Solving Partial Differential Equations With the Use of Neural Networks

Achraf Atifi
Department of Geosciences, University of Oslo

https://github.com/achat97/FYS-STK4155

(Dated: December 17, 2023)

The one-dimensional heat equation with given boundary and initial conditions is solved using a finite difference method and a neural network. The finite difference method used is the forward time-centered space method (FTCS). The neural network we have trained that gave the best approximation has six layers with 500 neurons each, with the ELU function as the activation. The FTCS method was used for two different step sizes in space, $\Delta x = 0.1$ and $\Delta x = 0.01$. The step size in time was given as $\Delta t = (\Delta x)^2/2$. The smaller step size required 100 times more steps in time to reach $t = 1$. This is also what gave the best overall error, including the neural network. The neural network, however, provided us with a reliable, continuous, and differentiable function with negligible error and was, as such, the best choice for solving the heat equation.

CONTENTS

I. INTRODUCTION

Differential equations are something we often are exposed to daily, whether it's directly or indirectly. If you are checking the weather, you are essentially looking at the solution of differential equations — namely the Navier-Stokes equations. We are slightly imprecise with our language when we say it's a solution because it is actually an approximation. The solution to the Navier-Stokes equations is actually one of the great problems in physics, and if you manage to solve these equations, you are awarded $1,000,000. Therefore, finding precise approximations to the solution of these differential equations is highly important.

The most common way to solve differential equations today is by finite difference or element methods. There are various schemes one can use, e.g. the Forward Euler, Runge-Kutta, Crank-Nicolson and many more. Nonetheless, sometimes the methods can be unstable, inaccurate, cost-inefficient, etc. Most importantly, these methods discretise the domain and provide discretised solutions. In addition, they may not be differentiable everywhere. Therefore, we will explore how to implement and use a neural network to solve a differential equation. Neural networks can obtain differentiable solutions that can be evaluated over the whole domain and are not restricted to discrete points [6][7].

II. THEORY

A. The Heat Equation

The differential equation we are going to look at is the one-dimensional heat equation. We are going to look at the following problem

$$\begin{cases} u_{xx}(x,t) = u_t(x,t) & x \in (0,1), t > 0 \\ u(0,t) = u(1,t) = 0 & t \geq 0 \\ u(x,0) = \sin(\pi x) & x \in (0,1) \end{cases} \tag{1}$$

and with these boundary and initial conditions, this can be solved analytically. This is done by separation of variables. That is, we assume that the solution can be separated into a product of functions where each function depends solely on either x or t. We thus have the following ansatz

$$u(x,t) = X(x)T(t) \tag{2}$$

and if we plug this into equation (1), we get the following

$$X''(x)T(t) = X(x)T'(t)$$

and if we dive by $X(x)T(t)$ on both sides, we get

$$\frac{X''(x)}{(x)} = \frac{T'(t)}{T(t)} = -k$$

where k is a constant. Why we have put a negative sign in front of the constant will become clear shortly. But now we have arrived at two equations

$$X''(x) + kX(x) = 0 \tag{3}$$

$$T'(t) + kT(t) = 0 \tag{4}$$

which are two ordinary differential equations (ODEs). These can be solved easily compared to most partial differential equations (PDEs). Let us start by looking at equation (3). We will first show what kind of values k can take. We do that by multiplying both sides by $X(x)$ and integrating over the domain. We start with the left-hand side

$$\int_0^1 X''(x)X(x)dx = [X'(x)X(x)]_0^1 - \int_0^1 (X'(x))^2 dx$$

$$= -\int_0^1 (X'(x))^2 dx$$

where we have used the boundary conditions. We set this equal to the right-hand side

$$-\int_0^1 (X'(x))^2 dx = -k\int_0^1 (X(x))^2 dx$$

$$\rightarrow k = \frac{\int_0^1 (X'(x))^2 dx}{\int_0^1 (X(x))^2 dx}$$

where both the numerator and denominator are obviously nonnegative. Since we are only interested in nonzero solutions, i.e. solutions which are not zero everywhere, we see that $k > 0$. Now, let $\zeta = \sqrt{k}$ and we can rewrite equation (3) as

$$X''(x) + \zeta^2 X(x) = 0$$

and this second-order linear homogeneous ODE has a known general solution for when $k > 0$. This is

$$X(x) = c_1 \cos(\zeta x) + c_2 \sin(\zeta x)$$

and plugging in the boundary conditions gives us

$$X(0) = c_1 = 0$$

$$X(1) = c_2 \sin(\zeta) = 0$$

and we have to choose a ζ that satisfies this condition. This is obtained by letting $\zeta_n = n\pi$ for $n = 1, 2, 3, \ldots$. However, if we look at the initial condition, we see that this must be $\zeta = \pi$. Thus, the solution to the ODE in equation (3), is given by

$$X(x) = \sin(\zeta x) = \sin(\pi x) \tag{5}$$

and now we need to find $T(t)$. We see that we can rewrite equation (4) as

$$\frac{dT(t)}{dt} = -kT(t) \rightarrow \frac{1}{T(t)}dT = -kdt$$

and if we integrate this, we get

$$\int \frac{1}{T(t)}dT = -k\int dt$$

$$\rightarrow \ln(T(t)) = -kt + C \rightarrow e^{\ln(T(t))} = Ce^{-kt} = Ce^{-\zeta^2 t}$$

now let $C = 1$, the solutions is then

$$T(t) = e^{-\zeta^2 t} = e^{-\pi^2 t} \tag{6}$$

thus the analytical solution of the heat equation shown in (1) is

$$u(x,t) = \sin(\pi x)e^{-\pi^2 t} \tag{7}$$

which satisfies the boundary and initial conditions [8].

