Solving partial differential equations with neural networks

FYS-STK4155 – Project 3

Stjepan Salatovic
github.com/stipesal/FYS-STK4155

December 5, 2021

Abstract

This study focuses on using machine learning for solving partial differential equations. More specifically, we use fully connected feedforward neural networks trained using Adam and a loss function given by the differential equation. The loss function is constructed using a finite set of training data coupled with problem-dependent physics, initial, and boundary conditions. The results suggest, that the neural networks outperform established numerical solvers on coarse grids, while providing a grid-free parameterization of the solution. We confirm the results using the 1D heat and linear advection equations. Finally, we apply the described methods to linear eigenvalue problems.

Contents

1 Differential equations

Differential equations are deeply rooted in almost every scientific field and even govern some of the most fundamental laws inherent to them. Phenomena in physics, chemistry, biology, and economics are often modeled by a highly complicated system of ordinary and partial differential equations, for which explicit solutions are only available in a few exceptional cases. Consequently, the use of sophisticated numerical methods is inevitable.

1.1 Scientific machine learning

Machine learning techniques, and especially artificial neural networks, have proven to be groundbreaking in the fields of computer vision and natural language processing, where extremely complex relationships between predicting variables and target variables were successfully modelled. These relationships were already used by Lagaris et al. in 1998 [2], where differential equations were reformulated in form of optimization problems, by replacing the unknown solution with a neural network using the universal approximator property of them. Optimization problems of this vein have the advantage of providing grid-free, differentiable, and closed-form analytical solutions. The approach we use, is similar to the methods described by Raissi et al. in 2019 [3]. They introduced *physics informed neural networks* (PINNs), that is neural networks that are trained to solve supervised learning tasks while respecting any given law of physics described by partial differential equations. On the other hand, Ricky et al. (2019) [1] used the theory of numerical methods for differential equations to introduce a new class of neural networks successfully. This leads to the thought, that there is a deep connection between the iterative procedure of both, neural networks and numerical methods for differential equations.

1.2 The abstract PDE

In this subsection we will introduce partial differential equations (PDE). In principle, all methods presented in the following can be applied to ordinary differential equations (ODE). In this report we restrict ourselves to one-dimensional (nonlinear) time-dependent PDEs of the form

$$\begin{aligned} \mathbf{u}_t &= \mathbf{L}\left(\mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}\right), \\ \mathbf{u}(x,0) &= \mathbf{u}_0(x), \\ \mathbf{L}_b(X_l,t) &= \mathbf{u}_l(t), \\ \mathbf{L}_b(X_r,t) &= \mathbf{u}_r(t), \end{aligned}$$

where $\mathbf{u} : [X_l, X_r] \times [0,T] \longrightarrow \mathbb{R}^m$ is the vector of unknowns. The right side $\mathbf{L}$ is a (nonlinear) differential operator involving the first and second spatial derivative of u. The initial condition is given by the function $\mathbf{u}_0 : [X_l, X_r] \longrightarrow \mathbb{R}^m$. $\mathbf{L}_b$ describes an operator defining the boundary conditions. Strictly speaking, the actual PDE is given only in the first of the above equations. Only in connection with the equations that follow, which are necessary for uniqueness of a solution, it becomes a so-called initial boundary value problem (IBVP).

Heat equation

The heat equation is the prototypical parabolic PDE. It describes the diffusion of some quantity (such as heat) over time in a solid medium (such as a rod). We consider it in one space dimension with homogeneous Dirichlet boundary conditions

$$\begin{aligned} u_t &= c u_{xx}, \\ u(x,0) &= u_0(x), \\ u(0,t) &= u(1,t) = 0, \end{aligned}$$

where $(x,t) \in (0,1) \times (0,T)$ and $c > 0$ denotes the diffusion coefficient. So we are looking for a function u in two variables, space x and time t, which satisfies all the above equations (i. e. solves the IBVP).

Exact solution

One can give solutions even in closed formulas as follows. It is easy to verify that the solution to the heat equation is given by

$$u(x,t) = \sum_{n=1}^{\infty} b_n \sin(n\pi x) e^{-cn^2\pi^2 t},$$

where b_n for $n \in \mathbb{N}$ are the Fourier coefficients of the initial condition

$$b_n = 2 \int_0^1 u_0(x) \sin(n\pi x)\, dx.$$

Figure 1 down below shows the exact solution to the heat equation $u(x,t)$ for $c = 1$, $T = 0.5$, and initial data $u_0(x) = \sin(\pi x)$ at different points in time. One can see the basic structure of the underlying physics. The larger the time level t the lower the temperature $u(x,t)$ in the medium and it homogenizes to the value zero.

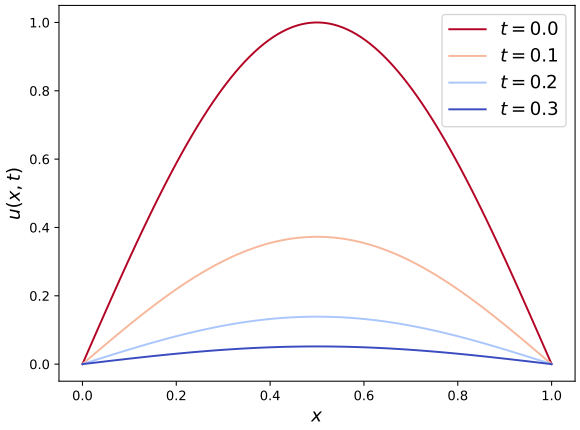


Figure 1. Solution to the heat equation for $c = 1$, $T = 0.5$, and intial condition $u_0(x) = \sin(\pi x)$ at different time points.

Furthermore, Figure 21 in the Appendix shows two solutions of the heat equation for two different diffusion coefficients $c = 0.2$ and $c = 1$ and with same initial data $u_0(x) = \sin(\pi x)$. We note that, the higher the diffusion coefficient $c > 0$ the faster the temperature $u(x,t)$ drops with time, making the heat IBVP stiffer and thus more complicated for solvers to deal with.

Maximum principle

The physical self-evidence that heat does not arise from nothing is mathematically reflected in the so-called maximum principle:

$$\max_{(x,t)\in\Gamma} u(x,t) = \max_{(x,t)\in\Omega} u(x,t)$$

where $\Omega = [X_l, X_r]\times[0,T]$ denotes the total domain and $\Gamma = \{(x,t) \in \Omega : t = 0 \vee x = X_l \vee X_r\}$ the three-sided boundary. The maximum value (over time and space) of the temperature $u(x,t)$ is to be found either at the beginning of the considered time interval or at the edge of the considered space region.

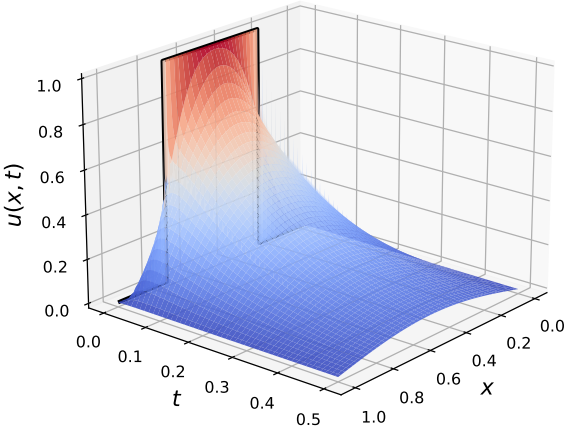


Figure 2. Solution of the heat equation with $c = 0.5$, $T = 0.5$, and rough initial condition $u_0(x) = \mathbb{1}_{\{x\in(0.25,0.75)\}}$. The initial data is highlighted as a black line. A Crank-Nicolson method paired was used to calculate the approximations on a fine grid consisting of $J = 1000$ and $N = 1000$ points in space and time.

