mathcal{O}(h^4) \right|, \end{aligned} \tag{3.4}$$

and therefore say that the method has local order $\mathcal{O}(k) + \mathcal{O}(h^2)$. Note that equation (3.4) contains terms of the form u_{tt} and u_{xxxx} which are unknown,

thus making it impossible to compute the truncation error exactly. However, the local order tells us at which rate the error decreases as we refine the mesh size h and the step size k. Note that this is just local information about the error which ideally, but not generally, also holds for the global error. Since local errors can propagate throughout the entire system, and therefore accumulate in each iteration, giving an adequate estimate about the behaviour of the global error requires in general a more profound study of the method and the problem at hand. A method for which the global error follows the same order of magnitude as the local error is called *stable*. In other words, a stable method guarantees that the global error has the same behaviour as the local error. The order of the local error is usually easy to obtain using Taylor expansions as we did above. This provides us with a good indicator about the accuracy of our numerical approximation.
Even though the theory of stability analysis is well studied in the literature, performing such an analysis is problem dependent and not always straightforward, especially in the case of nonlinear problems. This makes it a major drawback of numerical methods, as a numerical scheme with a low local error but without stability will usually not give a good approximation to the exact solution.

Due to the nonlinear nature of our example problem, performing a stability analysis is not trivial. Instead one would consider the linearised version of the Burgers' equation (2.1), which is also known as the linear convection-diffusion equation, given as

$$u_t + cu_x = \nu u_{xx}, \tag{3.5}$$

where c is a constant. Even for this simplified since linearised problem, performing a stability analysis still requires a lot of computations by hand. Hence we will not bother with the details here and instead refer to [5], where the author computed the stability condition for the FTCS approach to the linearised version of our example problem (3.5) to be given as

$$k \leq \min\left\{\frac{2\nu}{c}, \frac{h^2}{2\nu}\right\}. \tag{3.6}$$

In the setting of our example problem, just guaranteeing stability is not enough to obtain accurate approximations to the exact solution. Since we are only approximating the occurring derivatives, the numerical method is prone to generate oscillations around the discontinuity that is developing for $x = 0$ at around $T > 0.4$ if the spatial grid is not fine enough. In order to avoid this unwanted behaviour, we need to make sure that the spatial grid size is

fine enough. But what does "fine enough" mean? Similarly to the stability analysis, the answer to this question is problem dependent and not trivial at all. For our example problem using an FTCS approach, however, the condition on the grid size to avoid oscillations around the discontinuity can be explicitly determined as

$$h \leq \frac{2\nu}{c}. \tag{3.7}$$

For more details on how to derive this condition, see [5].

Note that the two conditions to guarantee stability (3.6) and to avoid oscillations (3.7), respectively, apply to the linearised version of the Burgers' equation (3.5). In order to apply those results to the nonlinear Burgers' equation, we need to specify the value of the constant c. A reasonable choice is given by the maximum absolute value of the initial condition, i.e.

$$c = \max |u_0(x)| = 1.$$

After this theoretical discussion about how well we can approximate the exact solution and which conditions need to be satisfied, it is interesting to see if these theoretical findings also hold in practice. As a measure for the global error we will use the ℓ^2 and the maximum error, defined as

$$\begin{aligned} \varepsilon(h,k) &= \left(h \cdot k \sum_{j,t} (u_j^n - u(x_j, t^n)^2 \right)^{1/2} \\ \varepsilon_{max}(h,k) &= \max_{i,j} |u_j^n - u(x_j, t^n)|. \end{aligned}$$

We consider both errors because the exact solution is almost discontinuous meaning that a large maximum error ε_{max} could still allow for a relatively small ℓ^2 error ε. Thus we want to make sure that both errors decrease at a similar rate.

Since the discontinuity is occurring in space, we will keep the step size k fixed while decreasing the mesh size h. In doing so we can calculate the experimental order of convergence q_ε as

$$q_\varepsilon = \frac{\log(\varepsilon(h_1, k)/\varepsilon(h_2, k))}{\log(h_1/h_2)}.$$

If the method is stable, we would expect $q_\varepsilon \approx 2$, due to (3.4).

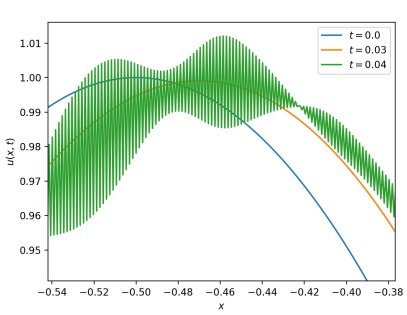
H	ε	q_ε	ε_{max}	$q_{\varepsilon_{max}}$	Note
160	∞	-	∞	-	$h \geq 2\nu$
320	1.02e-02	-	2.16e-01	-	
640	1.93e-03	2.40	3.60e-02	2.58	
1280	4.67e-04	2.05	9.02e-03	2.00	
2560	∞	-	∞	-	$k \geq h^2/(2\nu)$

Table 1: FTCS: Errors and convergence rates; $k = 1 \times 10^{-4}$.

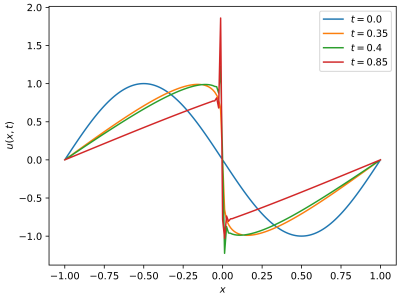
Now we have all the necessary tools to compute a numerical approximation of the problem defined by the equations and study the behaviour of the error. Using a step size of $k = 1 \times 10^{-4}$ and a number of $H \in \{160, 320, 640, 1280, 2560\}$, we observe the errors and convergence rates shown in Table 1.

The results confirm our theoretical findings. Using a step size of $k = 1 \times 10^{-4}$ together with $H \in \{320, 640, 1280\}$ yields a numerical solution that approaches the exact one at a convergence rate of 2 for both the maximum and the overall ℓ^2 error which meets our expectations. If, however, we take too many spatial discretisation points as in the case $H = 2560$, the stability condition (3.6) is violated. After only a few iterations, the numerical solution starts oscillating on the entire spatial domain until it blows up at some point as shown in Figure 2a. But if on the other hand we take too few spatial discretisation points, as in the case $H = 160$, the numerical solution introduces oscillations around the discontinuity since the condition (3.7) is violated. This behaviour is illustrated in Figure 2b. Here we can see that as soon as the exact solution becomes discontinuous (i.e. at around $T > 0.4$), the numerical approximation starts oscillating around the discontinuity. These oscillations amplify every time step until the solution blows up at some point.

Throughout this section we have derived a numerical method to approximately solve our example problem involving the nonlinear viscous Burgers' equation. We discussed the accuracy properties of this scheme and how to choose step size and mesh size such that we can expect good results. Unfortunately, the conditions on the step size and the mesh size only apply to this specific problem. In the next section we will therefore look at another discretisation technique that aims to solve those issues in a more general fashion.



(a) Unstable solution using $H = 2560$.



(b) Oscillations around the discontinuity using $H = 160$.

Figure 2: Typical behaviour of unstable and oscillating solutions, caused by an inappropriate choice of the mesh size h; $k = 1 \times 10^{-4}$.

3.2. Upwind Methods with Implicit Diffusion

To avoid the oscillations around the discontinuity, we can use a so called *upwind scheme* for the convection term uu_x. The idea is to use a one sided approximation for the derivative rather than symmetric approximations. The reason why an upwind scheme can be beneficial lies in the fact that the convection term models the propagation of a wave at speed u that is centered around the discontinuity at $x = 0$. Hence there is an asymmetry in the information flow which is responsible for the oscillations around the discontinuity that we could observe in the previous section when the grid was not fine enough. To remedy this problem we can choose to take one-sided approximations of the derivative depending on the direction of the information flow, i.e. depending on the sign of u. Historically this idea was first proposed by Courant, Isaacson, and Rees [6].

In mathematical terms a first order upwind approximation to the convection term in the point (x_j, t^n) can be expressed as follows:

$$uu_x = u(x_j, t^n)u_x(x_j, t^n) \approx u^+ u_x^- + u^- u_x^+, \tag{3.8}$$

where

$$\begin{aligned} u_x^- &= \frac{u_j^n - u_{j-1}^n}{h}, & u_x^+ &= \frac{u_{j+1}^n - u_j^n}{h}, \\ u^+ &= \max(u_j^n, 0), & u^- &= \min(u_j^n, 0). \end{aligned} \tag{3.9}$$

Note that in equation (3.8) the lower and upper indices indicating the spatio-temporal grid point are omitted for readability reasons. In case we want to emphasise the grid point, we will use the notation $[u^+u_x^- + u^-u_x^+]_j^n$.

To also improve on the stability properties, we can choose to approximate the diffusion term with an implicit central difference scheme rather than an explicit one as we did in the previous section. It is well known that implicit finite difference schemes have better stability properties, and therefore allow for larger choices of the time step h than explicit ones. However, they come at the cost of extra computation time since they require to solve a linear system of equations at every time step.