We also want compare the neural network with the solution of (1) using some finite difference schemes. For the time derivative, we will implement the Forward Euler scheme

$$\begin{aligned} u_t(x,t) &\approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \\ &= \frac{u(x_i,t_{j+1}) - u(x_i,t_j)}{\Delta t} \end{aligned} \tag{8}$$

for the spatial derivative, we will implement the centered-difference scheme. This is given by

$$\begin{aligned} u_{xx}(x,t) &\approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2} \\ &= \frac{u(x_{i+1},t_j) - 2u(x_i,t_j) + u(x_{i-1},t_j)}{(\Delta x)^2} \end{aligned} \tag{9}$$

and these equations are valid for $i = 1, 2, \ldots, n$ and $j \geq 0$. This combination of schemes is often called the forward time-centered space method, or FTCS.

Now, we need to satisfy boundary conditions and initial conditions, which are given by

$$\begin{cases} u(x_0,t_j) = u(x_{n+1},t_j) = 0 & j \geq 0 \\ u(x_i,t_0) = \sin(\pi x_i) & i = 1, 2, \ldots, n \end{cases} \tag{10}$$

now, for simplicity, we use the following notation $u(x_i,t_j) = u_i^j$. We then end up with the following scheme

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{(\Delta x)^2} \tag{11}$$

and we can solve for u_i^{j+1}. We can rewrite this in vector form if we let

$$\boldsymbol{A} = \frac{1}{(\Delta x)^2}\begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \quad \text{and} \quad \boldsymbol{u}^j = \begin{pmatrix} u_1^j \\ u_2^j \\ u_3^j \\ \vdots \\ u_n^j \end{pmatrix}.$$

then equation (11) can be rewritten as

$$\boldsymbol{u}^{j+1} = (\boldsymbol{I} + \Delta t \boldsymbol{A})\boldsymbol{u}^j \tag{12}$$

[8].

Lastly, one important thing that one needs to be aware of. FTCS is an explicit method. These methods have a tendency to become numerically unstable if the time step is too large. The criterion for stability is $\Delta t/(\Delta x)^2 \leq 1/2$. Thus, for a given step size Δx, we will always choose the time step to be $\Delta t = (\Delta x)^2/2$.
Another thing we need to note is that the forward Euler and centred difference have been derived using Taylor expansions which are truncated. The time derivative is first-order accurate, while the spatial derivative is second-order accurate. More specifically, this means that the excluded terms are at most proportional to Δt and Δx^2, respectively [8].

B. Neural Network

Let us now look at how we can use a feedforward neural network to solve a differential equation. This is very different from how we have solved the equation so far.
First, let's do a summary of the different parts of a neural network. We will not, however, discuss the details thoroughly. For that see, the previous project [4], or see [1], [3] and [5].

Now, a neural network consists of several layers, and the number of layers is flexible. The necessary ones are the input and output layers. This is usually not sufficient, and the real benefits of using a neural network are when you include one or more layers in between the input and output layers, i.e. you have one or more hidden layers.

All the layers contain neurons or nodes. These take the data from all the nodes in the previous layer and produce new data to pass on. The data passed on is modified by some weights w and the receiving node adds a bias b to the data as well. If it is a hidden layer, the number of nodes is flexible and the input is modified by some activation function before it is passed on. The output layer may or may not have an activation function, depending on the output you are expecting. Lastly, the nodes in the input layer are just the input data.

Let us materialise what we have talked about so far. Let $L-1$ denote the number of hidden layers. Thus, the layer L is the output layer. Let N_l denote the number of neurons in the layer l and let $f(x)$ be an activation function. The output from the i-th node in the l-th layer can be written as

$$a_i^{(l)} = f(z_i^{(l)}) \tag{13}$$

where the input is

$$z_i^{(l)} = \sum_{j=1}^{N_{l-1}} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)} \tag{14}$$

or we can write the output from all the nodes in vector form as

$$\boldsymbol{a}^{(l)} = \boldsymbol{f}(\boldsymbol{a}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}) \tag{15}$$

where the output from the layer L, $\boldsymbol{a}^{(L)}$, is then the prediction.

If we let the weights and biases be denoted as $\boldsymbol{P} = \{\boldsymbol{W}, \boldsymbol{b}\}$, we can then assess the error by a cost function $C(\boldsymbol{P})$. We can then make use of the backpropagation algorithm to calculate the gradient of the cost with respect to the different weights and biases. The gradient in combination with some gradient descent method, can be used to find the optimal weights and biases to produce a better prediction. The most common gradient descent method to use, and the one we are going to use, is stochastic gradient descent.
After all this, we do a new feedforward pass and get a new prediction. Everything is repeated until we reach an adequate error [1][5].

1. *Differential Equations*

We will now look at how neural networks solve differential equations.
The basis for neural networks being able to solve these types of problems and much more is the universal approximation theorem. This theorem roughly says that a neural network can approximate any continuous function, to any precision, if it has at least one hidden layer [1][5]. Thus, we are able to approximate $u(x,t)$ in equation (1).

Now, in the previous project [4], we explored how we could use a neural network to fit a function to some data, as well as classification problems. The only thing that changed between those two scenarios was the use of an activation function in the output layer and the cost

function.
We are going to treat the output layer the same way as we did when trying to fit a function to some data. That is to have one output node at the output layer with no activation. This is rather obvious. Given an input, we want one output. If the output layer had an activation function, we would get the activation function as a result.

The second thing we need to change is the cost function. In order to define our cost, we need to look at the following first. An arbitrary PDE of some function g with N inputs can be written as

$$f(x_1, \dots, x_N, \frac{\partial g}{\partial x_1}, \dots, \frac{\partial g}{\partial x_N}, \frac{\partial^2 g}{\partial x_1 x_2}, \dots, \frac{\partial^n g}{\partial x_N^n}) = 0$$

and we can rewrite equation (1) in this way as well. This is then

$$f(x, t, u_{xx}, u_t) = u_{xx} - u_t = 0 \tag{16}$$

which is our PDE.