Smoothing property

Another interesting property is that even if u has a discontinuity point at time $t = t_0$, the function u is continuous in space at any time $t > t_0$. Thus, if two rods of different temperature are firmly joined at $t = t_0$, the average temperature will abruptly adjust at the junction and the temperature curve will be continuous through both rods. This can be seen exemplary in the following Figure 2. Depicted is the solution to the heat equation for the non-smooth initial condition $u_0(x) = \mathbb{1}_{\{x\in(0.25,0.75)\}}$ and a relatively low diffusion coefficient of $c = 0.5$. One observes a smoother, continuous temperature immediately after $t > 0$.

Advection equation

Advection describes the transport of a substance or quantity. The linear advection equation is a prototypical example of a hyperbolic PDE and is used to design efficient numerical methods for similar type of problems. We consider it in one space dimension

with periodic boundary conditions

$$\begin{aligned} u_t + cu_x &= 0, \\ u(x,0) &= u_0(x), \\ u(0,t) &= u(1,t), \end{aligned}$$

where $(x,t) \in (0,1) \times (0,T)$ and $c > 0$ denotes the propagation speed of the wave. Physically, the initial condition is transported with constant propagation speed c to the right, since $c > 0$, and to the left for $c < 0$.

Exact solution

One can easily verify that $u(x,t) = u_0(x-ct)$ solves the actual advection equation (i. e.shifted initial data u_0), but the periodic boundary conditions are not satisfied. Nevertheless, it should be clear here that also an exact solution of the above IVBP, i.e. with periodic boundary conditions, can be found by requiring periodicity of the initial condition u_0 on $[X_l, X_r]$. For simplicity, in this report we only compute an approximation using a suitable solver on a sufficiently fine grid.

Figure 3 shows the approximation to the advection equation $u(x,t)$ for $c = 2$, $T = 0.5$, and rough initial data $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$ at different points in time.

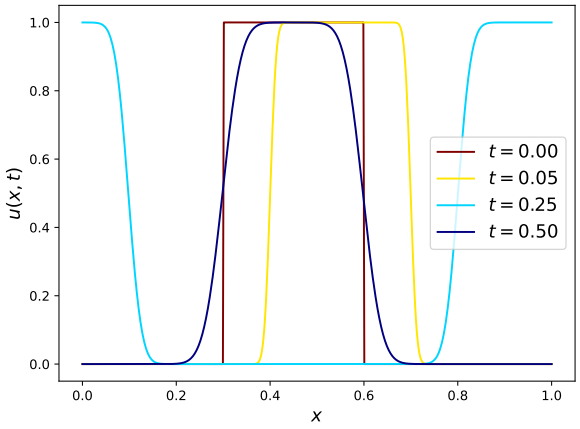


Figure 3. Approximation to the advection equation for $c = 2$, $T = 0.5$, and initial condition $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$ at different time points.

One can see the initial condition u_0 shifting with time at speed $c > 0$. Also the periodic boundary conditions at $x = 0$ and $x = 1$ can be seen well: Each curve has the same value at $x = 0$, as at $x = 1$. Nevertheless, the figure is a little deceptive, since the exact solution would always be discontinuous, like the initial condition. The dark blue curve at time $T = 0.5$ should actually agree with the initial condition, i.e. the brown curve at time $t = 0$. However, one would need a much finer grid to represent this with a numerical method. This approximated solution was calculated using a standard finite difference method with 1000 points in both, space and time.

Figure 23 in the Appendix shows two solutions to the advection equation for two different speed coefficients $c = 0.5$ and $c = 2$ and with same initial data $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$. We note that, the higher the speed coefficient $c > 0$ the faster the wave shifts, making the advection IBVP stiffer and thus more complicated for solvers to deal with.

Conservation

The wave (or the substance) that propagates according to the advection equation, transports all of its properties, too. Those properties that are carried with the advected substance are conserved properties such as energy. For instance, we have that

$$\int_{X_l}^{X_r} u(x,t_1)\,dx = \int_{X_l}^{X_r} u(x,t_2)\,dx$$

for arbitrarily $t_1, t_2 \in (0,T)$. Numerical methods, when used in physics for instance, are often designed in such a way that they preserve these properties. The numerical methods used in Section 2.2 preserve this property too.[1]

[1] This is a great way to test ones code.

2 Finite difference method

The finite difference method (FDM) is a very popular numerical method used for solving both, ordinary and partial differential equations. The idea is to reduce the continuous differential equation to a system of discrete difference equations, a process known as discretization. This is achieved by approximating the (nonlinear) spatial derivatives with difference quotients at finitely many grid points. Depending on the difference scheme, the resulting linear system of equations often has a sparse band structure, which is exploited and can be solved efficiently using highly optimized methods.

For a general finite-difference method the discretization is carried out as follows. For simplicity, a uniform, i. e. equidistant, discretization of space $[X_l, X_r]$ and time $[0, T]$ is used. For space we use $J+2$ equidistant grid points $x_j = j\Delta x$ for $j = 0, \dots, J+1$ with mesh size $\Delta x = \frac{1}{J+1}$. Similarly, for time we use $N+1$ equidistant grid points $t_n = n\Delta t$ for $n = 0, \dots, N$ with time step $\Delta t = \frac{T}{N}$. With $\mathbf{U}_j^n \approx \mathbf{u}(x_j, t^n)$ we denote the approximated solution in grid point x_j at time level t_n. Further, $\mathbf{U}^n = \{\mathbf{U}_j^n\}_{j=0}^{J+1}$ refers to the vector of approximate solutions at the time level t^n.

2.1 Heat equation

To solve the heat equation on the introduced grid, we consider the parametrized three-point finite difference scheme

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \frac{c\,(1-g)}{\Delta x^2}\left(U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}\right) + \frac{c\,g}{\Delta x^2}\left(U_{j-1}^{n} - 2U_j^{n} + U_{j+1}^{n}\right),$$

with a parameters $g \in \mathbb{R}$. Setting $g = 1$ yields a purely explicit method in time, while setting $g = 0$ yields an purely implicit method.[2] The scheme is designed in such a way, that it embeds several well known standard finite difference schemes, which we present below. For this, let the CFL[3] number

$$\alpha = \frac{c\Delta t}{\Delta x^2},$$

where $c > 0$ is the diffusion coefficient of the heat equation.

[2] Note that the method is implicit as soon as $g \neq 0$.

[3] Richard Courant, Kurt Friedrichs, Hans Lewy (1928)

FTCS

The first one is the FTCS (Forward Time Central Space) method, which is the most basic scheme to approximate the heat equation in one dimension. It is revived by setting $g = 1$ in the above general scheme. It utilizes explicit (or forward) Euler in time and a second-order central difference quotient for the spatial derivative. The scheme is given by

$$U_j^{n+1} = U_j^n + \alpha\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right).$$

To understand how the stencil acts in the space-time domain, an illustration is given in Figure 4. One can show, that the error of this method is of

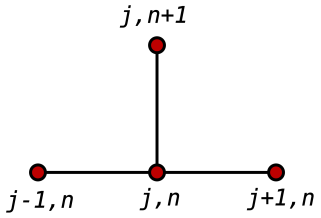


Figure 4. Illustration of the FTCS stencil. Index (j, n) corresponds to U_j^n.

order $\mathcal{O}(\Delta t) + \mathcal{O}\left(\Delta x^2\right)$ and is numerically stable (hence convergent), if and only if the CFL condition $\alpha \leq \frac{1}{2}$ is satisfied. If $\alpha > \frac{1}{2}$ then the method is numerically unstable leading to exploding results.