Hence we approximate the diffusion term in the point (x_j, t^n) as

$$\nu u_{xx} = \nu u_{xx}(x_j, t^n) \approx \nu \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{h^2}. \tag{3.10}$$

Note that compared to the first approach in (3.2), the right hand side now uses the solution at time point t^{n+1} instead of t^n.

Plugging the approximations (3.8), (3.10) and the explicit Euler time discretisation (3.3) into the Burgers' equation (2.1) yields the following first order upwind scheme:

$$u_j^{n+1}\left(1 + 2\frac{k\nu}{h^2}\right) - u_{j+1}^{n+1}\left(\frac{k\nu}{h^2}\right) - u_{j-1}^{n+1}\left(\frac{k\nu}{h^2}\right) = u^n - k[u_x^- u^+ + u_x^+ u^-]_j^n.$$

We can see that the left hand side with the terms at time t^{n+1} utilises the solution at three different points in space which requires to solve a system of linear equations at every time step as mentioned above.

The errors of the first order upwind scheme together with the convergence rates are presented in Table 2. We can see that the numerical solution for every chosen mesh size $h = 1/H$ approximates the exact solution reasonably well and the numerical scheme seems stable. However, the convergence rate of the error approaches only 1, as the upwind approximation of the convection term (3.8), (3.9) utilises first order accurate approximations of the first derivative. Hence the convergence of the error is slower than for the FTCS scheme in the previous section, yielding a larger error. Note that due to the improved stability properties, a larger step size of $k = 1 \times 10^{-3}$ was used. In fact, a smaller time step would not increase the accuracy significantly, as the spatial error is dominating at this point.

H	ε	q_ε	ε_{max}	$q_{\varepsilon_{max}}$
160	2.22e-02	-	2.65e-01	-
320	1.43e-02	0.63	1.91e-01	0.47
640	8.21e-03	0.80	1.18e-01	0.69
1280	4.48e-03	0.87	6.68e-02	0.82
2560	2.36e-03	0.92	3.52e-02	0.93

Table 2: First order upwind: Errors and convergence rates; $k = 1 \times 10^{-3}$.

The two performed adjustments in the discretisation of the convection and the diffusion term (3.8), (3.10) gave us more flexibility when choosing the time step k and the mesh size h. However, the increased flexibility came at the cost of a lower convergence rate of the error compared to the FTCS approach since the approximation of the derivative in the upwind term is only first order accurate. To regain the second order convergence of the error, while keeping the advantages of the upwind scheme, we can choose a second order accurate upwind approximation of the derivative in the convection term, given by

$$u^-_{x,j} = \frac{3u^n_j - 4u^n_{j-1} + u^n_{i-2}}{2h}, \quad u^+_{x,j} = \frac{-u^n_{j+2} + 4u^n_{j+1} - 3u^n_j}{2h}.$$

Unfortunately, this scheme is known to have worse stability properties which requires to take a smaller time step again. For more details on this approximation and how to derive it, see [7].

The errors of the second order upwind scheme together with the convergence rates are presented in Table 3. As expected, the convergence orders of both errors approach 2 until $H = 2560$. Furthermore, the solution is stable for any mesh size $h = 1/H$. This time we also included $H = 5120$ to discuss another drawback of numerical methods. As we can see the convergence rate drops at this grid size and the more we increase H the more the convergence rate will decrease, as long as we keep the step size k fixed. This behaviour is caused by the temporal error which at this point is large enough to not let the overall error decrease at the expected convergence rate any further. Hence in order to maintain the convergence rate of 2, a simultaneous refinement of both step size k and mesh size h would be necessary.

We can conclude that an upwind discretisation of the nonlinear convection term is a good way to handle the discontinuity that is caused by such. Together with an implicit representation of the diffusion term we can further guarantee

H	ε	q_ε	ε_{max}	$q_{\varepsilon_{max}}$
160	1.11e-02	-	1.35e-01	-
320	5.62e-03	0.98	8.76e-02	0.62
640	2.08e-03	1.43	3.51e-02	1.32
1280	6.33e-04	1.72	1.10e-02	1.67
2560	1.72e-04	1.88	2.86e-03	1.95
5120	7.00e-05	1.30	8.70e-04	1.71

Table 3: Second order upwind: Errors and convergence rates; $k = 1 \times 10^{-4}$.

stability. However, we have seen that even though we have a stable solver, the error will not always reduce with the local convergence rate as we refine the spatial grid, since for a given step size k at some point the temporal error will cause the overall error to not reduce any further. Although this might not be very problematic in the setting at hand, as we already obtain quite accurate results with the chosen grid size, the situation might be very different when the underlying equation is more complex and the computational domain Ω much bigger. In that case the computational effort could become unaffordably large to generate accurate approximations in a reasonable time.

3.3. A Linear Solver Using the Hopf-Cole Transformation

As we have seen in the FTCS approach, a difficulty when solving nonlinear PDEs are the discontinuities that can be caused by such. Using the upwind approach in the previous section, we have seen how this issue can be handled when the discontinuity is caused by a convection term. For the problem at hand, there is another way to handle this, which is to transform the one-dimensional nonlinear viscous Burgers' equation (2.1) into the linear heat equation using the Hopf-Cole transformation as presented in [8]. Avoiding the nonlinear term does not only save us the trouble of worrying about the oscillations around the discontinuity but it also heavily simplifies the stability analysis.

We will now go through the necessary steps to transform the viscous Burgers' equation into the heat equation using the Hopf-Cole transformation. First, we

observe that the viscous Burgers' equation (2.1) can be written as

$$u_t = \left(\nu u_x - \frac{u^2}{2}\right)_x. \tag{3.11}$$

The Hopf-Cole transformation was first proposed in [9] and is given by

$$\psi = \exp\left(-\frac{1}{2\nu}\int u dx\right),$$

which we can rewrite for u as

$$u = -2\nu \log(\psi)_x = -2\nu\frac{\psi_x}{\psi}. \tag{3.12}$$

Applying this to the left hand side (LHS) of (3.11) we can write

$$LHS = -2\nu \log(\psi)_{xt} = -2\nu \log(\psi)_{tx} = -2\nu\left(\frac{\psi_t}{\psi}\right)_x \tag{3.13}$$

and similarly the right hand side (RHS)

$$\begin{aligned} RHS &= \left[-2\nu^2\left(\frac{\psi_x}{\psi}\right)_x - 2\left(\nu\frac{\psi_x}{\psi}\right)^2\right]_x \\ &= \left[-2\nu^2\frac{\psi_{xx}\psi - \psi_x^2}{\psi^2} - 2\nu^2\frac{\psi_x^2}{\psi^2}\right]_x \\ &= -2\nu^2\left(\frac{\psi_{xx}}{\psi}\right)_x. \end{aligned} \tag{3.14}$$

Equating LHS (3.13) and RHS (3.14) as well as integrating with respect to x yields

$$\psi_t = \nu\psi_{xx} + C(t)\psi,$$

where $C(t)$ is a function depending on the boundary conditions. If $C(t) = 0$, i.e. for a periodic boundary, we are left with the one-dimensional heat equation

$$\psi_t = \nu\psi_{xx}, \tag{3.15}$$

with the initial condition

$$\psi_0 = \psi(0,x) = \exp\left(-\frac{1}{2\nu}\int -\sin(\pi x)dx\right) = \exp\left(\frac{1}{2\nu\pi}[1-\cos(\pi x)]\right). \tag{3.16}$$

By using the transformation (3.12), the boundary conditions can be given as

$$\psi_x(-1,t) = \psi_x(1,t) = 0, \quad t > 0. \tag{3.17}$$

In order to find a finite difference approximation of the heat equation (3.15) together with the initial condition (3.16) and the boundary conditions (3.17) the diffusion term can then be approximated in the same fashion as we did in the FTCS approach using an explicit second order central difference stencil (3.2) with local order $\mathcal{O}(h^2)$. Thus the semi-discrete approximation of the heat equation reads as

$$\vec{\psi}_t = A\vec{\psi},$$

with $\vec{\psi}$ and $\vec{\psi}_t$ being the vector of the solution and its derivative with respect to time at the spatial discretisation points $\{x_j\}_{i=0}^{H}$. Hereby, A is an $(H+1) \times (H+1)$ matrix, given as

$$A = \begin{pmatrix} -2c & 2c & & & & & \\ c & -2c & c & & & & \\ & c & \ddots & \ddots & & & \\ & & \ddots & \ddots & c & & \\ & & & c & -2c & c & \\ & & & & 2c & -2c \end{pmatrix}, \quad c = \frac{\nu}{h^2}.$$

Note how the first and last line of the matrix A incorporate the boundary conditions (3.17).