When solving PDEs, one needs to satisfy some conditions. In our case, we have both initial and boundary conditions. Thus, we have to limit the neural network such that it satisfies this. This is done by using a trial solution. This is written as

$$g_{trial}(x, t) = h_1(x, t) + h_2(x, t, N(x, t, P)) \tag{17}$$

where each function has a specific purpose. The function $h_1(x, t)$ is a function that ensures that $g_{trial}(x, t)$ satisfies the initial and boundary conditions. $N(x, t, P)$ is the output of the neural network, and $h_2(x, t)$ ensures that this output is zero when the initial or boundary conditions are supposed to be met. Our case is pretty trivial. The only thing we need to ensure is that the function is $u(x, 0) = \sin(\pi x)$ and $u(0, t) = u(1, t) = 0$. We see that we could just let $h_1(x, t) = \sin(\pi x)$, we just need to make sure that the second term is zero at $x = 0$, $x = 1$ and $t = 0$. This can be something like $h_2(x, t) = x(1 - x)t \cdot N(x, t, P)$. Thus, our trial solution can be written as

$$g_{trial}(x, t) = \sin(\pi x) + x(1 - x)t \cdot N(x, t, P) \tag{18}$$

and we can now maybe see the overarching goal. That is for $g_{trial}(x, t)$ to be as close to the true function $u(x, t)$ as possible [1][6].

Now, we can finally define our cost. First, let $g(x, t)_{trial} = g(x, t)$. We see that we want our trial function to approximate equation (16) as well as possible, i.e. we want $g_{xx} - g_t$ to be small or preferably zero. Our target value is therefore zero. Thus, we can use the mean square error of this as the cost function. For N data points, we have that

$$\begin{aligned} C(P) =& \frac{1}{N} \sum_{i=1}^{N} (f(x_i, t_i g_{xx}(x_i, t_i), g_t(x_i, t_i)))^2 \\ =& \frac{1}{N} \sum_{i=1}^{N} (g_{xx}(x_i, t_i) - g_t(x_i, t_i))^2 \end{aligned} \tag{19}$$

which is the cost function we want to minimise.

2. *Optimisation and Network Configuration*

Now that we know how our neural network is going to look like, we can discuss some of the details.
To even start the first feedforward pass, we need to have initialised the weight and biases, and have chosen the activation functions for the hidden layers. As we have mentioned earlier, we are not using an activation for the output layer.

We are going to try different types of activation function for the hidden layers. These are the logistic/sigmoid function, tanh, ReLU and ELU. These are given as follows in the same order as we listed them

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{20}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{21}$$

$$\text{ReLU}(x) = \max(0, x) \tag{22}$$

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases} \tag{23}$$

where we are going to use $\alpha = 1$ in the ELU function. Now, the way we are going to initialise the bias is just by setting them all to 0.1, just so that all the nodes have something they can backpropagate. The way we are going to initialise the weights is going to depend on which activation function we are going to use. They all pick random values from a normal distribution with zero mean, but the variance will differ.
When using the logistic function or tanh, we will use the so-called Glorot or Xavier initialisation. This uses a variance of

$$\text{Var}[\boldsymbol{W}^{(l)}] = \frac{2}{N_{l-1} + N_l}$$

where N_l is as we know the number of nodes in layer l.
For ReLU and ELU, we are going to use He initialisation, this uses a variance of

$$\text{Var}[\boldsymbol{W}^{(l)}] = \frac{2}{N_{l-1}}$$

and the benefits of using such initialisation instead of just picking random values from a normal distribution

with variance equal to one is that they speed up training, as well as minimise the chance of vanishing or exploding gradients during training [2].

Talking about gradients, we will use stochastic gradient descent to find the optimal weights and biases. Let's do a short rundown of this gradient descent method. We divide the data into so-called mini-batches of size $m = N/M$, where N is the number of data points, and M is the number of mini-batches. A mini-batch is picked at random, and the gradient of the cost function shown in equation (19) is calculated for the data points in the picked mini-batch, instead of the whole dataset. We then step in the direction of the negative gradient in the cost domain, towards a lower cost. This is repeated until the gradient is sufficiently small [3].

The size of the step is determined not only by the size of the gradient but also by the learning rate. This can take many forms. For example, one can have a constant learning rate for all parameters or even a unique learning rate for each parameter that changes with time. We are going to use the latter, which are adaptive optimisation methods. The one we specifically are going to use is Adam. This is because Adam is a robust and fast optimisation method that includes momentum and has lower bias compared to, e.g. RMSProp.
Adam approximates the first moment and second uncentered moment of the gradient by the use of an exponentially weighted moving average of the gradient. The first moment is used as the momentum, while the second moment is used to accumulate the gradients, which we use to divide the global learning rate by. If you recall, this is so that we can adapt the learning rate to the local landscape. For example, if the region is flat, which is of no interest to us, the second moment will be small, and the learning rate will be large. Since an exponential moving average approximates the moments, the past accumulated gradients which are of no interest, become insignificant with time [3].
The algorithm of SGD with Adam is shown in algorithm 1.

Here, $\boldsymbol{\ell}(\boldsymbol{f})$ denotes the loss function. In our case, this is the mean squared error. Also, the parameters we are going to update are the weights and biases. We are also going to use the default values for Adam during our training of the neural network.

At last, we are not going to develop our own neural network. We will utilise the machine learning library `TensorFlow` in Python. This library can be used for both building the neural network itself and for the differentiation of the trial function.