BTCS

The second method is the BTCS (Backward Time Central Space), which is the implicit version of FTCS. It uses the implicit (or backward) Euler in time and the same second-order central difference in space, resulting in the same order $\mathcal{O}(\Delta t) + \mathcal{O}\left(\Delta x^2\right)$. It is revived by choosing $g = 0$, and the resulting procedure is given by

$$U_j^{n+1} = U_j^n + \alpha\left(U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}\right).$$

Figure 5 illustrates the corresponding stencil. The advantage of the BTCS method compared to FCTS is the numerical stability, which is preserved for every time step $\Delta t > 0$. The disadvantage, however, are the computational extra costs, solving an implicit equation in every time step.

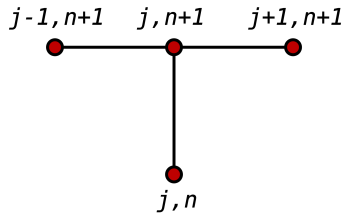


Figure 5. Illustration of the BTCS stencil. Index (j, n) corresponds to U_j^n.

CTCS

The third method, CTCS (Crank-Nicolson Time Central Space), uses the same second-order central difference quotient for the spatial derivative, but averages the results of explicit and implicit Euler in time.[4] Setting $g = 1/2$ yields the CTCS method

$$U_j^{n+1} = \frac{\alpha}{2}U_{j-1}^n + (1-\alpha)U_j^n + \frac{\alpha}{2}U_{j+1}^n + \frac{\alpha}{2}U_{j-1}^{n+1} - \alpha U_j^{n+1} + \frac{\alpha}{2}U_{j+1}^{n+1}.$$

As with BTCS, the CTCS method is numerically stable and convergent with a higher order of error $\mathcal{O}\left(\Delta t^2\right) + \mathcal{O}\left(\Delta x^2\right)$. Working half implicit and half explicit in time, this method uses six grid points in total, in order to calculate the next value, which is depicted as a stencil in Figure 6.

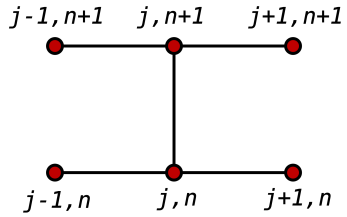


Figure 6. Illustration of the CTCS stencil. Index (j, n) corresponds to U_j^n.

Matrix notation

The general scheme presented for the heat equation is equivalent to the matrix version

$$LU^{n+1} = MU^n,$$

where

$$M = I + \alpha g B,$$
$$L = I + \alpha(g-1)B,$$

with

$$B = \begin{pmatrix} -2 & 1 & & 0 \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ 0 & & 1 & -2 \end{pmatrix} \in \mathbb{R}^{(J+2)\times(J+2)}.$$

All of the methods presented were implemented using this matrix notation coupled with highly optimized linear solvers for sparse systems.

Comparison of the standard methods

In Table 1 one can see the error comparison of the standard schemes induced by the three-point FDM for the heat equation, that is FTCS, BTCS, and CTCS. For this experiment we use $c = 1$, $T = 0.5$, and the initial condition $u_0(x) = \sin(\pi x)$. Further we use $J = 100$ points in space, and choose $N \in \mathbb{N}$ such that the CFL condition $\alpha \leq 1/2$ is fulfilled for the FTCS scheme.[5] This results in $N = 10201$ points in time. The solutions associated with this table are shown as 2D images in Figure 22 in the Appendix.

The errors are all of the same order of magnitude and one can actually say that all three methods perform equally well. This can be seen from the errors in the Table down below as well as in Figure 22. Also, the thermal coefficient $c > 0$ is not too large, so the explicit method FTCS does not encounter any problems due to a too stiff PDE. The opposite is true actually, as FTCS performs minimally better than the two implicit methods BTCS and CTCS.

2.2 Advection equation

To solve the advection equation on the introduced grid, we consider the parametrized two-point finite difference scheme

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \frac{c\,(g-1)}{\Delta x}\left(U_j^{n+1} - U_{j-1}^{n+1}\right) - \frac{c\,g}{\Delta x}\left(U_j^n - U_{j-1}^n\right),$$

[4] Also known as the *implicit trapezoidal method*.

[5] To be more precise, we choose $N = \lceil 2cT(J+1)^2 \rceil$.

Table 1. Comparison of the standard schemes. Experiment conducted with $c = 1$, $T = 0.5$, and $u_0(x) = \sin(\pi x)$. Further, $J = 100$ points in space and $N \in \mathbb{N}$ points in time such that CFL condition is met.

Method	Error
FTCS	5.781×10^{-5}
BTCS	5.875×10^{-5}
CTCS	5.828×10^{-5}

with a parameter $g \in \mathbb{R}$. Again, setting $g = 1$ yields a purely explicit method in time, while setting $g = 0$ yields an purely implicit method. Also, the scheme is designed in such a way, that it embeds several well known standard finite difference schemes, which we present below. Similar to the heat equations the CFL number

$$\alpha = \frac{c\Delta t}{\Delta x},$$

where $c > 0$ is now the speed coefficient of the advection equation.

Upwinding

All of the methods resulting from the general two-point scheme are so-called upwind differencing schemes. Upwinding is often used with hyperbolic partial differential equations, or in particular, when the underlying physical problem contains some directional bias. The side, where the wave (or substance) propagates to, is called downwind side, and the opposite side is called upwind side. A finite difference scheme is called an upwind scheme, simply if the quotient for the spatial derivative contains more points in the upwind side, than in the downwind side. That is why the point U_{j+1} is "missing" in the upper two-point scheme compared to the three-point one for the heat equation. An upwind finite difference scheme is therefore also sort of directional biased.

FTBS

The first one is the so-called FTBS (Forward Time Backward Space) scheme, which is known as the first-order forward upwind method for advection equation. It uses the explicit (or forward) Euler in time and the first-order upwind difference quotient for approximating the first spatial derivative. By setting $g = 1$, the scheme is given by

$$U_j^{n+1} = U_j^n - \alpha \left(U_j^n - U_{j-1}^n \right).$$

One can show, that this method provides an error of order $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x)$ and is numerically stable (hence convergent), if and only if the CFL condition $\alpha = \frac{c\Delta t}{\Delta x} \leq 1$ is satisfied. Figure 7 illustrates the corresponding stencil in space and time, where one can clearly sees the upwind finite difference quotient.

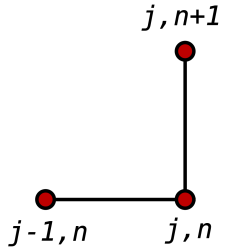


Figure 7. Illustration of the FTBS stencil. Index (j, n) corresponds to U_j^n.

BTBS

The second method is the BTBS (Forward Time Backward Space) scheme, which is the implicit version of FTBS. It uses the implicit Euler in time and the same first-order upwind difference in space, resulting also with an error-order of $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x)$. It is revived by changing $g = 0$ resulting in the scheme

$$U_j^{n+1} = U_j^n - \alpha \left(U_j^{n+1} - U_{j-1}^{n+1} \right).$$

Figure 8 illustrates the stencil which works implicit in time. The advantage of the BTBS method compared to FTBS method is, again, the numerical stability, which is preserved for every time step $\Delta t > 0$.

CTBS

The third method, CTBS (Crank-Nicolson Time Backward Space), uses the same first-order upwind difference for the spatial derivative as FTBS and

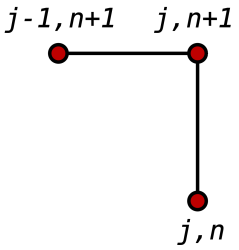


Figure 8. Illustration of the BTBS stencil. Index (j, n) corresponds to U_j^n.