To guarantee better stability properties, we will now use a fourth order accurate Runge-Kutta scheme instead of a simple Euler forward approximation. Hence the time integration is given by the iteration formula

$$\psi^{n+1} = \psi^n + \frac{1}{6}k(w_1^n + 2w_2^n + 2w_3^n + w_4^n), \tag{3.18}$$

with

$$\begin{aligned} w_1^n &= A\psi^n, \\ w_2^n &= A(\psi^n + 0.5k \cdot w_1^n), \\ w_3^n &= A(\psi^n + 0.5k \cdot w_2^n), \\ w_2^n &= A(\psi^n + k \cdot w_3^n). \end{aligned}$$

Once we have the discrete approximation ψ_j^n to the heat equation we can retransform those values back to the Burgers' equation using (3.12) together

H	ε	q_ε	ε_{max}	$q_{\varepsilon_{max}}$	Note
160	2.55e-01	-	1.28e-00	-	
320	6.33e-02	2.01	3.07e-01	2.07	
640	1.57e-02	2.01	6.97e-02	2.14	
1280	3.93e-03	2.00	1.70e-02	2.04	
2560	9.82e-04	2.00	4.22e-03	2.01	
5120	∞	-	∞	-	$Q > 1$

Table 4: Hopf-Cole approach: Errors and convergence rates; $k = 1 \times 10^{-4}$.

with a second order central in space approximation of the first spatial derivative ψ_x, i.e.

$$u_j^n = -\frac{\nu}{h}\left(\frac{\psi_{j+1}^n - \psi_{j-1}^n}{\psi_j^n}\right).$$

As mentioned earlier, one of the main advantages of transforming the nonlinear Burgers' equation into the linear heat equation is the fact that we can easily perform a stability analysis. The stability region of the Runge-Kutta solver (3.18) can be looked up in any standard literature dealing with the method of finite differences, e.g. [10], and reads as

$$Q(\lambda_\omega) \coloneqq \left|1 + k\lambda_\omega + \frac{(k\lambda_\omega)^2}{2} + \frac{(k\lambda_\omega)^3}{6} + \frac{(k\lambda_\omega)^4}{24}\right| \leq 1,$$

with λ_ω being the eigenvalues of the matrix A which in general can be easily computed numerically, or in this case even analytically, as

$$\lambda_\omega = \frac{e^{i\omega h} - 2 + e^{-i\omega h}}{h^2} = \frac{(2i)^2}{h^2} \cdot \left(\frac{e^{i\omega h/2} - e^{-i\omega h/2}}{2i}\right)^2 = -\frac{4}{h^2}\sin\left(\frac{\omega h}{2}\right)^2.$$

The errors of the derived scheme together with the convergence rates for a step size $k = 1 \times 10^{-4}$ are presented in Table 4. Both errors approach the expected convergence rate of 2 until the method is not stable anymore ($H = 5120$). Compared to the previous methods, we can see that the ℓ^2 error ε and the maximum error ε_{max} are much closer to each other thanks to the transformation of the underlying equation.

3.4. Some Concluding Remarks on the Numerical Approaches

During the course of the first part of this work we have seen several numerical approaches to solve the example problem.
We started out by using a simple FTCS approach that was easy to derive but showed to have bad stability properties and further was prone to oscillations around the discontinuity. Even though we were able to mathematically determine which choices of the mesh size h and the step size k avoid those unwanted side effects, the performed analysis was rather elaborate and problem specific. By using an implicit representation of the diffusion term and an upwind one for the convection term, those issues could be resolved. It is important to note, however, that the correct choice of mesh size h and step size k can still influence whether the optimal convergence rate is obtained. Moreover, the implicit representation comes at the cost of increased computation time, as we need to solve a system of linear equations at every time step. While in the case of the example problem this is not an issue, as the computational domain Ω is quite small, this might be different when the considered domain becomes larger.
Finally, we also looked at the possibility of transforming the Burgers' equation into the linear heat equation which avoided the occurrence of the oscillations generated by the convection term around the discontinuity and simplified the stability analysis. However, this transformation only applies to the considered Burgers' equation and in general it is not possible to transform nonlinear equations into linear ones.

The conducted analyses showed that we cannot use numerical approaches as an out-of-the-box solution for solving PDEs but instead need to make sure that the method produces a stable approximation. Furthermore, a special treatment of occurring discontinuities is usually necessary which requires a deeper understanding of the underlying problem.
While those things could be easily handled for the example problem, this might not be the case when the underlying problem gets more complex such that we do not know where the discontinuities arise or such that a very coarse discretisation of the spatio-temporal domain Ω is required which can make the computational effort unaffordably large.
In the next part of this work, we will study the degree to which machine learning solvers are able to approximately solve PDEs given some initial and boundary conditions. Of particular interest is the question whether such an approach is able to overcome the aforementioned limitations of the finite difference methods.

4. Physics Informed Neural Network Solutions

After considering the classical approaches for solving a nonlinear PDE in the previous chapter, we will now take a look at how methods of another discipline, namely machine learning, can be utilised. Even though this field gained a lot of attention during the last decades, leading to substantial improvements and a wide range of applications, its methods were barely used to solve problems within the domain of scientific computing such as our example problem (2.1)–(2.3), i.e. finding the solution to a PDE given some initial and boundary conditions. The reason for this lies in the different nature of the problem settings. In machine learning one is typically interested in making a prediction or a decision based on some training data. Hence many methods within the field of machine learning are suitable to solve regression-like problems, i.e. finding the relationship between dependent and independent variables (features and targets, respectively). In our example problem we are also interested in finding such a relationship, i.e. the function $u(x,t)$. However, we are not given any training data but instead the underlying physical dynamics and some initial and boundary conditions. Even if we could generate some training data, and therefore consider the problem of solving a PDE as a regression problem, it is not clear how to include the underlying physical dynamics of the equation into the model. Therefore we need to go beyond the standard methods and find a way to include this knowledge.

In [1] the authors presented a solution to this issue and introduced the concept of *physics informed neural networks*. The idea is to interpret the problem of finding the solution to a PDE as a regression problem and to use the initial and boundary conditions to sample training data. The regression problem is then solved using a neural network. The underlying dynamics of the PDE are included via a regularisation term in the loss function by applying automatic differentiation to the network's output. In the following section we will discuss this idea in more detail and introduce the underlying mathematical concepts.

4.1. The Mathematical Concepts

4.1.1. Neural Networks

As stated in the introductory part of this chapter, we are interested in solving a regression-like problem, i.e. estimating the continuous relationship between a dependent variable u (the target) and one or more independent variables

$\mathbf{z} = (z_1, \ldots, z_p)^T$ (the features), given a set of training data $\mathcal{T} = \{u_i, \mathbf{z_i}\}_{i=1}^{N_\mathcal{T}}$.[2] Hence our goal is to find a function $\hat{u} = f(\mathbf{z}; \theta)$ that approximates the true function u for any input $\mathbf{z}$.[3] Here f is typically a nonlinear function that is given by the underlying regression method and dependent on a set of unknown parameters $\theta = (\theta_1, \ldots, \theta_m)^T$. Ideally θ is chosen such that

$$\theta_{opt} = \min_\theta ||f(\mathbf{z}; \theta) - u(\mathbf{z})|| \tag{4.1}$$

for a given norm and any input $\mathbf{z}$. However, we cannot evaluate equation (4.1) on any $(u, \mathbf{z})$-pair, but only on the training data $\mathcal{T}$. Thus we approximate θ_{opt} by

$$\theta = \min_\theta ||f(\mathbf{z_i}; \theta) - u_i||, \quad i = 1, \ldots, N_\mathcal{T} \tag{4.2}$$

using the training data $\mathcal{T}$. The process of finding θ is called *model training*. Depending on the number of parameters m introduced by the method f, finding θ can be very complex, requiring that we rather look for an approximation of θ itself.

We will now define common machine learning methods for the regression function f.

Definition 1. A *generalised linear regression model* is given by the function

$$f(\mathbf{z}; \theta) = h\left(\sum_{j=1}^{p} w_j z_j + b\right) = h(\mathbf{w}\mathbf{z} + b),$$

where h is often referred to as the *activation function*.[4] The trainable parameters of this model are the *offset* b and the *weights* $\mathbf{w} = (w_1, \ldots, w_p)^T$, i.e.

$$\theta = (b, w_1, \ldots, w_p)^T.$$

Definition 2. An *artificial neural network* (ANN) is a sequential construction of several generalised linear regression models. A network with L *hidden layers*

[2] In the setting of the example problem, we have $\mathbf{z} = (x, t)$.

[3] In accordance with the one-dimensional Burgers' equation, we assume here $u, \hat{u}$ and f to be scalars. However, all the presented tools naturally generalise to higher dimensional functions as well.

[4] The activation function is a hyper-parameter defined by the engineer. Common choices are *sigmoid*, tanh or *ReLU*. For more details on these and other activation functions, see [11].

and N *nodes* per hidden layer can be expressed as follows:

$$
\begin{aligned}
q^{(0,n)} &= h\left(b^{(0,n)} + \sum_{i=1}^{p} w_i^{(0,n)} z_i\right), && n = 1, \dots, N \\
q^{(l,n)} &= h\left(b^{(l,n)} + \sum_{i=1}^{N} w_i^{(l,n)} q^{(l-1,i)}\right), && n = 1, \dots, N, \quad l = 1, \dots, L-1 \\
q^{(L)} &= h\left(b^{(L)} + \sum_{i=1}^{N} w_i^{(L)} q^{(L-1,i)}\right).
\end{aligned}
$$

Each node represents one generalised linear regression model. Note that the network uses $\mathbf{z}$ as an input on the first layer. The output of the last layer combines the previous layers iteratively and the ANN can be summarised as

$$\hat{u} = f(\mathbf{z}; \theta) = q^{(L)}.$$

The trainable parameters θ of this model are the set of all offsets b and all weights w from each node.