Algorithm 1: SGD with Adam

Require: Global learning rate η (default: 0.001)
Require: Exponential decay rates for moments $\rho_1, \rho_2 \in [0, 1)$ (default: $\rho_1 = 0.9, \rho_2 = 0.999$)
Require: Small constant δ for numerical stability (default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialise 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
initialise time step $t = 0$
while stopping criteria not met **do**:
  Sample a mini-batch $\{(\boldsymbol{x}_1, \boldsymbol{y}_1) \dots (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$
  Compute gradient: $\boldsymbol{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} \boldsymbol{\ell}(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), \boldsymbol{y}_i)$
  Update time step: $t = t + 1$
  First moment estimate: $\boldsymbol{s} = \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
  Second moment estimate: $\boldsymbol{r} = \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
  Bias correction of $\boldsymbol{s}$: $\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1 - \rho_1^t}$
  Bias correction of $\boldsymbol{r}$: $\hat{\boldsymbol{r}} = \frac{\boldsymbol{r}}{1 - \rho_2^t}$
  Compute update: $\Delta\boldsymbol{\theta} = -\eta \frac{\hat{\boldsymbol{s}}}{\delta + \sqrt{\hat{\boldsymbol{r}}}}$
  Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

C. Error Assessment

After calculating our approximations using FTCS or the neural network, we need to assess the error. During the training of the neural network, we've seen that we tell the network how well we have performed by calculating the mean squared error. By definition, this gives a greater weight to larger errors. This is great for training so that we don't get large deviations in our solution. For assessing the overall error after training, there is really no need to give this extra weight. Therefore, we will be using the mean absolute error, which is defined as

$$MAE(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{N} \sum_{i=0}^{N-1} |y_i - \tilde{y}_i|. \tag{24}$$

where $\mathbf{y}$ is the target values and $\tilde{\mathbf{y}}$ is the prediction [1].

III. RESULTS AND ANALYSIS

Before we look at the results of the FTCS method and the neural network, let us first see what we should expect. The analytical solution is shown in figure 1.
A physical interpretation of what is happening can be to consider a rod of some length L, in our case this is just 1. The units are not specified, but the principle still applies. Now, imagine that the ends are cooled to a temperature of $T = 0$ at all times. The initial temperature of the rod will follow a sine wave as shown in figure 1. As time passes, the cold will propagate inwards, and eventually, the rod will have a uniform temperature of 0.

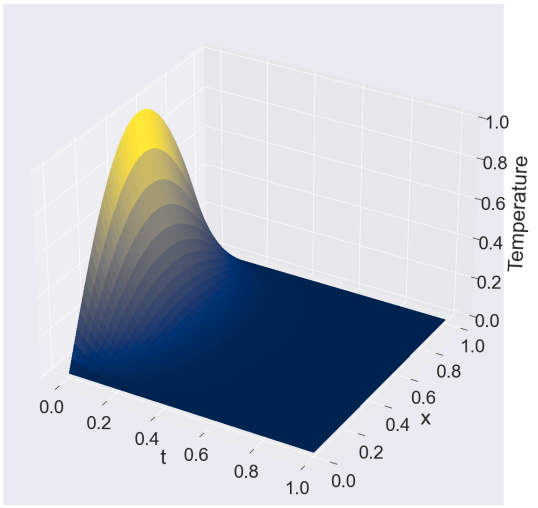

Figure 1: A plot of equation (7) for $x, t \in [0, 1]$

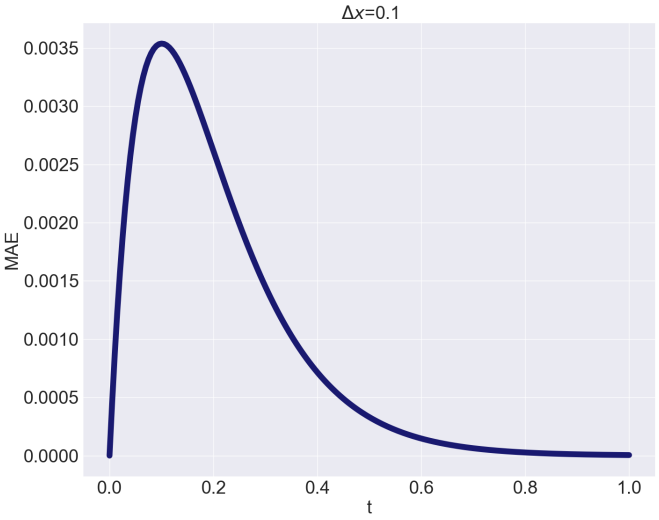

Figure 2: The mean absolute error of the FTCS method with $\Delta x = 1/10$ compared with the analytical solution in equation (7).

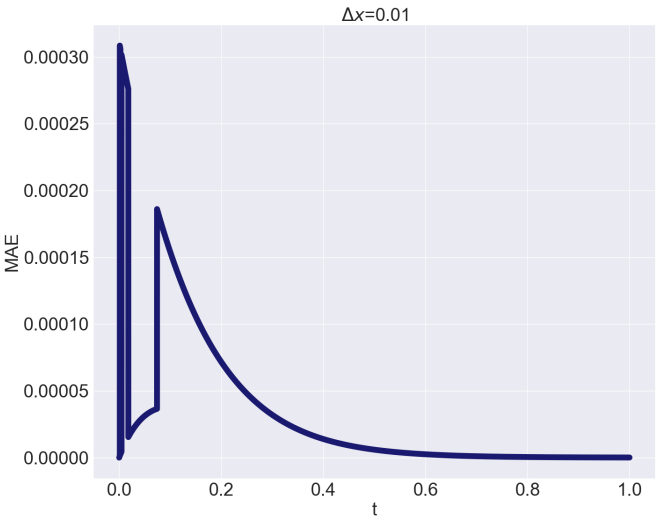

Figure 3: The mean absolute error of the FTCS method with $\Delta x = 1/100$ compared with the analytical solution in equation (7).