BTBS, but uses the mean of the implicit and explicit Euler in time. Setting $g = 1/2$ recovers the scheme

$$U_j^{n+1} = U_j^n - \frac{\alpha}{2}\left(U_j^{n+1} - U_{j-1}^{n+1} + U_j^n - U_{j-1}^n\right).$$

This method is numerically stable and is convergent with an error of order $\mathcal{O}\left(\Delta t^2\right) + \mathcal{O}\left(\Delta x\right)$. Working half implicit and half explicit in time, but upwind in space, this method uses four grid points in total, in order to calculate the next value, which is depicted as a stencil in Figure 9.

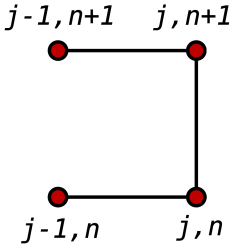


Figure 9. Illustration of the CTBS stencil. Index (j, n) corresponds to U_j^n.

Matrix notation

As with the heat equation, we will give the matrix equations necessary for our implementation for the two-point FDM. One can check that

$$LU^{n+1} = MU^n$$

with

$$M = I - \alpha g B,$$
$$L = I + \alpha(1-g)B,$$

where

$$B = \begin{pmatrix} 1 & 0 & \cdots & 0 & -1 \\ -1 & 1 & \ddots & & 0 \\ & \ddots & \ddots & \ddots & \vdots \\ & & -1 & 1 & 0 \\ 0 & & & -1 & 1 \end{pmatrix} \in \mathbb{R}^{(J+2)\times(J+2)},$$

is equivalent to introduced two-point scheme for the advection equation. Note the structure of the matrix B (and therefore also L and M) which is almost tridiagonal. The last entry in the first row of B is necessary for the periodic boundary conditions. Again, all the methods were implemented by using this matrix version.

Comparison of the standard methods

In Table 2 one can see the error comparison of the standard schemes induced by the two-point FDM for the advection equation, that is FTBS, BTBS, and CTBS. For this experiment we use $c = 2$, $T = 0.5$, and the initial condition $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$. Further we use $J = 500$ points in space, and choose $N \in \mathbb{N}$ such that the CFL condition $\alpha \leq 1$ is fulfilled for the FTBS scheme.[6] This results in $N = 1002$ points in time. The solutions associated with this table are shown as 2D images in Figure 24 in the Appendix.

For this experiment, FTBS performs best with an error of 1.012×10^{-3}, which can be observed also in Figure 24 if one looks very closely. Nevertheless the errors are of the same order of magnitude and one can actually say that again, all three methods perform somewhat equally well.

Table 2. Comparison of the standard schemes. Experiment conducted with $c = 2$, $T = 0.5$, and $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$. Further, $J = 500$ points in space and $N \in \mathbb{N}$ points in time such that CFL condition is met.

Method	Error
FTBS	1.012×10^{-3}
BTBS	5.806×10^{-3}
CTBS	3.459×10^{-3}

[6] To be more precise, we choose $N = 2\lceil cT(J+1)\rceil$.

3 Which neural networks did we study?

In this Section, we introduce the neural networks that we will use to solve (partial) differential equations. In particular, we will discuss the difference to the neural networks we have learned and worked with so far. We use the notation introduced in the second report for artificial neural networks.

The main goal

Recall the abstract PDE from Section 1.2, that is

$$\begin{aligned} \mathbf{u}_t &= \mathbf{L}\left(\mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}\right), \\ \mathbf{u}(x,0) &= \mathbf{u}_0(x), \\ \mathbf{L}_b(X_l,t) &= \mathbf{u}_l(t), \\ \mathbf{L}_b(X_r,t) &= \mathbf{u}_r(t). \end{aligned}$$

The goal now is to find a neural network $g \in \mathcal{F}(L,p)$ with a fixed, but arbitrary, network architecture (L,p), that replaces the unknown solution of the IBVP above. We would like to adjust the weights $\theta \in \mathbb{R}^{D(L,p)}$ of the network in such a way that the network solves the PDE, the initial, and the boundary conditions as well as possible.

3.1 Network architecture

Given a number of hidden layers $L \in \mathbb{N}_0$ and hidden layer sizes $p = (p_0, \ldots, p_{L+1}) \in \mathbb{N}^{L+2}$, recall that a neural network $g \in \mathcal{F}(L,p)$ is nothing but a function

$$g : \mathbb{R}^{p_0} \longrightarrow \mathbb{R}^{p_{L+1}}.$$

Let $p_0 = 2$ and $p_{L+1} = m$, since the goal is to find the solution of the PDE

$$\mathbf{u} : [X_l, X_r] \times [0,T] \subset \mathbb{R}^2 \longrightarrow \mathbb{R}^m.$$

3.2 Training and test data

An important question that should have been asked by now is the one about the existence of training data. The answer is: there is none. We sample randomly uniformly distributed training data on the space-time domain of the PDE as

$$(x,t) \sim \mathcal{U}\left([X_l, X_r] \times [0,T]\right).$$

The training (test) data set $\mathcal{F}$ consists then of simply $N \in \mathbb{N}$ of such space-time data pairs.

3.3 A suitable loss function

Since there are three conditions to meet in order to solve an IVBP, we also use three components in the loss function. Each of the components measures the quality of the neural network with respect to the underlying physics (i.e. the PDE), the initial conditions, or the boundary conditions. In the following, we abbreviate by $g = g_\theta(x,t)$ the output of the network g equipped with parameters $\theta \in \mathbb{R}^{D(L,p)}$ when applied to $(x,t) \in [X_l, X_r] \times [0,T]$. The loss for the underlying physics, that is the actual PDE, is given by the squared error

$$L_{physics}(\theta) = \sum_{(x,t)\in\mathcal{D}_{physics}} \left(g_t - \mathbf{L}(g, g_x, g_{xx})\right)^2.$$

The error loss for the violation of the initial condition can be measured by the squared error

$$L_{init}(\theta) = \sum_{(x,t)\in\mathcal{D}_{init}} \left(g - \mathbf{u}_0(x)\right)^2.$$

Similarly, the loss for the boundary condition can be calculated by

$$L_{bound}(\theta) = \sum_{(x,t)\in\mathcal{D}_{bound}} \left(g - u_B(t)\right)^2.$$

The total loss function is then given by the sum of all the three components

$$L(\theta) = L_{physics}(\theta) + L_{init}(\theta) + L_{bound}(\theta).$$

Now here a form of stochastic gradient descent can be used to approximate

$$\theta^* = \underset{\theta\in\mathbb{R}^{D(L,p)}}{\arg\min}\ L(\theta).$$

Also when solving PDEs, extensive hyperparameter tuning for the learning rate, batch size, and number of epochs is required. As in Project 2, regularization can be introduced, but we refrain from doing so in this work, as we were never able to overfit to the data. Note that each loss function needs its own training (and test) data set, since e.g.

$$\mathcal{D}_{physics} = \{(x,t) \in (x,t) \in [X_l, X_r] \times [0,T]\}$$

while

$$\mathcal{D}_{init} = \{(x,t) \in (x,t) \in [X_l, X_r] \times \{0\}\}.$$

3.4 Implementation

For the implementation, we simply extended Pytorch's *nn.Module* class to a simple *ANN* (artificial neural network) class. This is basically a feedforward fully connected neural network as we theoretically introduced in Project 2. Subclasses called *PDENet* and *EigenNet* then inherit from *ANN* to solve PDEs or linear eigenvalue problems. So the only thing needed is to write a subclass (e.g. *HeatNet*) that inherits from *PDENet* and implement an associated loss function for the IBVP, i.e. which measures PDE, initial and boundary loss goodness. Then everything can be trained with PyTorch's optimization algorithms. Certainly, an even more abstract, and object-oriented *PDENet* could have been formulated, but this would go beyond the technical scope of this work.