Remark. Note that in the above definition of ANNs, every layer $q^{(i,\cdot)}$ uses only the output of the previous layer as an input, the same activation function h and the same number of nodes N per hidden layer. Generally speaking these are just simplifications and it is possible to design more complex ANN architectures, see for example [12]. In this work, however, we will restrict ourselves to ANNs as formulated in Definition 2.

The power of Neural Networks for regression-like problems becomes very clear by the following theorem.

Theorem 1. *A Neural Network as formulated in Definition 2 with at least one hidden layer and a sigmoidal activation function can approximate any continuous function to any degree of accuracy.*

The theorem was first stated and proven by Cybenko in 1989, see [13] for the original formulation and its proof. Together with similar results it is known as one of the *universal approximation theorems*.

Even though the theorem tells us that we can theoretically approximate any continuous function, this may not only require more than the available training data but also a lot of nodes leading to a large set of parameters θ. Furthermore,

the theorem does not provide any information on how to obtain this network and its parameters θ. Hence the result is rather of theoretical than of practical relevance.

Let us now take a closer look at how to obtain the model parameters θ, i.e. the process of model training. As stated earlier, we want to choose the parameters in a way that the distance between the predicted solution of the network and the true solution on the training data is minimised by some norm, see equation (4.2). At this point it is common practice in machine learning to introduce a loss function $L(\theta, \mathcal{T})$ to measure the aforementioned distance. Common choices for the loss function are the *mean absolute error* (MAE) and the *mean squared error* (MSE) defined as

$$L_{MAE} = \frac{1}{N_{\mathcal{T}}} \sum_{i=1}^{N_{\mathcal{T}}} |f(\mathbf{z_i}; \theta) - u_i| \quad \text{and} \quad L_{MSE} = \frac{1}{N_{\mathcal{T}}} \sum_{i=1}^{N_{\mathcal{T}}} (f(\mathbf{z_i}; \theta) - u_i)^2.$$

Finding the global minimum of the loss function $L(\theta, \mathcal{T})$ can be quite complex, especially in the case where the set of parameters, and consequently the dimensionality, is large. Thus in most cases one has to settle with a local optimum. In order to find such an optimum most training procedures, also referred to as *optimisers*, make use of *gradient descent*, an iterative method to find the optimum of a (possibly high-dimensional) function. The idea is to move the parameters θ at each iteration step i closer to its optimum, i.e.

$$\begin{aligned} \theta^{(i+1)} &= \theta^{(i)} - \gamma[\nabla_\theta L(\theta^{(i)}, \mathcal{T})] \\ &= \theta^{(i)} - \gamma g^{(i)} \end{aligned} \tag{4.3}$$

with γ the *step length*, also known as the *learning rate*, and $g^{(i)} = \nabla_\theta L(\theta^{(i)}, \mathcal{T})$ the gradient of the loss function with respect to the model parameters θ in the point $\theta^{(i)}$.
Note the importance of the learning rate γ in the update equation (4.3). If we choose γ to be too small, the method will only converge very slowly towards its optimum. If contrary γ is too large, the method will overshoot this optimum. Hence it is desirable to adapt the learning rate during the training process.
Furthermore, it would be helpful to keep track of the past gradient at every iteration. If it points in the same direction as the previous gradient we would like to accelerate, otherwise decelerate. This method, known as *momentum*, helps to move faster in the relevant directions by avoiding oscillations, and is thus very useful in ravine-like environments of the loss function [14]. In its

simplest form momentum can be implemented as

$$m^{(i+1)} = \beta m^{(i)} + \gamma g^{(i)}$$
$$\theta^{(i+1)} = \theta^{(i)} - m^{(i+1)},$$

with β the momentum factor, usually set around 0.9.

One of the most popular gradient descent optimisers, that implements both an adaptive learning rate and momentum, is the *Adam* optimiser [15], which will also be used during the course of this work. Adam, which stands for adaptive moment estimation, uses an exponentially decaying average of the squared gradients $[g^{(i)}]^2$ to realise the adaptive learning rate and an exponentially decaying average of the gradients $g^{(i)}$ to include momentum.
By using the following estimates of the first and the second moment of the gradient

$$m^{(i+1)} = \beta_1 m^{(i)} + (1-\beta_1) g^{(i)}$$
$$v^{(i+1)} = \beta_2 v^{(i)} + (1-\beta_2)[g^{(i)}]^2$$

and their bias-corrected versions[5]

$$\hat{m}^{(i)} = \frac{m^{(i)}}{1-\beta_1^t}$$
$$\hat{v}^{(i)} = \frac{v^{(i)}}{1-\beta_2^t},$$

the update-formula of the Adam optimiser can be given as

$$\theta^{(i+1)} = \theta^{(i)} - \frac{\gamma}{\sqrt{\hat{v}^{(i)}} + \varrho} \hat{m}^{(i)},$$

where ϱ is an infinitesimal constant to avoid division by zero. The authors of Adam suggest to choose $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\gamma = 0.001$ and $\varrho = 1 \times 10^{-8}$ [15].

The gradient of the loss function $g^{(i)}$ with respect to each parameter can be efficiently computed by applying the chain rule and iteratively computing one layer at a time starting from the last layer. In doing so, redundant calculations of intermediate terms are avoided. This idea is known as *backpropagation*, and is generalised by the concept of *automatic differentiation*, which we will introduce now.

[5] For more details on the bias we refer to the original work [15].

4.1.2. Automatic Differentiation

Automatic differentiation, also known under the terms *algorithmic differentiation* or *computational differentiation*, is a well established technique to evaluate the derivatives of numeric functions. Other techniques to solve this task include *numerical* and *symbolic differentiation*. While numerical differentiation, using the method of finite differences, introduces discretisation errors that can lead to instabilities as we have seen in the first part of this work, symbolic differentiation requires the function of interest to be given in a closed form expression, thus limiting the algorithmic control flow.
Hence neither numerical nor symbolic differentiation seem suitable for computing the derivatives of high-dimensional functions as we need to calculate when performing gradient based optimisation of the loss function during the training process of a neural network. The state-of-the-art-solution to this problem is known as *backpropagation*. The concept itself was discovered several times independently but found its way into the machine learning community in 1986 (see [16]). We will see in this section that backpropagation is just a special case of automatic differentiation. Unfortunately, the two terms have widely been used interchangeably in the machine learning community, undermining the power of automatic differentiation and preventing the field of machine learning from taking advantage of such. In their survey on automatic differentiation in machine learning [17], the authors aim to resolve this misunderstanding and point out that the benefits machine learning can take from automatic differentiation go far beyond backpropagation.

Automatic differentiation makes use of the fact that every numerical function is just a sequence of elementary arithmetic operations, such as addition, subtraction, multiplication and division, combined by elementary functions, such as exponential, logarithmic or trigonometric functions. Via the chain rule of differentiation, any derivative of this sequence can be represented. Consequently the resulting expression can be evaluated using the known derivatives of the elementary functions.
While automatic differentiation can be executed in many ways, depending on the order in which the chain rule is traversed, the most common ones are referred to as *forward* and *reverse mode* automatic differentiation. In the following we will give an intuitive introduction to the mechanics of both modes, following the same example as presented in [17].

Consider the example function to be given as

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \tag{4.4}$$

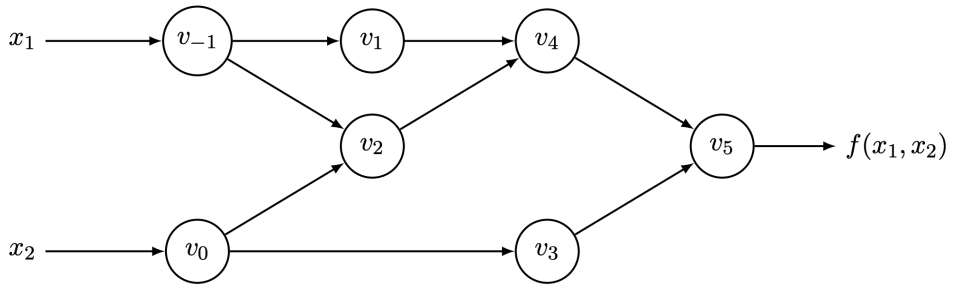


Figure 3: The computational graph of the example function (4.4). The definitions of the variables $v_{-1}, \ldots, v_5$ are given by the primal trace (see Table 5). Taken from [17].

together with its computational graph as shown in Figure 3. Here each node v_i, also called *primal*, represents one elementary operation. From a computational point of view, evaluation of the function f corresponds to evaluating the v_i's one after another according to the computational graph. The corresponding *primal trace*, i.e. the definition of the primals v_i, is given in the first column of Table 5.
In order to compute the derivative of f with respect to x_1 in a given point (x_1, x_2) using forward mode automatic differentiation, we only need to consider the primals together with their derivatives

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1}$$

by using the chain rule. This yields us the *tangent trace*, shown in the second column of Table 5. Note that we can evaluate the primals v_i in lockstep with their corresponding tangents $\dot{v}_i$ starting from the input nodes v_{-1} and v_0 and eventually resulting in the output node

$$\dot{v}_5 = \frac{\partial v_5}{\partial x_1} = \frac{\partial f}{\partial x_1}. \tag{4.5}$$

Hence we can evaluate the derivative with respect to x_1 at any point (x_1, x_2) just by computing the values of the evaluation and its tangent trace. If we were interested in computing the derivatives with respect to x_2, all we need to do is change equation (4.5) accordingly and based on that update the tangent trace.