A. Finite Difference Method

Let us now look at how well the FTCS method performs. We will compare two cases, one with $\Delta x = 1/10$ and one with $\Delta x = 1/100$. Recall that the time step is set to be $\Delta t = (\Delta x)^2/2$. We will look at the solution up to $t = 1$. The amount of steps we need to take to reach this is given by $n = t/\Delta t$. We see that for the first case, when taking large steps in space, we are taking in total of 200 steps in time. For the second case, however, we are taking 20,000 steps in time! The amount is larger with two orders of magnitude. Let us see if this is worth the cost.

The mean absolute error of $\Delta x = 1/10$ is shown in figure 2, while the MAE of $\Delta x = 1/100$ is shown in figure 3. First of all, note the scales of the y-axis on the graphs. The largest error when using $\Delta x = 1/100$ is about 11 times smaller than the largest error when we have $\Delta x = 1/10$. So we can with confidence say that a smaller step size gives us a much better error. In addition, the largest error just happens in short spikes, while the error with the large step is a smoother, Gaussian-shaped error with a longer tail.

This shouldn't really come as a surprise. We saw in section II A. that the FTCS method is derived by truncating the Taylor series and that the remainder is proportional to the step sizes. We can see that we reduced the step sizes by a factor of 10, and the error decreased accordingly! Therefore reducing the step sizes would reduce the errors we make when dropping the remaining terms in the series.
Nonetheless, this comes at a computational cost as we have said. Even though the smaller step size is much more accurate, the large step size still has a very small error. There is therefore a trade-off. If computational resources are limited and a slightly less favourable performance is acceptable, one should opt for the larger step size. If not, one should consider a smaller step size.

B. Neural Network

Let us now look at how well a neural network does. We are going to train and test the network on a 25×25 grid for $x, t \in [0, 1]$. We thus have 625 points in total, where 80% will be used for training the neural network, and 20% will be used to test the neural network to see how well it has done. This is because the neural network can overfit the training data, i.e. it learns the training

data well, and if we were to assess the error on the data it was trained on, it would give us a good result. However, this model would fail to generalise and would perform poorly on unseen data.

We first need to determine the architecture of the model, therefore we are going to do a grid search on the number of hidden layers and neurons per hidden layer. The model will initially be trained using the logistic function, as shown in equation (20). We have decided to train the neural network for 5000 epochs, with a mini-batch equal to the size of the training set. So we are essentially not using batches. This is because we found the computational cost to be very high due to the differentiation of the trial function for each batch. Even though we find the gradient for a much larger dataset, the speed-up is substantial.
Now, let us look at the result, which is shown in figure 4. We can see that the best results are in the middle. If the complexity is too low, the network can't learn the underlying pattern and will underfit. On the other end, we could say that the model is overfitting, and therefore performing poorly on the test data. This is not necessarily the case. During training, it had a tendency to get stuck at a relatively high loss. This did not happen when the complexity was low. Why is that?

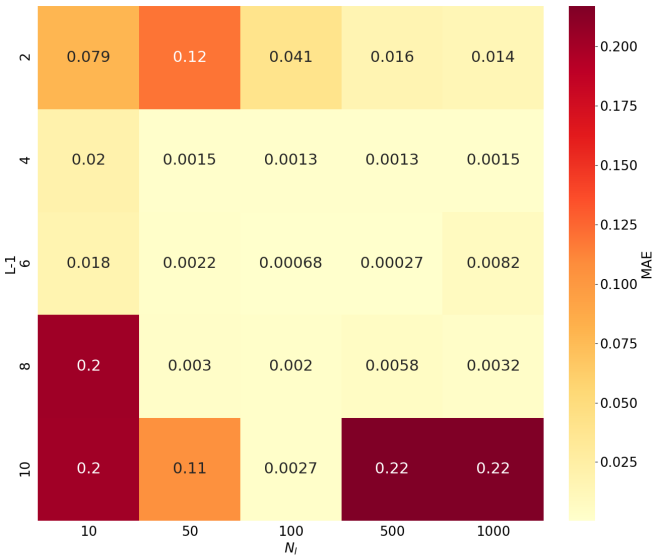

Figure 4: MAE heatmap for prediction of (1) using a neural network with test data. Shown for various number of hidden layers $L-1$ and neurons per hidden layer N_l.

If we recall, the cost function (19), is a function of the weights and biases. When we increase the number of layers and neurons, we increase the input space substantially. This in turn can make the landscape of the cost more rugged and complex, with a substantial amount of local minimums. Therefore, it seems likely that neural network got stuck in such a minima and converged prematurely.

We see that the best error is achieved with 6 hidden layers with 500 neurons each. However, if we use 6 layers with 100 neurons, the error is about 2.5 times larger, but we have reduced the number of neurons by 2400. This will speed up the computation time considerably. We will, however, stick with what has given us the lowest error.

So far, we have only used the sigmoid function as an activation function. Other activation functions may give us a better error or may speed up the computational time. In figure 5, we see the loss at each iteration during training. We see that the sigmoid/logistic function is a bit slower than the other activations. We see that it probably encounters a local minima that it struggles to get out of when the MSE is about 16, but will eventually escape. The sigmoid function is however less noisy than the others when it ultimately reaches the smaller values. Nonetheless, they all eventually stabilise.
Anyhow, we are still better off with another activation, not only because of the speed, but also because the sigmoid function is prone to vanishing gradients. This actually applies to the hyperbolic tangent as well [2].