The following Python Code 1 is an example of how to solve the heat equation with the HeatNet. The API is always based on that of common machine learning libraries.

```
import numpy as np
import torch
from neural_pde import HeatNet
from torch import nn

# Initialize neural net.
p = [2, 50, 50, 1]
activation = nn.Tanh
heat_net = HeatNet(p, activation)

# Heat equation data.
c = 1
space = (0, 1)
T = 0.5
u0 = lambda x: torch.sin(np.pi * x)
heat_net.set_problem(c, u0, space, T)

# Train neural net.
n_epochs=100
batch_size=128
lr=1e-2
heat_net.train(n_epochs, batch_size, lr)
```

Code 1. Example of solving the heat equation with *HeatNet* in Python.

4 Data & Results

In this Section, we present the results obtained using the neural network to solve the heat and advection equation in comparison to established standard solvers. Also, we will look at a particular type of differential equation that allows to determine eigenvalues of a real and symmetric matrix.

For all of the following PDE experiments we use $(X_l, X_r) = (0, 1)$ and $T = 0.5$. For the networks we always use the tangent hyperbolicus activation function and training (test) data set size of $N = 1000$ ($N = 100$).

4.1 Heat equation

Let the diffusion coefficient be $c = 1$ and the intial data $u_0 = sin(\pi x)$, such that the exact solution of the heat equation is given by

$$u(x, t) = sin(\pi x)e^{-\pi^2 t}.$$

A grid with $J = 100$ points in space and $N = 100$ points is time is used to compute an approximation with the standard solver BTCS, that is implicit Euler coupled with a central, second order approximation to the second spatial derivative. For the neural network we choose a relatively simple architecture of $L = 2$ hidden layers and layer sizes $p = (2, 50, 50, 1)$, since, as described above, the unknown solution is a function

$$\mathbf{u} : [X_l, X_r] \times [0, T] \subset \mathbb{R}^2 \longrightarrow \mathbb{R}^m.$$

The networks is trained using PyTorch's Adam optimizer with 100 epochs, a batch size of 128, and a learning rate of 1×10^{-2}. Figure 10 shows the loss $L(\theta)$ over the epochs for train and test data. As expected, the loss is dropping considerably in time (note the logarithmic scale), as a tuning for all of the hyperparameters was performed.

Table 3 shows the error of the BTCS method compared against the one achieved by the PDENet. The error is calculated by taking the mean squared error of the difference between prediction and exact solution over all the grid points. We can see that the BTCS method performs slightly better with 1.191×10^{-4}, against the PDENet with 2.571×10^{-4}.

A qualitative comparison can be seen in the next three Figure 11, 12, and 13. The 2D projection

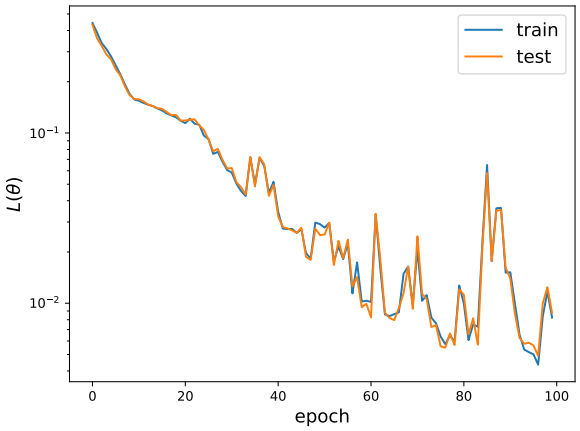


Figure 10. Loss $L(\theta)$ over the epochs for train (blue) and test (orange) data.

Table 3. Comparison of BTCS and the PDENet for solving the heat equation.

Method	Error
BTCS	1.191×10^{-4}
PDENet	2.571×10^{-4}

on the space-time plane in Figure 11 shows that both methods, BTCS and PDENet, provide qualitatively equivalent results.

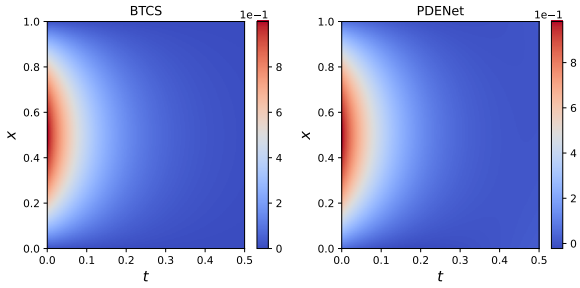


Figure 11. Prediction of the BTCS (left) and PDENet (right) for the grid given by $J = N = 100$ points in both, space and time.

A more revealing comparison is given by Figure 12. There, the difference between the prediction and the exact solution is shown in a 3D plot. It can be observed that the PDENet has problems to catch the PDE, especially at the initial time, and generally has difficulties with the Dirichlet boundary conditions. The consistent BTCS solver also provides a more consistent solution.

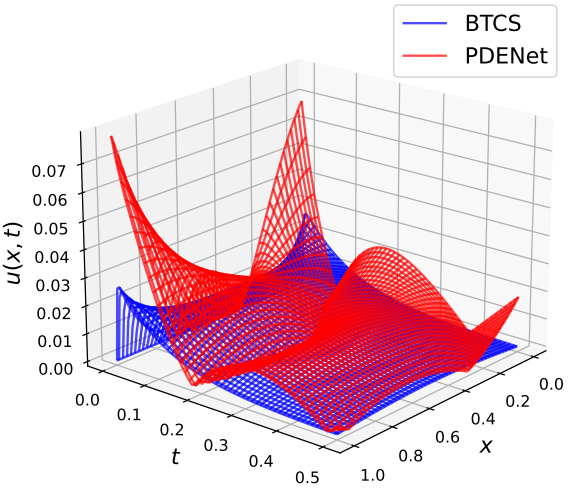


Figure 12. Error between the prediction and the exact solution for the BTCS (blue) and PDENet (red) as 3D wireframe plot.

Figure 13 shows a 1D projection onto the space-temperature plane at different times, namely $t = [0, 0.1, 0.2, 0.3]$. One can clearly see that the PDENet has difficulty in satisfying the initial and boundary condition. However, it must also be said that the FDM solvers start with the fixed initial condition, and thus have a "head start".

4.2 Advection equation

Let the wave speed be $c = 2$ and the intial data $u_0(x) = \mathbb{1}_{\{x \in (0.3, 0.6)\}}$. A grid with $J = 500$ points in space and $N = 500$ points is time is used to compute an approximation with the standard solver BTBS, that is implicit Euler coupled with a first order upwind for approximating the spatial derivative. For the neural network we use one hidden layer more, that is $L = 3$, and layer sizes $p = (2, 50, 50, 50, 1)$. The networks is trained using an Adam optimizer with 200 epochs, a batch size of 128, and a learning rate of 5×10^{-3}. Figure 14 shows the loss $L(\theta)$ over the epochs for train and test data.

Table 4 shows the error of the BTBS method compared against the one achieved by the PDENet.

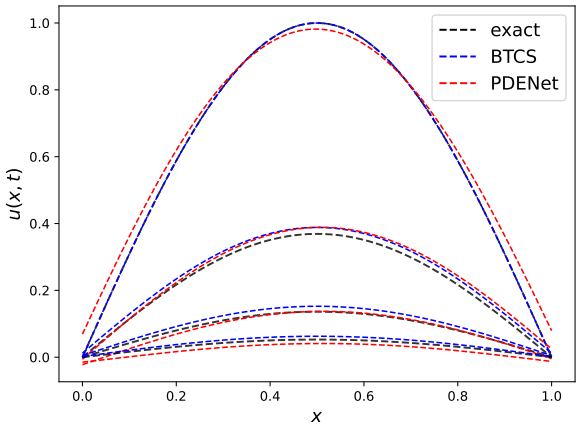


Figure 13. Comparison between the exact solution (black), and the approximations obtained by the BTCS (blue) and PDENet (red) at the time points $t = [0, 0.1, 0.2, 0.3]$. The grid used consists of $J = N = 100$ points in both, space and time.