Primal trace *(forward)*	Tangent trace *(forward)*	Adjoint trace *(reverse)*
$v_{-1} = x_1$	$\dot{v}_{-1} = \frac{\partial v_{-1}}{\partial x_1} = 1$	$\bar{v}_{-1} = \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = \frac{1}{x_1} + x_2$
$v_0 = x_2$	$\dot{v}_0 = \frac{\partial v_0}{\partial x_1} = 0$	$\bar{v}_0 = \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0} = x_1 - \cos(x_2)$
$v_1 = \ln(v_{-1})$	$\dot{v}_1 = \frac{\partial v_1}{\partial x_1} = \frac{1}{x_1}$	$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = 1$
$v_2 = v_{-1} v_0$	$\dot{v}_2 = \frac{\partial v_2}{\partial x_1} = x_2$	$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = 1$
$v_3 = \sin(v_0)$	$\dot{v}_3 = \frac{\partial v_3}{\partial x_1} = 0$	$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = -1$
$v_4 = v_1 + v_2$	$\dot{v}_4 = \frac{\partial v_4}{\partial x_1} = \frac{1}{x_1} + x_2$	$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = 1$
$v_5 = v_4 - v_3$	$\dot{v}_5 = \frac{\partial v_5}{\partial x_1} = \frac{1}{x_1} + x_2$	$\bar{v}_5 = \frac{\partial y}{\partial v_5} = 1$

Table 5: Forward and reverse mode automatic differentiation for the example function (4.4). *Left:* The primal trace of the computational graph (Figure 3). *Center:* The tangent trace used in forward mode automatic differentiation. *Right*: The adjoint trace used in reverse mode automatic differentiation.

This idea generalises to any function $f : \mathbb{R}^n \to \mathbb{R}^m$ allowing to compute the full Jacobian. Note that in the case $n = 1$, all m derivatives can be calculated in one *forward pass*, i.e. one evaluation of the primal and the tangent trace from the input towards the output nodes. However, if $n > 1$, n forward passes are required, one for each input variable x_i.

While in forward mode the tangent trace is evaluated from input to output, the derivatives in reverse mode are propagated backwards from a given output y. To do so we consider the *adjoints*

$$\bar{v}_i = \frac{\partial y}{\partial v_i},$$

i.e. the contributions of the change in each variable v_i towards the change in the output y which yields us the *adjoint trace* as presented in the third column of Table 5. After computing the primal trace from the input to the output

nodes, we can evaluate the adjoint trace backwards, i.e. starting from the output node v_5 and ending up with the input nodes

$$\bar{v}_{-1} = \bar{x}_1 = \frac{\partial y}{\partial x_1}$$
$$\bar{v}_0 = \bar{x}_2 = \frac{\partial y}{\partial x_2}.$$

Hence we can evaluate the derivative of f with respect to both variables x_1 and x_2 in any point (x_1, x_2) in just one *reverse pass*, i.e. one evaluation of the forward primal trace and the reverse adjoint trace.

Generally speaking we note that for a given function $f : \mathbb{R}^n \to \mathbb{R}^m$ forward mode takes n forward passes to compute the full Jacobian while reverse mode takes m reverse passes for the same task. Even though reverse passes are more memory demanding, due to the adverse evaluation of the primal and adjoint traces, reverse mode automatic differentiation is known to be more efficient in cases where $n >> m$. This applies especially to scalar functions with many inputs, i.e. when $m = 1$ and $n >> 1$.

One example of such scalar functions is the loss function used while training a neural network. Here we are interested in the gradient with respect to the network's parameters in order to find an optimal parametrisation θ as discussed in the previous section. Since the loss function is scalar-valued with a large number of parameters, reverse mode automatic differentiation is the optimal choice. As the derivatives are propagated backwards from the output towards the inputs, this is often referred to as backpropagation.
In the context of neural networks, however, automatic differentiation cannot only be used to determine the derivatives of the loss function with respect to the network's parameters in order to find an optimum of the loss function, but also to determine the derivatives of the network with respect to its inputs. This can become quite useful especially when considering neural networks in the setting of partial differential equations as we will see in the next section.

4.1.3. Physics Informed Neural Networks

We will now turn our attention back to the example problem defined in the equations (2.1)–(2.4) involving the one-dimensional Burgers' equation with the given initial and boundary conditions. Remember the involved set of equations

$$\begin{cases} u_t + uu_x = \nu u_{xx}, & \text{(4.6a)} \\ u(x,0) = -\sin(\pi x), & \text{(4.6b)} \\ u(1,t) = u(-1,t) = 0, & \text{(4.6c)} \end{cases}$$

for $x \in [-1,1], t \in [0,1]$ and $\nu = \frac{1}{0.01\pi}$. While in the first part of this work we attempted to remedy the problem via several numerical approaches by representing the occurring derivatives via finite differences, we will now take a different perspective by considering the problem of finding a solution that fulfills the set of equations (4.6) as a regression problem and hereby make use of automatic differentiation.

Let $f(\mathbf{z};\theta)$ denote a neural network with two inputs $\mathbf{z} = (x,t)$ and one output $\hat{u}$ aiming to approximate the true solution u to the example problem. In order to train the neural network with the methods described in section 4.1.1 we will assume for now that we are given some training data $\mathcal{T} = \{u_i, \mathbf{z_i}\}_{i=1}^{N_\mathcal{T}}$. Recall that the universal function approximation theorem tells us that (in theory) we can approximate the solution u to the problem up to any degree of accuracy if $\mathcal{T}$ is sufficiently rich. However, the dynamics of a problem like this can be very complex, thus requiring a lot of training data. Note that at this point it is neither clear how these data points can be obtained nor does the proposed regression approach include the knowledge of the underlying dynamics given by the equations (4.6). Hence a regression approach that is only relying on training data does not seem very promising to solve the example problem.
In order to include the knowledge of the underlying dynamics into the regression model, the authors in [1] propose to apply automatic differentiation to the network's output $\hat{u}$ with respect to the model inputs x and t and then consider the condition

$$g(\hat{u}(\mathbf{z})) = \hat{u}_t(\mathbf{z}) + \hat{u}(\mathbf{z})\hat{u}_x(\mathbf{z}) - \nu\hat{u}_{xx}(\mathbf{z}) = 0, \tag{4.7}$$

which is following from the underlying PDE (4.6a). To evaluate the expression (4.7), we can generate a uniformly distributed set of *collocation points*

$$\mathcal{C} = \{\mathbf{z}_j\}_{j=1}^{N_\mathcal{C}} = \{(x_j,t_j)\}_{j=1}^{N_\mathcal{C}} \sim U(\Omega)$$

and compute the approximate neural network solution in these points, i.e.

$$\hat{u}_j = \hat{u}(\mathbf{z_j}) = f(\mathbf{z_j};\theta), \quad j = 1,\dots,N_\mathcal{C}$$

together with the required derivatives $\hat{u}_t(\mathbf{z_j}), \hat{u}_x(\mathbf{z_j})$ and $\hat{u}_{xx}(\mathbf{z_j})$ via automatic differentiation.

By evaluating equation (4.7) we know how well the solution generated by the network f satisfies the dynamics (4.6a). Consequently this information can be used during model training by including it in the loss function, e.g.

$$L(\mathcal{T}, \mathcal{C}, \theta) = L^{\text{train}}(\mathcal{T}, \theta) + L^{\text{physics}}(\mathcal{C}, \theta).$$

When utilising the mean absolute error as the applied loss function, this corresponds to

$$L(\mathcal{T}, \mathcal{C}, \theta) = \frac{1}{N_{\mathcal{T}}} \sum_{i=1}^{N_{\mathcal{T}}} |f(\mathbf{z_i}; \theta) - u_i| + \frac{1}{N_{\mathcal{C}}} \sum_{j=1}^{N_{\mathcal{C}}} |g(\mathbf{z_j})|.$$

While the collocation points can just be sampled from Ω, the training data can be directly generated from the underlying initial and boundary conditions (4.6b) and (4.6c).
Note that the contribution of $L^{\text{physics}}(\mathcal{C}, \theta)$ towards the overall loss function acts as a regularisation term pushing the network to detect the physical dynamics. At the same time the initial and Dirichlet boundary conditions are included via the training data. It is worth mentioning that also non Dirichlet boundary conditions can easily be encoded via the loss function in the same fashion as the underlying PDE is encoded.