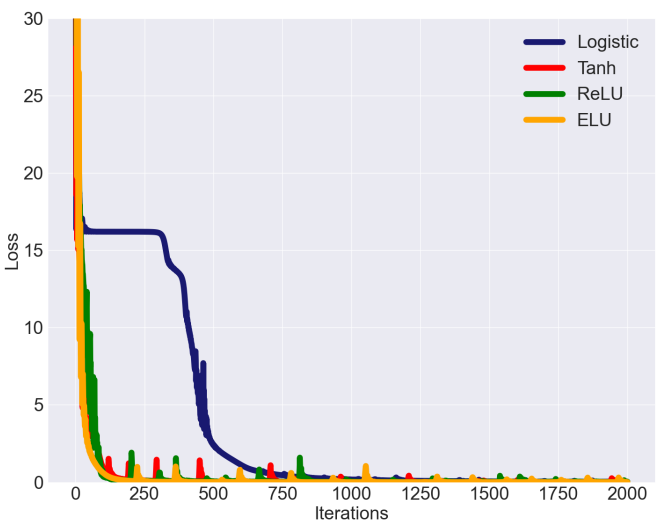

Figure 5: The loss (MSE) at each iteration during training is shown for different activation functions. The neural network used 6 layers with 500 neurons in each

In figure 6, we see these two functions. They both have in common that they flatten out at the very top and end. We therefore see that if the output of the neurons is at the very top or at the very bottom, the gradient here is essentially zero and the training will stop. As a result, it is wise to choose one of the other activation functions.

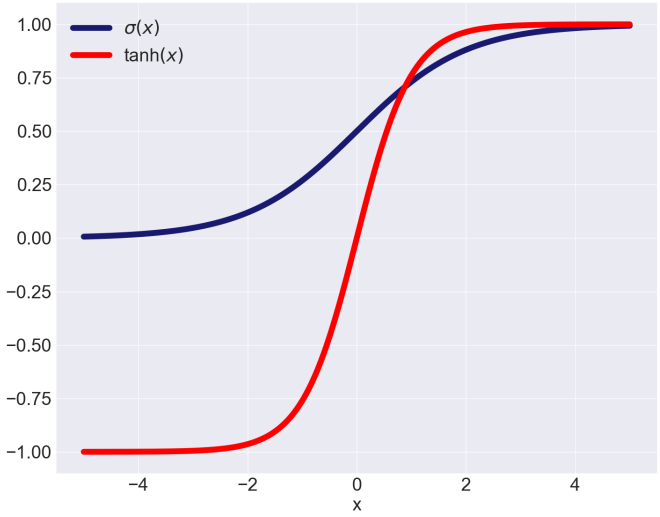

Figure 6: A plot of the sigmoid function (20) and hyperbolic tangent (21).

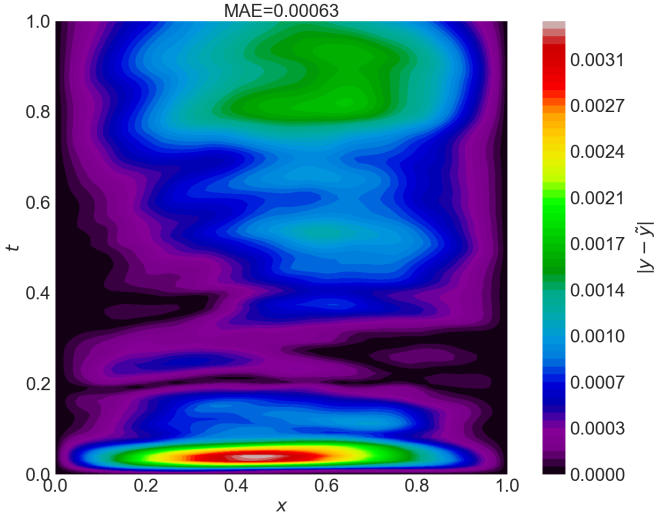

Figure 7: Heatmap of the pointwise error made by the neural network on a 100×100 grid for $x, t \in [0, 1]$.

We see that ELU looks more stable and with little large-scale noise. The neural network we are going to use moving forward will therefore have six layers with 500 neurons each, using the ELU function as the activation, and is trained for 5000 epochs. We found that training for more than this gave no further benefit.

Now, let us see how well our neural network does on a 100×100 grid for $x, t \in [0, 1]$. The result is shown in figure 7. The result looks somewhat like what we saw for the FTCS method in figure 2 and 3. The largest errors are at the very start and will gradually decrease with time. The error in the neural network will, however, start to increase slightly right before $t = 0.4$. It is no surprise that the very beginning and the peak of the sine waves have the most significant error. This is where the function has its highest complexity. We see from figure 1 that the solution is essentially set to zero after about $t = 0.4$, which is easier to interpret. Nonetheless, the error is still small and is a very good approximation. Even though the absolute error is large at the beginning, the relative error is still small. Recall that the value is close to 1, so an error of 0.003 is not very significant. Thus, we can say that the solution of the neural network is in good agreement with the analytical solution.

C. Comparison of Methods

Let us now see how the neural network compares to the FTCS method. We will look at the two cases shown in figure 2 and 3, i.e. we will look at $x, t \in [0, 1]$, where $\Delta x = 1/10$ and $\Delta x = 1/100$. Also recall that $\Delta t = (\Delta x)^2/2$. The result is shown in figure 8 and 9.
We see that when $\Delta x = 1/10$, the neural network will have an overall better performance than FTCS. However, we notice what we pointed out in figure 7 earlier, which is an increasing error starting just before $t = 0.4$. However, this error is very small, and it is still a good prediction.
When $\Delta x = 1/100$, the error using the FTCS method gives a better result, which is very close to the analytical solution as we have seen earlier. We see that the neural network will have an error that is many times larger.