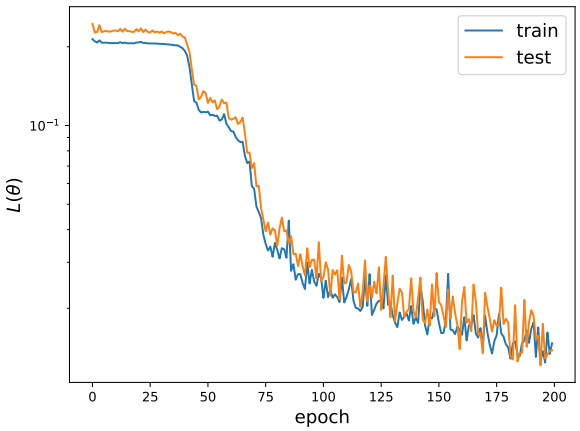


Figure 14. Loss $L(\theta)$ over the epochs for train (blue) and test (orange) data.

The error is calculated by taking the mean squared error of the difference between prediction and exact solution over all the grid points. We can see that this time the neural PDENet performs slightly better with 5.338×10^{-3}, against the standard BTBS with 1.076×10^{-2}.

A qualitative comparison can be seen in the next three Figure 15, 16, and 17. The 2D projection on the space-time plane in Figure **??** shows that both methods, BTBS and PDENet, provide qualitatively equivalent results, although one could argue that

Table 4. Comparison of BTBS and the PDENet for solving the advection equation.

Method	Error
BTBS	1.076×10^{-2}
PDENet	5.338×10^{-3}

the PDENet has a slightly sharper boundary for the secound (incoming) wave.

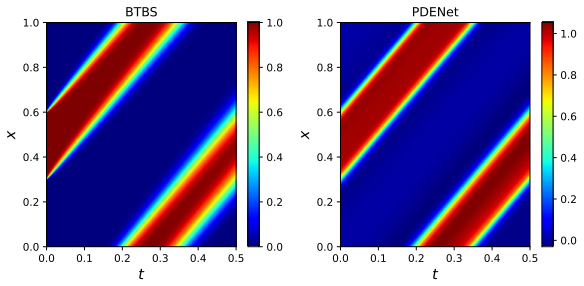


Figure 15. Prediction of the BTBS (left) and PDENet (right) for the grid given by $J = N = 500$ points in both, space and time.

A more revealing comparison is given by Figure 16. There, the difference between the prediction and the exact solution is shown in a 3D plot. It can be observed that both of the methods, BTBS and PDENet, have the same problems capturing the edges of the moving rough initial condition.

Figure 17 shows a 1D projection onto the space-temperature plane at different times, namely $t = [0.1, 0.4]$. One can clearly see that the PDENet even over and undershots the initial condition. This is not the case for a consistent solver such as the BTBS method.

4.3 Eigenvalue problem

In this Subsection we try to replicate the work of Yi et. al (2004) [4] using our networks, namely to determine eigenvalues of a matrix. In the following, we briefly describe the basics of the idea and the goal of the work.

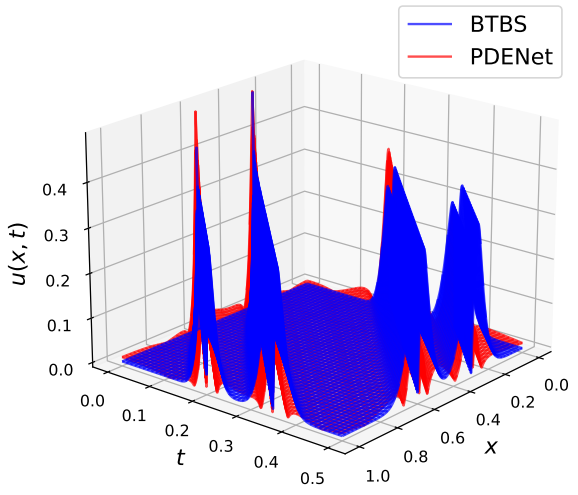


Figure 16. Error between the prediction and the exact solution for the BTBS (blue) and PDENet (red) as 3D wireframe plot.

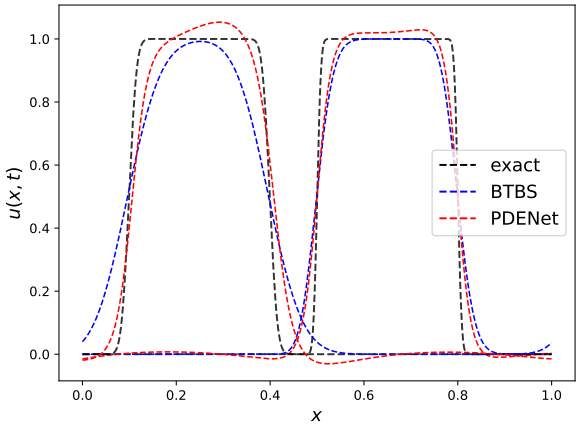


Figure 17. Comparison between the exact solution (black), and the approximations obtained by the BTBS (blue) and PDENet (red) at the time points $t = [0.1, 0.4]$. The grid used consists of $J = N = 500$ points in both, space and time.

Setting

For this, let $A \in \mathbb{R}^{n\times n}$ be a symmetric matrix. Then set up the ODE

$$x_t = f(x)$$

with $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ and

$$f(x) = \left[x^\top xA - x^\top Ax\right] x.$$

So for an arbitrary, but fixed, symmetric matrix $A \in \mathbb{R}^{n\times n}$ we get an ODE, whose solution, as we will see, gives information about the eigenvalues (actually eigenvectors) of A.

The supporting theorem

Let the set of equilibrium points be

$$E = \{\xi \in \mathbb{R}^n : f(\xi) = 0\}.$$

Further, denote by $\sigma(A)$ the set of all eigenvalues of A, and by V_λ the eigenspace corresponding to the eigenvalue $\lambda \in \sigma(A)$. Then the following theorem holds:

$$E = \bigcup_{\lambda\in\sigma(A)} V_\lambda.$$

This means that (non-trivial) equilibria and eigenvectors are equivalent in this problem. Further they show that for every initial value $x(0) = x_0 \in \mathbb{R}^n \setminus \{0\}$ the solution $x : \mathbb{R} \longrightarrow \mathbb{R}^n$ of the ODE converges to an equlibirum, or eigenvector. Moreover, convergence always happens to the eigenvector belonging to the largest eigenvalue λ_1 of A if the initial value $x_0 \in \mathbb{R}^n$ is not orthogonal to the eigenspace V_{λ_1}.