Since the physical dynamics are directly encoded into the loss function of the neural network, the authors named this concept *physics informed neural networks*. While the approach seems quite straightforward at the first glance, it is novel in the sense that automatic differentiation is not only used to push the set of parameters θ towards the optimum of the loss function, but also to encode the derivatives of the network directly into the loss function, allowing to include the underlying dynamics into the model. The late discovery of this possibility might be due to the aforementioned misconception of the terms automatic differentiation and backpropagation.

4.2. A Physics Informed Neural Network Solver

After introducing the underlying theory we will now inspect how a physics informed neural network performs on the example problem in practice. We want to point out that these networks, just as other machine learning methods, in general do not produce a deterministic solution as it was the case for the numerical approaches. Not only are the initial weights and biases of each node usually chosen at random, but also can the training procedure itself make use

Parameter	Value	Note
L	8	Number of hidden layers
N	20	Number of nodes per hidden layer
h	tanh	Activation function
optimiser	Adam	Routine to find optimal parametrisation θ
L^{train}	MAE	Loss function on the training data
L^{physics}	MAE	Loss function on the collocation points
$N_{\mathcal{T}}$	100	Number of training data
$N_{\mathcal{C}}$	10,000	Number of collocation points

Table 6: Network configuration to solve the example problem.

of randomness. In order to obtain reproducible results, the simulations in this section are performed using a predefined seed of the random number generator. The implementation uses the Tensorflow library which provides the necessary algorithmic structures to define and train ANNs via the Keras API. At the same time it offers enough freedom to implement a loss function that operates on the model directly and applies automatic differentiation [18].

4.2.1. Results

As suggested in [1], the neural network was made up of 8 hidden layers with each 20 nodes and a hyperbolic tangent activation function. The $N_{\mathcal{T}}$ training samples were selected, such that 50% of the data came from the initial condition and 25% came from each of the two boundaries. Unlike the authors in [1], we decided to use the mean absolute error as a loss function, as it has been shown to converge faster than the mean squared error. Additionally, we employed the Adam optimiser during model training as opposed to L-BFGS, a quasi-Newton optimisation technique [19]. The set of collocation points $\mathcal{C}$ was generated using a *Latin Hypercube Sampling* strategy [20]. An overview of all the chosen network configurations is given in Table 6.
Since the optimiser will most likely not be able to find the global minimum of the loss function exactly, we need to decide on a criterion on when to stop model training. In order to validate that physics informed neural networks are capable of solving the example problem, we chose to train the network until the solution, evaluated on a discrete grid covering the entire spatio-temporal domain Ω, reaches an error of less than $\varepsilon_{MSE} = 5 \times 10^{-4}$.

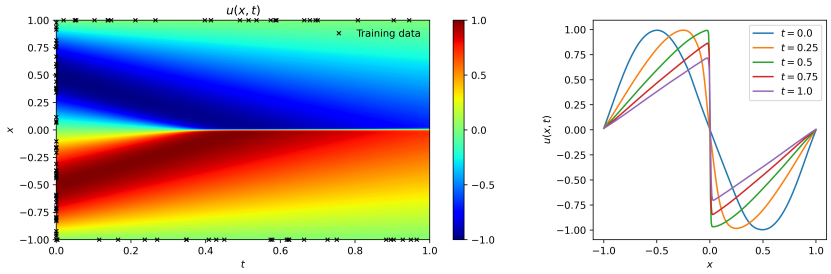


Figure 4: The predicted solution of the physics informed neural network trained until an error of less than $\varepsilon_{MSE} = 5 \times 10^{-4}$ is achieved. *Left*: Contour plot on the entire (x, t)-domain together with the chosen training data from the initial and boundary conditions. *Right*: Snapshots at selected time points.

The computed solution of the physics informed neural network is presented in Figure 4. We can see that the network is able to detect the underlying dynamics and approximate the exact solution reasonably well.
However, there is an important flaw in this procedure: the chosen stopping criterion for the model training requires the knowledge of the exact solution. In a real world problem we would not know the exact solution but we would still like to have a criterion on when to stop model training and that gives us an estimate of the error or at least an idea of the behaviour of the error, similar to the convergence rate for the numerical solvers. To achieve this it is common practice to use a validation set and to stop model training when the error on this validation set increases, which is a sign of overfitting. This concept is known as *early stopping*. However, the procedure is not very applicable to the problem at hand, as the validation set would be chosen from the same domain as the training data, i.e. the initial and boundary conditions, whereas model evaluation should be performed using data points from the entire spatio-temporal domain Ω. Furthermore, overfitting in the classical sense does not seem to be a relevant issue for physics informed neural networks, since we are using noiseless training data from the initial and boundary conditions while the loss on the collocations points acts as a regularisation term.
By taking a closer look at the learning process in the next section, we hope to get a better idea on when to stop model training without using the exact solution.

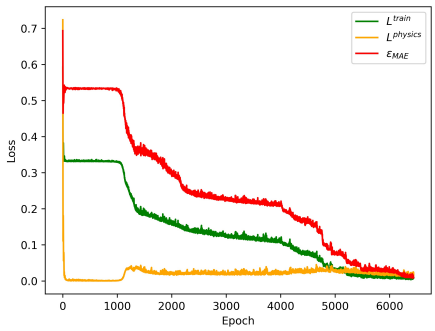


Figure 5: The losses L^{train} and L^{physics} together with the error ε_{MAE} during model training.

4.2.2. Understanding the Learning Process of the Burgers' Equation

In order to better understand the learning process of physics informed neural networks we can inspect the behaviour of the error together with the two loss functions, namely the loss on the training data L^{train} and the loss on the collocation points L^{physics}, as presented in Figure 5.

We can make two crucial observations:

1. The loss on the collocation points L^{physics} drops very quickly close to zero, indicating that the method does not struggle to find a solution that follows the imposed dynamics given by equation (4.6a). A further investigation of the predicted solution in the beginning of the training process shows that the network finds the trivial zero solution as shown in Figure 6, where the solution after 50 epochs is presented.

2. The behaviour of the error ε_{MAE} seems very similar to the behaviour of the loss on the training data L^{train} which indicates that the loss on the training data is a reasonable estimate of the error, and therefore can be used as a stopping criterion.

Note that the second observation is not trivial. First, a low loss on the collocation points does not necessarily imply that the error is mainly originating from the training data, since the loss on the collocation points is measured differently as the error and the loss on the training data. Second, the training data is not chosen from the entire spatio-temporal domain Ω but only from the initial and boundary conditions.

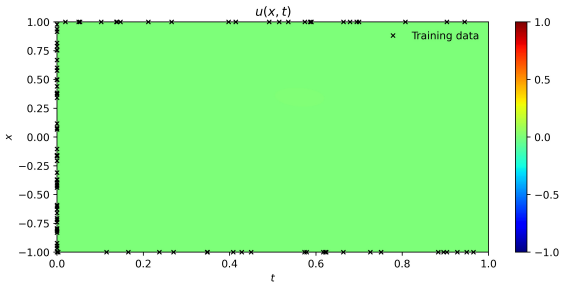


Figure 6: Contour plot of the detected trivial solution to the Burgers' equation after 50 training epochs.

Moreover, we can observe that the error and the losses are very stable within the first 1000 epochs. This can likely be explained by the fact that the optimiser of the loss function hits a strong local optimum at this point. As mentioned in section 4.1.1, this is a very typical behaviour when training neural networks, though unwanted.

In order to provide further evidence that the method itself does not struggle to detect the underlying dynamics, we can take the predicted initial conditions of the neural network solver at an earlier stage, e.g. after 3000 epochs, where the training error is still large, and use them as initial condition for a numerical solver. Under the hypothesis that the physics informed neural network detected the correct dynamics of the problem, the numerical approach should yield results that are very similar. Following this idea we choose a second order upwind approach with $H = 1280$ and $k = 1 \times 10^{-4}$ as the numerical solver. The predicted solution of the physics informed neural network after 3000 epochs and the second order upwind solution using the predicted initial condition of this network are shown in Figure 7.

We can clearly see that both approaches compute very similar solutions. The ℓ^2 difference between the two solutions is $\varepsilon = 1.07 \times 10^{-2}$. This result shows that the physics informed neural network is indeed able to detect the governing dynamics and mostly suffers from the error made on the training data.