Now, since we are working with such small numbers, it can be difficult to know if these errors are significant or not. For example, we see that the peak of the mean absolute error for the neural network in figure 8 or 9 is close to 0.002. As we have mentioned in the discussion of figure 7, the values of the analytical solution where this error is achieved, are close to 1. Thus a mean absolute error of 0.002 is insignificant. However, when we move to later times which are close to $t = 1$, the prediction can be many times larger or smaller, and the error will show up to be small in numerical value. We can see this in figure 10 and 11. The first shows the plot when the neural network is close to its highest mean absolute error in figure 8 and 9. The solution is close to being indistinguishable from the true solution, even though the highest mean absolute error is in this time region.

If we now look at 11, it shows the solution around the second peak at about $t = 0.8$. Here, the neural network can be as high as about ten times larger than the actual solution. In practice, this is negligible for this particular problem since it is so close to zero. However, if the prediction shown in figure 10 was ten times larger, it would not be negligible.

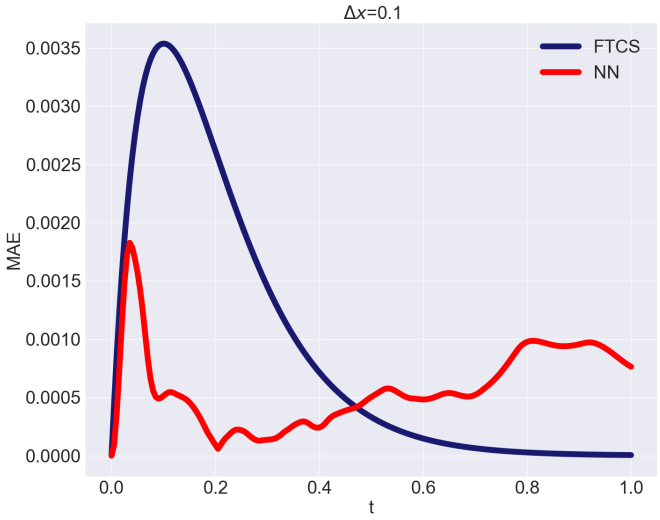

Figure 8: The mean absolute error of the FTCS method and neural network with $\Delta x = 1/10$ compared with the analytical solution in equation (7).

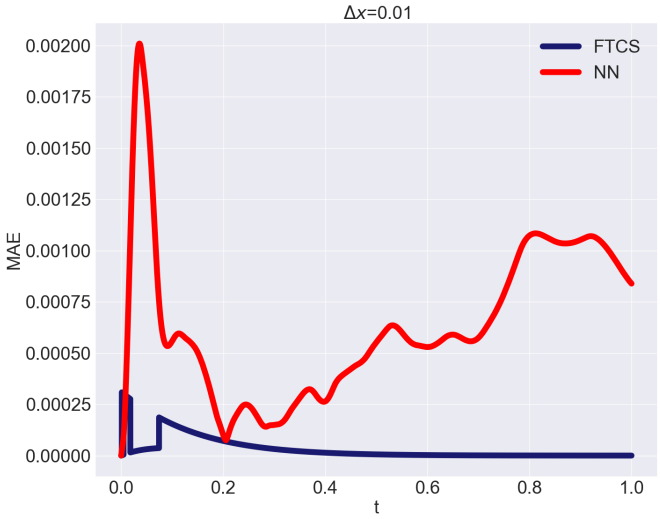

Figure 9: The mean absolute error of the FTCS method and neural network with $\Delta x = 1/100$ compared with the analytical solution in equation (7).

Since we are dealing with such small numbers, it is difficult to say what is causing this. It can be that the neural network simply fails to learn or it can be the trial function because the neural network is scaled by t. What is most likely, however, is that it is a round-off error. Our neural network uses 32-bit floating-point numbers. This has a precision to about 7 decimal points. Thus, during training, this round-off error can accumulate with the weights and biases, leading eventually to a false output.

Finally, should we choose the FTCS or the neural network for our problem? As we have seen, the neural network provides a small error. Nonetheless, the FTCS

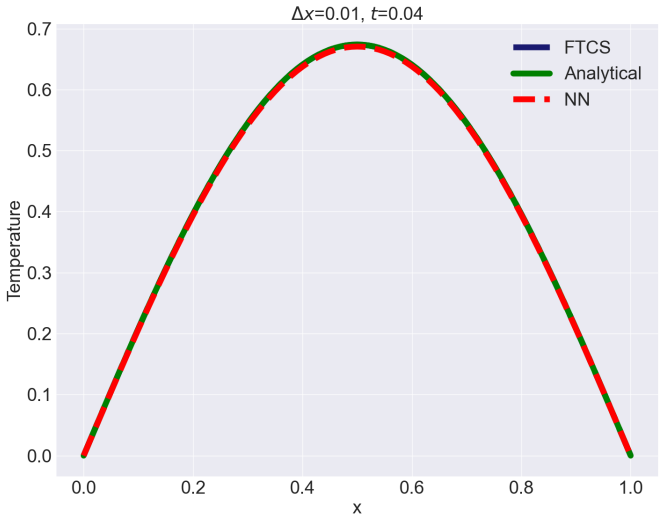

Figure 10: Show the analytical solution and approximation of the solution of (1) for $t = 0.04$ and $x \in [0, 1]$ with a step size of 0.01 in space using the FTCS method and a neural network.