Experiments

Using a real, symmetric matrix $Q = \mathbb{R}^{6\times6}$, we generate a symmetric matrix with $A = \frac{1}{2}(Q + Q^\top)$. So the goal is to solve the above ODE with the neural network. We use $L = 2$ hidden layers with 50 neurons each. The structure of the solution of the above ODE $x : \mathbb{R} \longrightarrow \mathbb{R}^6$ completes the network architecture $p = (1, 50, 50, 6)$. An initial condition $x_0 = (1, \ldots, 1) \in \mathbb{R}^6$ and an end time of $T = 1$ were chosen. The network $g \in \mathcal{F}(L, p)$ is trained for 20 training epochs, with a batch size of 128 and a learning rate of 1×10^{-3} using Pytorch's Adam optimizer. Training takes approximately 0.35 seconds and the resulting loss curve is depicted in the following Figure 18. Note that, as with the *PDENets*, training and test data are randomly generated on $[0, T]$. The very smooth, briskly falling loss curves are striking, suggesting that the problem is well posed, so to speak, and can

be well optimized. After only 20 training epochs, a sufficiently small loss is recorded. The actual result

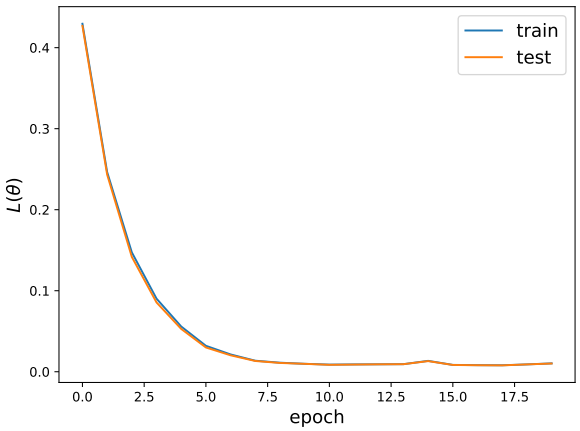


Figure 18. Loss $L(\theta)$ over the epochs for train (blue) and test (orange) data.

can be seen in the next Figure 19. You can see the output of the network $g(T)$ at the end time $T = 1$ in each epoch, respectively. The values of the 6 output eurons are shown in color, while the entries of the true eigenvector corresponding to the largest eigenvalue of the matrix $A \in \mathbb{R}^{6\times 6}$ are shown as black dashed lines. It can be seen that convergence occurs early and that the *EigenNet* represents the largest eigenvector very well. By the way, given a

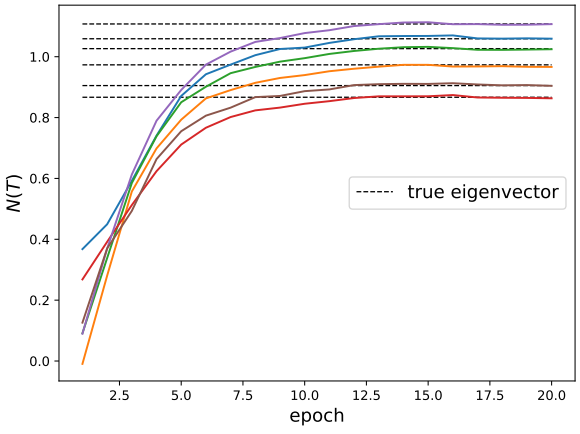


Figure 19. Output of the neural network $g(T)$ at the end time $T = 1$ for every of the 20 epoch (color) and the exact eigenvector corresponding to the largest eigenvalue (black dashed lines).

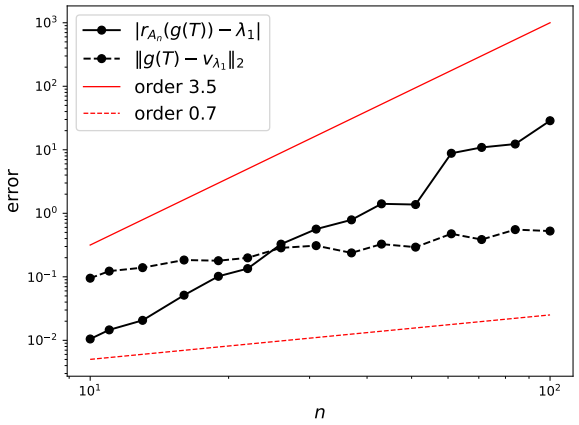


Figure 20. Errors $|R_{A_n}(g(T)) - \lambda_1|$ (solid black) and $\|g(T) - v_{\lambda_1}\|_2$ (dashed black) against the matrix size $n \in \mathbb{N}$. Approximated orders are shown in solid and dashed red.

matrix $A \in \mathbb{R}^{n\times n}$, the Rayleigh quotient

$$R_A(x) = \frac{x^\top Ax}{x^\top x}$$

can be used to determine the eigenvalue λ to an available eigenvector $x \in \mathbb{R}^n$ of the matrix A. The exact largest eigenvalue of A in this case is 3.0908 and the *EigenNet* yields 3.0907.

Error scaling with $n \in \mathbb{N}$

To investigate how the error scales with the dimension $n \in \mathbb{N}$ of the matrix $A \in \mathbb{R}^{n\times n}$, we perform the following experiment. We use 15 logarithmic placed matrix sizes n between 10 and 100 and create a matrix $A_n \in \mathbb{R}^{n\times n}$ for each. Using a neural network of architecture $p = (1, 100, 100, 100, n)$, the *EigenNet* is then trained using $T = 5$, $x_0 = (1, \dots, 1) \in \mathbb{R}^n$, 50 training epochs, with a batch size of 256 and a learning rate of 1×10^{-3} for each n. The error $|R_{A_n}(g(T)) - \lambda_1|$, as well as the deviation $\|g(T) - v_{\lambda_1}\|_2$ to the true eigenvalue v_{λ_1} of the respective matrix A_n is shown in Figure 20. The approximated orders of $n^{3.5}$ (solid red) and $n^{0.7}$ (dashed red) are also plotted. Thus, the deviation to the eigenvalue is of order $\mathcal{O}(n^{3.5})$, which is actually very bad. However, one must also add that the (hidden) network architecture p is independent of n and should fairly scale with it.

5 Discussion

After presenting two prototypical examples of PDEs, the heatequation and the advection equation, we introduced the discretization method of FDMs as possible solvers. This included three standard methods for each of the two problems, whose advantages, disadvantages, and performance were compared. Subsequently, we introduced neural networks that are able to solve PDEs by constructing a certain loss function structure, which respects underlying physics, as well as initial and boundary conditions. Theoretical as well as practical, especially implementation, aspects were considered. The results showed that neural networks of this type can be used as solvers for differential equations, although traditional solvers are in some ways more consistent. For example, the general two-point scheme for the advection equation conserves energy by construction, which is obviously not the case with the *PDENet*. On the other hand, the neural network has the considerable advantage of representing a real (grid-free) parameterization of the solution of a differential equation. In particular, when the grids are not too fine, neural networks can hold their own with traditional solvers, if not outperform them. Another application of the performance of these networks was demonstrated in solving a linear eigenvalue problem. In further work, we would like to focus on training data generation in particular, as we think that the most freedom and potential is hidden there. Since there is no fixed training data available, the data could also be regenerated every epoch to better represent the space-time domain of the PDE and possibly get better results. Also, quasi-random numbers with low discrepancy can be used to obtain a more homogeneous training process, cf. quasi Monte Carlo.

References

[1] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG].

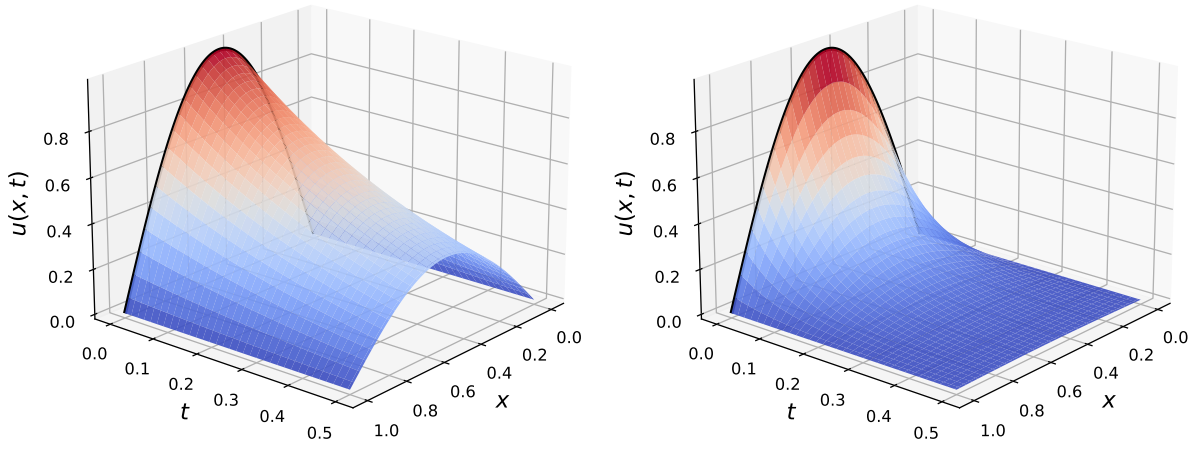
[2] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. ISSN: 1045-9227. DOI: 10.1109/72.712178. URL: http://dx.doi.org/10.1109/72.712178.