Following up on this observation it is worth to take a closer look at the training data. Remember that we decided to choose the training data only from the initial and boundary conditions. If we look at the problem at hand, we have a simple Dirichlet boundary condition $u(x, t) = 0$ for $x \in \{-1, 1\}$ and $t \in [0, 1]$. Due to its simplicity we would expect that most of the complexity in the

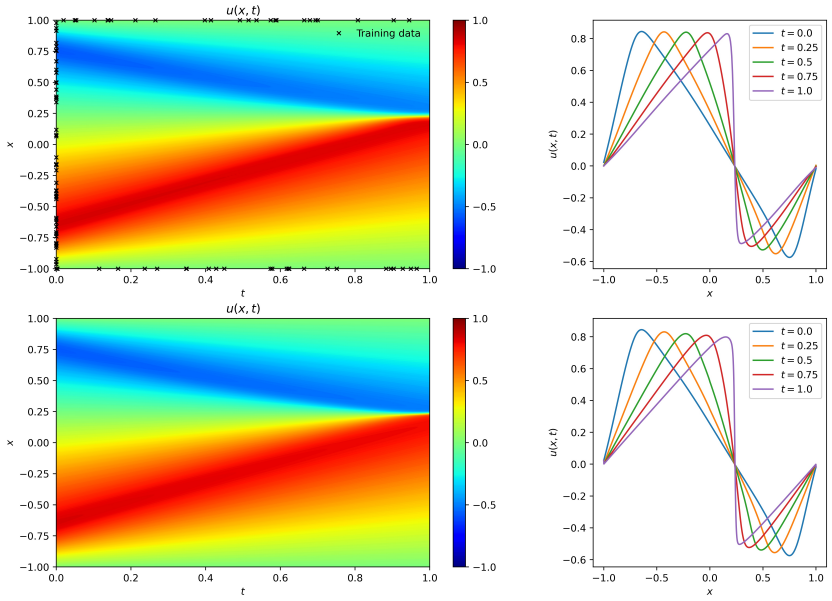


Figure 7: A comparison of a physics informed neural network and a second order upwind solution using the same faulty initial data. *Top*: Neural network prediction, *Bottom*: Second order upwind approximation. *Left*: Contour plot on the entire (x, t)-domain together with the chosen training data from the initial and boundary conditions. *Right*: Snapshots at selected time points.

training data comes from the initial condition, which is given by

$$u(x, 0) = -\sin(\pi x), \quad x \in [-1, 1].$$

If we split the loss on the training data L^{train} even further into the loss obtained on the initial and boundary conditions, L^{initial} and L^{boundary}, respectively, we see that similar to the loss on the collocation points, the loss on the boundary data also drops quickly close to zero, see Figure 8. Moreover, the overall error seems to behave very similar to the loss on the initial data. These observations suggest that the overall model performance is strongly influenced by the performance at the initial condition.

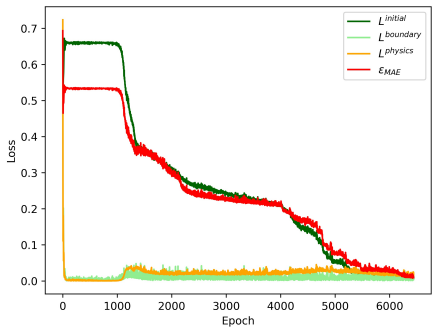


Figure 8: The losses L^{initial}, L^{boundary} and L^{physics} together with the error ε_{MSE} during model training.

4.2.3. The Importance of the Initial Condition

To further investigate the importance of the initial condition when finding an approximate solution to a PDE using a physics informed neural network, we will now take a look at another example, i.e. the one-dimensional Allen-Cahn equation [21], given as

$$u_t - 0.0001u_{xx} + 5u^3 - 5u = 0, \quad x \in [-1, 1], \quad t \in [0, 1] \tag{4.8}$$

together with the following initial and boundary conditions

$$\begin{cases} u(x, 0) = x^2 \cos(\pi x) & \text{(4.9a)} \\ u(-1, t) = u(1, t) = 1. & \text{(4.9b)} \end{cases}$$

The exact solution to this problem is presented in Figure 9.

When attempting to solve this problem with a physics informed neural network using the same configurations as for the Burgers' equation, see Table 6, we observe that the predicted solution converges quickly to the one presented in Figure 10, which clearly does not coincide with the exact solution (Figure 9). At the same time both losses are considerably small, i.e. $L^{\text{train}} = 1.4 \times 10^{-3}$ and $L^{\text{physics}} = 5.8 \times 10^{-4}$, indicating that not only the physics but also the initial data are approximated well.

To explain this contradictive behaviour let us take a closer look at the underlying PDE (4.8) and the exact solution (see Figure 9). We can see that for all x where the initial condition is negative, i.e. $u(x, 0) < 0$, the solution decreases

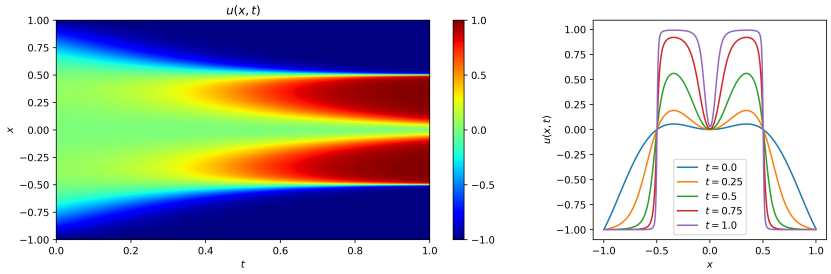


Figure 9: Exact solution to the Allen-Cahn equation (4.8) with the initial and boundary conditions (4.9). *Left:* Contour plot on the entire (x, t)-domain. *Right:* Snapshots at selected time points.

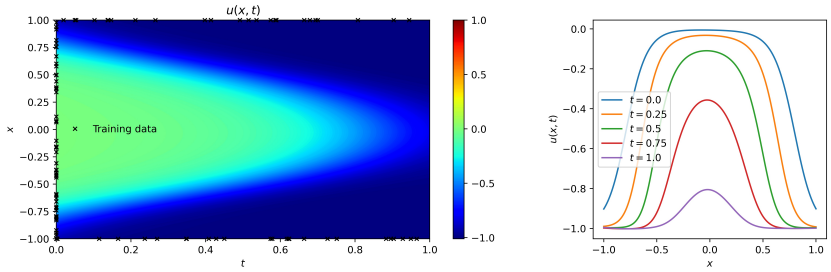


Figure 10: Physics informed neural network solution to the Allen-Cahn equation (4.8) with the initial and boundary conditions (4.9) after 1000 epochs. *Left:* Contour plot on the entire (x, t)-domain together with the chosen training data from the initial and boundary conditions. *Right:* Snapshots at selected time points.

over time. Contrary for all x where $u(x, 0) > 0$, the solution increases over time. Considering further that for the initial condition we have

$$\begin{aligned} 0 \leq u(x, 0) \leq \delta \quad & \text{for } |x| \leq 0.5 \\ u(x, 0) < 0 \quad & \text{for } |x| > 0.5, \end{aligned}$$

and δ relatively small, it can be noticed that a small inaccuracy in the prediction of the initial condition (4.9a) can have a substantial impact on the detected dynamics on the entire domain Ω, i.e. when the initial condition $u(x, 0)$ for $|x| \leq 0.5$ is incorrectly predicted to be negative. Considering again the predicted initial condition of the physics informed neural network, i.e. the snapshot for $t = 0$ in Figure 10, we observe that the solution in the area where $|x| < 0.5$ is indeed incorrectly predicted to be negative. The important thing to

notice here is that this prediction error has only a small impact on the loss on the training data L^{train}, as the true initial condition in this area is just slightly above zero. However, the dynamics following this predicted initial condition are completely different.
To be sure that the error in the predicted solution is related to the incorrect prediction of the initial data and not to the detected dynamics, we can plug the predicted initial data into a numerical solver once again. Under the assumption that the detected dynamics are correct, the numerical solver should generate a similar solution as the physics informed neural network did. As a numerical solver we choose an FTCS approach, similar to the one in section 3.1. For more details on the method and a plot of the numerical solution of the FTCS approach using the predicted initial condition of the physics informed neural network, see Appendix A. Both approaches compute very similar solutions with an ℓ^2 difference of $\varepsilon = 5.37 \times 10^{-2}$, indicating that the disability of the physics informed neural network to find a solution to the Allen-Cahn equation is indeed related to the fact that the initial condition for $|x| < 0.5$ is incorrectly predicted to be negative.

The observations made in this section suggest that the method of physics informed neural networks does not struggle to detect the dynamics imposed by a nonlinear PDE but rather suffers from an incorrect prediction of the initial condition. While the Burgers' equation could still be approximated reasonably well, we observed for the Allen-Cahn equation that a small discrepancy between the predicted and the exact initial condition can cause the physics informed neural network to generate a solution that looks very good, i.e. that yields a small error on both training and collocation points, but that turns out to be quite different from the desired solution. This result is of considerable importance, as it reveals a major drawback of physics informed neural networks. At the same time it provides substantial information on how to improve the method of physics informed neural networks for finding approximate solutions to PDEs.

5. Conclusion and Future Work

We started this work with the goal to study to what extent methods derived within the field of machine learning are applicable to solve one of the main problems of numerical analysis, namely the approximate solution of a PDE given some initial and boundary conditions.

While considering the one-dimensional Burgers' equation as an example problem for a nonlinear PDE, we started out by attempting to solve this problem using several finite difference approaches since the theory of such methods is well established.
This approach did not only provide us with a reasonable benchmark for a machine learning solver but also unveiled some of the main drawbacks when using numerical methods. We saw that on the one hand, we need to make sure that the method produces stable solutions. On the other hand, we need to be aware that discontinuities are in general hard to resolve, as they are likely to produce unwanted oscillations in the numerical approximation, and thus might require a special treatment.
In the context of the example problem those issues could be quite easily resolved by using an implicit representation of the diffusion term and an upwind one for the convection term. It is important to notice though that firstly the introduction of an implicit term can drastically increase the computational effort which can become problematic when the considered domain Ω is large. Secondly the introduction of the upwind scheme requires an understanding where the discontinuity is coming from, and is therefore a problem specific approach. Even though the theory of numerical methods to solve the example problem or similar ones is very well studied, it can be said that many solutions are tailored to the underlying problem, thus lacking generalisability. This motivated us to study the degree to which machine learning, in particular physics informed neural networks, can be utilised to solve such problems.