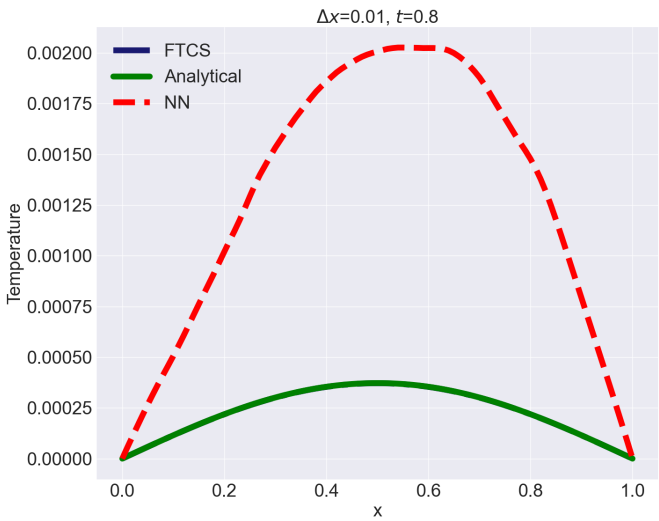

Figure 11: Show the analytical solution and approximation of the solution of (1) for $t = 0.8$ and $x \in [0, 1]$ with a step size of 0.01 in space using the FTCS method and a neural network.

method gives a better estimation when $\Delta x = 1/100$, however, this is mostly negligible, especially if we consider the benefits of using the neural network. After training the neural network, the computational cost is essentially non-existent. If we are using the FTCS with $\Delta x = 1/100$, we have seen that it takes 20.000 steps to compute the value at $t = 1$, which is significant. For the neural network, you simply plug in $t = 1$ into the function and the network just needs to do a single feedforward pass with already trained weights and biases to get the output.

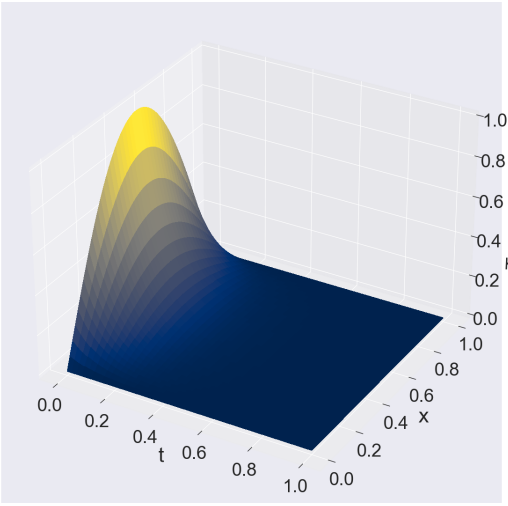

Figure 12: A plot of the approximated solution of (1) using a neural network.

Secondly, the input domain to FTCS solution is limited by the step sizes, i.e., it cannot compute the solution within a grid point. Let us say we are using $\Delta x = 0.1$, then you cannot get the solution for x=0.15, assuming we start at $x = 0$. However, the function we end up with after training the neural network is continuous and can be evaluated at any point. In addition, it is a differentiable equation. So, all in all, the neural network is the better choice. The approximation of the solution using the neural network for $x, t \in [0, 1]$ is shown in figure 12. By visually assessing this, it looks exactly like figure 1.

IV. CONCLUSION AND SUMMARY

We have looked at ways to solve the one-dimensional heat equation, with given initial and boundary conditions as shown in (1). We have used a finite difference method, which approximated the time derivative using the forward Euler method, and the second derivative in space was approximated with the centered-difference method. Together we called this the FTCS method, short for forward time-centered space. This was solved using a step size in space equal to $\Delta x = 1/10$ and $\Delta x = 1/100$. The step size in time was always $\Delta t = (\Delta x)^2/2$. We found that using $\Delta x = 1/100$ gave a much better approximation, but for evaluating the solution up to $t = 1$ required $20,000$ steps, while the larger step size only needed 200 steps.

We also trained a neural network with 6 hidden layers, with 500 neurons in each and with the ELU function shown in equation (23) as the activation. This resulted in a continuous and differentiable function which approximated the solution of the heat equation, with the given conditions very well. It performed generally better than the FTCS method when we used $\Delta x = 1/10$. However, the best error was achieved when $\Delta x = 1/100$ was used with the FTCS method. However, the error was generally very small, such that the better approximation was negligible. The neural network had a slightly higher error for larger t than both cases, which we presume to be due to round-off errors. All in all, the neural network seems to be the better choice because of the continuous and differentiable nature of this method.

Generally, solving differential equations using neural networks seems promising. If one is dealing with a problem which operates at the smallest scales, one should consider scaling up the problem such that one avoids round-off errors. We have also considered a fairly simple problem. If one is dealing with a very complex differential equation, such as the Navier-Stokes equations, the neural network would probably need a lot of training and tuning, which is time-consuming. The finite-difference methods are, however, easy to implement. Thus, it seems like neural networks are better when we are dealing with simple problems, but for larger problems, finite difference methods may be the quickest and simplest way.

[1] M. Hjorth-Jensen. (2021). *Applied Data Analysis and Machine Learning.* Available at: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html (Accessed: December 17, 2023).

[2] A. Géron. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (2nd edt.). O'Reilly Media.

[3] I. Goodfellow, Y. Bengio, A. Courville. (2016). *Deep Learning.* MIT Press. Available at: http://www.deeplearningbook.org (Accessed: December 17, 2023).

[4] A. Atifi. (2023). *Analysis of Classification and Regression Problems Using a Feedforward Neural Network.* Available at: https://github.com/achat97/FYS-STK4155/tree/main/Project2/Project (Accessed: December 17, 2023).

[5] M. A. Nielsen. (2015). *Neural Networks and Deep Learning.* Determination Press. Available at: http://neuralnetworksanddeeplearning.com/ (Accessed: December 17, 2023).

[6] I.E. Lagaris, A.C Likas, D.I Fotiadis. (1997). *Artificial neural networks for solving ordinary and partial differential equations.* IEEE transactions on neural networks, 9 5, 987-1000. https://doi.org/10.1109/72.712178

[7] M.M. Chiaramonte, M. Kiener. (2013). *Solving differential equations using neural networks.* https://cs229.stanford.edu/proj2013/ChiaramonteKiener-SolvingDifferentialEquationsUsingNeuralNetworks.pdf

[8] A. Tveito, R. Winther. (2005). *Introduction to Partial Differential Equations.* Springer Berlin, Heidelberg.