[3] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[4] Zhang Yi, Yan Fu, and Hua Jin Tang. "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix". In: *Computers Mathematics with Applications* 47.8 (2004), pp. 1155–1164. ISSN: 0898-1221. DOI: https://doi.org/10.1016/S0898-1221(04)90110-1. URL: https://www.sciencedirect.com/science/article/pii/S0898122104901101.

6 Appendix

Heat equation



(a) $c = 0.2$ **(b)** $c = 1$

Figure 21. Exact solution of the heat equation for two different values for the diffusion coefficient $c > 0$ and with initial condition $u_0(x) = \sin(\pi x)$. The grid consists of $J = 100$ and $N = 100$ points in space and time.

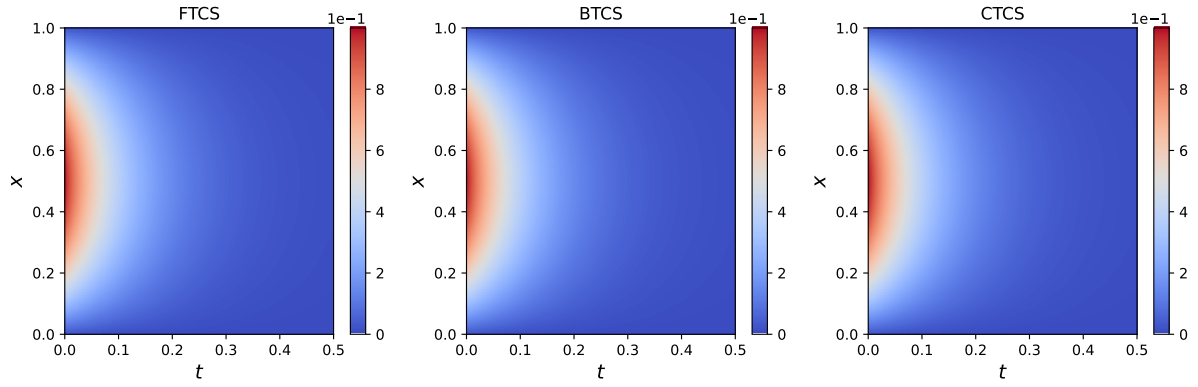
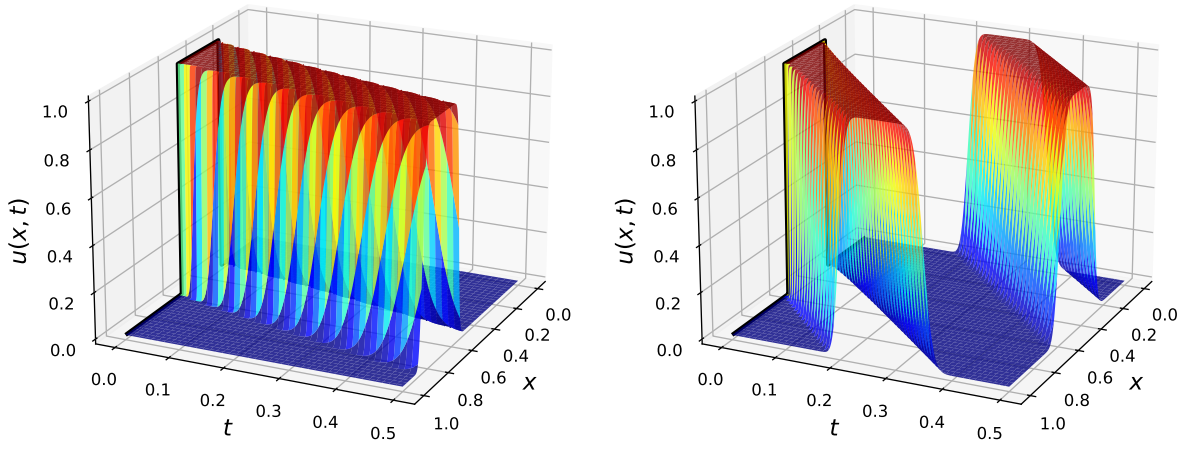


Figure 22. Comparison of the approximation to the heat equation by the standard methods resulting from the three-point FDM. We use $c = 1$, $T = 0.5$, and the initial condition $u_0(x) = \sin(\pi x)$. Further, $J = 100$ points in space, and $N \in \mathbb{N}$ such that the CFL condition $\alpha \leq 1/2$ is met for the FTCS scheme, which results in $N = 10201$ points in time.

Advection equation



(a) $c = 0.5$ **(b)** $c = 2$

Figure 23. Approximated solution to the linear advection equation for two different values for the speed coefficient $c > 0$ and with initial condition $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$. A forward Euler paired with first order upwinding (i. e. FTBS) was used to calculate the approximations on a fine grid consisting of $J = 500$ points in space, and $N \in \mathbb{N}$ points in time, such that the CFL condition $\alpha \leq 1$ is met. This results in $N = 252$ points in time for $c = 0.5$ (a) and $N = 1002$ points in time for $c = 2$ (b).

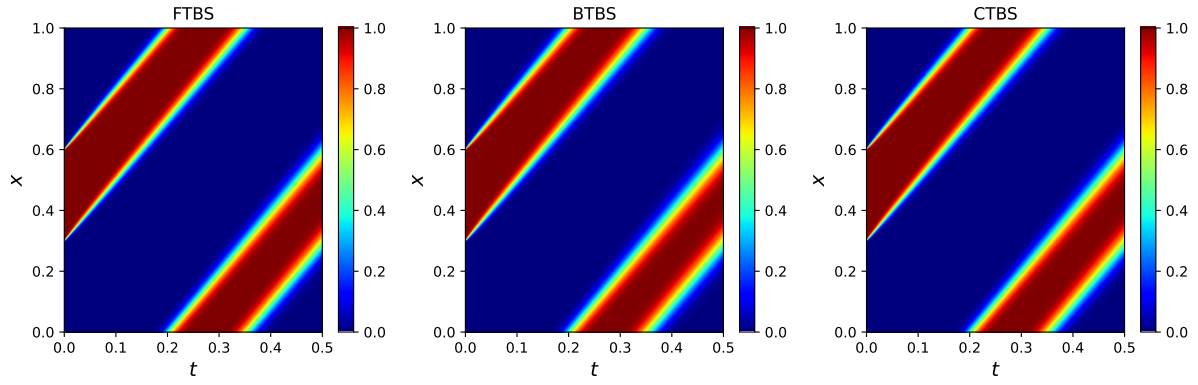


Figure 24. Comparison of the approximation to the advection equation by the standard methods resulting from the two-point FDM. We use $c = 2$, $T = 0.5$, and the initial condition $u_0(x) = \mathbb{1}_{\{x\in(0.3,0.6)\}}$. Further, $J = 500$ points in space, and choose $N \in \mathbb{N}$ such that the CFL condition $\alpha \leq 1$ is met for the FTBS scheme, which results in $N = 1002$ points in time.