As for the Burgers' equation we observed that the method was able to generate an approximate solution with an accuracy comparable to the previous methods. While the numerical approaches are prone to introduce oscillations around the discontinuities, the physics informed neural network does not seem to struggle at all to detect such phenomena being an important advantage of the machine learning approach.
Our further investigations, however, revealed a dependence between the overall model performance and the correct detection of the initial condition. For the Burgers' equation, this impacted mainly the efficiency of the method but not

the accuracy whereas for the Allen-Cahn equation, the method predicted a solution that is substantially different from the desired solution even though the error on the training data L^{train} and the error on the collocation points L^{physics} indicated a good approximation. This is a major drawback of physics informed neural networks since there is no way to see that the detected solution is not the desired one.

In terms of computational effort we want to point out that our experiments indicated that the machine learning approach is significantly more expensive than the numerical solvers. Though it should be noted that it is not straightforward to compare the effort of the different methods, as the neural network approach is not deterministic.

Lastly, it is worth mentioning that a physics informed neural network does not require any discretisation of the spatio-temporal domain Ω, and therefore generates a continuous solution, since we can obtain a prediction $\hat{u}(\mathbf{z})$ at any $\mathbf{z} = (x, t)$. On the contrary, the finite difference approximation is only discrete and a continuous solution can only be obtained by using interpolation.

To conclude, physics informed neural networks, and probably machine learning approaches in general, cannot yet replace conventional numerical methods in finding approximate solutions to PDEs, as the theory of the latter has been studied and developed for many years. Our investigations, however, showed that to some extent it is possible to solve such problems starting from a machine learning perspective as well. The fact that such solvers can be implemented without any concerns about stability and oscillations, as opposed to numerical methods, is a major advantage and should contribute to the further development of the field. Together with the observed dependence between the overall model performance and the performance on the initial condition, this opens up many options for future research.

As a first step following up on these observations, it would be interesting to better understand to what extent problems with a bistable nature, like the Allen-Cahn equation, can be solved using physics informed neural networks. Moreover, considering the lower computational efficiency of physics informed neural networks, it would be of great interest to investigate further the degree to which the training process can be accelerated by using different network architectures, activation functions and optimisers. Especially the latter seems promising since, as long as the network is sufficiently rich, it is not a question whether the loss function of the neural network has a minimum at which both parts of the error, i.e. L^{train} and L^{physics}, approach zero. The main challenge might very likely lie in finding this minimum. One could study for example to

what extent methods from evolutionary computing can be applied to this task as opposed to the Adam and the L-BFGS optimiser.
Lastly, the idea of enforcing the underlying dynamics into the model's loss function by applying automatic differentiation with respect to the model's input, could also be transferred to other regression techniques, e.g. gradient boosted trees.
Since the field of solving PDEs using machine learning is just evolving, exciting results can be expected in the future.

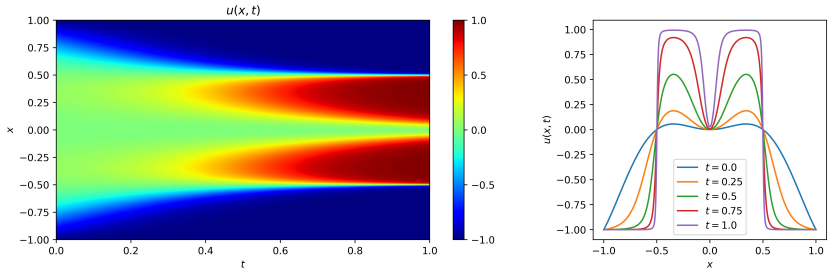


Figure 11: FTCS solution to the Allen-Cahn equation (4.8) using $H = 511$ and $K = 200$ with an error of $\varepsilon = 1.2 \times 10^{-2}$. *Left:* Contour plot on the entire (x, t)-domain. *Right:* Snapshots at selected time points.

A. The FTCS Solver for the Allen-Cahn Equation

Consider the following initial and boundary value problem

$$\begin{cases} u_t - 0.0001 u_{xx} + 5u^3 - 5u = 0 & \text{(A.1a)} \\ u(x, 0) = x^2 \cos(\pi x) & \text{(A.1b)} \\ u(-1, t) = u(1, t) = 1 & \text{(A.1c)} \end{cases}$$

with $x \in [-1, 1], t \in [0, 1]$, involving the one-dimensional Allen-Cahn equation as discussed in Section 4.2.3.

We aim to find an approximate solution to the problem (A.1) by using an explicit FTCS finite difference approach. The occurring derivatives in the point (x_j, t^n) are approximated as follows:

$$u_{xx} = u_{xx}(x_j, t^n) = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}$$
$$u_t = u_t(x_j, t^n) = \frac{u_j^{n+1} - u_j^n}{k}$$

with h and k being the mesh size and the step size, respectively. Hence the FTCS approximation reads as

$$u_j^{n+1} = u_j^n + k \left[0.0001 \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2} - 5(u_j^n)^3 - 5u_j^n \right].$$

The approximate solution of this solver using a relatively coarse discretisation with $H = 511$ and $K = 200$ has an error of $\varepsilon = 1.2 \times 10^{-2}$ and is presented in

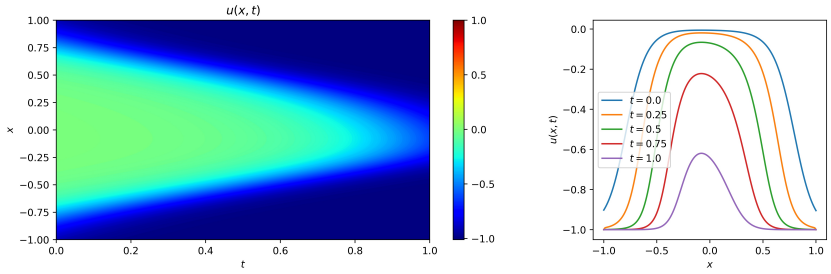


Figure 12: FTCS solution to the Allen-Cahn equation (4.8) using the predicted initial condition of the neural network solver after 1000 epochs. *Left:* Contour plot on the entire (x, t)-domain. *Right:* Snapshots at selected time points.

Figure 11.[6]

When using the initial condition predicted by the physics informed neural network, as described in section 4.2.3, the FTCS approach computes the solution shown in Figure 12, which is very similar to the solution of the physics informed neural network with an ℓ^2 difference of $\varepsilon = 5.37 \times 10^{-2}$.

[6] To evaluate the error, the exact solution to this problem on the same grid can be downloaded from `https://github.com/maziarraissi/PINNs`.

References

[1] M. Raissi, P. Perdikaris, and G. E. Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. 2017. arXiv: 1711.10561.

[2] H. Bateman. "Some Recent Researches on the Motion of Fluids". *Monthly Weather Review* 43 (1915), pp. 163–170.

[3] J. Burgers. "A Mathematical Model Illustrating the Theory of Turbulence". *Advances in Applied Mechanics* 1 (1948), pp. 171–199.

[4] C. Basdevant et al. "Spectral and finite difference solutions of the Burgers equation". *Computers & Fluids* 14.1 (1986), pp. 23–41.

[5] T. Vo-Duy. *Note on One Dimensional Burgers Equation*. 2018.

[6] R. Courant, E. Isaacson, and M. Rees. "On the solution of nonlinear hyperbolic differential equations by finite differences". *Communications on Pure and Applied Mathematics* 5 (1952), pp. 243–255.

[7] C. Hirsch. *Numerical Computation of Internal and External Flows*. Butterworth Heinemann, 2007.

[8] S. Kutluay, A. Bahadir, and A. Özdeş. "Numerical solution of one dimensional Burgers equation: explicit and exact-explicit finite difference methods". *Journal of Computational and Applied Mathematics* 103.2 (1999), pp. 251–261.

[9] E. Miller. *Predictor-corrector studies of Burger's model of turbulent flow*. University of Delaware, M.S. Thesis. 1966.

[10] R. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[11] C. Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. arXiv: 1811.03378.

[12] C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. 2018.

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

[14] S. Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747.

[15] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980.

[16] D. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". *Nature* 323 (1986), pp. 533–536.

[17] A. G. Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767.

[18] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467.

[19] D. C. Liu and J. Nocedal. "On the Limited Memory BFGS Method for Large Scale Optimization". *Mathematical Programming* 45 (1989), pp. 503–528.

[20] M. Stein. "Large Sample Properties of Simulations Using Latin Hypercube Sampling". *Technometrics* 29.2 (1987), pp. 143–151.

[21] S. M. Allen and J. W. Cahn. "A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening". *Acta Metallurgica* 27.6 (1979), pp. 1085–1095.