Solving the heat-equation using neural networks

Ivar Stangeby

December 20, 2019

Abstract

In this project we compare a standard finite difference method to neural networks in the solution of the heat equation. We show that the neural network severely underperforms when compared to the finite differences, and that the run-time is significantly slower. This gives us no immediate reason to chose a network based approach over classical methods. All relevant code can be found on GitHub:

github.com/qTipTip

1 Introduction

Partial differential equations have widespread use in the sciences, in fields ranging from structural engineering to economics. The solution of such equations in an accurate and precise way is of paramount importance.

Several approaches to the solution of such equations exists, with the more notable being *finite difference* and *finite element*-methods. These two methods approach the problem at hand from two different angles. In the finite difference schemes, the differential operator is discretized, typically using schemes arising from Taylor approximations to function. Finite element methods on the other hand, instead of discretizing the operators, discretize the function space for which the solution is sought.

In recent years, the use of artificial intelligence in the solution of partial differential equations has started to gain some traction, see for instance the DeepXDE-network [1] and the Deep Galerkin Method (DGM) [2].

The neural network approach to solving PDEs is meshless, that is, it does not rely on any underlying discretization of the domain as opposed to finite difference and finite element methods. Furthermore, the boundary conditions of the problem are encoded directly in the cost-function used to optimize the neural network. In finite elements the solution space is constructed in such a way as to automatically satisfy the boundary conditions, and in finite differences the boundary conditions must be imposed explicitly.

2 Theory

2.1 The heat equation

We wish to solve the one-dimensional heat-equation. The problem reads as follows. Find u such that

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \tag{1}$$

subject to the boundary conditions $u(0,t) = u(1,t) = 0$ and the initial condition

$$u(x,0) = \sin(\pi x). \tag{2}$$

We interpret the problem as that of finding the temperature gradient in a rod of fixed length $L = 1$.

An analytical solution

In order to assess a-posteriori model performance, we need to know the exact solution to our specific instance of the heat equation.

Following [3] we derive an analytical solution. We assume separation of variables and see whether this leads us to anything conclusive. We write

$$u(x,t) = X(x)T(t) \tag{3}$$

where X carries the spatial dependence and T carries the temporal dependence of the problem. By taking the required partial derivatives we obtain $X''(x)T(t) = X(x)T'(t)$ from which we can deduce that the ratios

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)} = -\lambda \tag{4}$$

are constant. From the boundary conditions, we have that

$$u(0,t) = X(0)T(t) = 0, \tag{5}$$

$$u(1,t) = X(1)T(t) = 0. \tag{6}$$

By examination, we note that if $T(t) = 0$ we obtain the trivial solution $u = 0$, which does not satisfy the boundary conditions. Hence $T \neq 0$. Thus to satisfy the boundary conditions we must have $X(0) = X(1) = 0$.

The ratios in Equation (4) gives rise to two ODEs, one in time and one in space:

$$X''(x) + \lambda X(x) = 0, \tag{7}$$

$$T'(t) + \lambda T(t) = 0. \tag{8}$$

We have obtained boundary conditions for the spatial ODE, hence we start by solving that.

As the constant λ is unknown, we have to consider three cases. The cases $\lambda < 0$ and $\lambda = 0$ can be shown to lead to $u = 0$ which we discard. Thus we assume $\lambda > 0$. The solution to Equation (7) is then

$$X(x) = A\cos(\sqrt{\lambda}x) + B\sin(\sqrt{\lambda}x). \tag{9}$$

By imposing the boundary conditions we obtain $A = 0$ and $B\sin(\sqrt{\lambda}) = 0$. Again, if we allow $B = 0$, we obtain $u = 0$, so we discard this possibility. Hence $\sin(\sqrt{\lambda}) = 0$ from which we can deduce that

$$\lambda = \pi^2 n^2 \tag{10}$$

for $n \in \mathbb{N}^+$. This tells us that we have an infinite number of solutions, one for each n:

$$X_n(x) = B_n \sin(\pi n x) \tag{11}$$

for some unknown constant B_n.

We now solve for the spatial component T. For each possible choice of λ we obtain

$$T_n'(t) + \pi^2 n^2 T_n(t) = 0, \tag{12}$$

which has solutions

$$T_n(t) = C_n e^{-n^2\pi^2 t}. \tag{13}$$

Combining Equations (11) and (13) we get the family of solutions

$$u_n(x,t) = B_n C_n \sin(n\pi x) e^{-n^2\pi^2 t}. \tag{14}$$

In principle, we would have to consider the fourier coefficients of the infinite sum of these solutions, however, by examining our initial conditions, we see that u_1 is the desired solution, with $B_1 C_1 = 1$.

Thus, our closed form solution to the one-dimensional heat equation with our prescribed boundary and initial conditions is:

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t}. \tag{15}$$

2.2 Finite differences

We wish to solve the heat equation over the domain $\Omega = [0, L] \times [0, T]$. Discretizing using N grid points in the spatial direction and M grid points in the temporal direction, we obtain step sizes $\Delta x = L/(N-1)$ and $\Delta t = T/(M-1)$. Of the myriad of different finite difference schemes, we choose the *forward in time—centered in space* (FTCS).

Forward in time

We start by discretizing the time-differential in Equation (1) by taking a first-order Taylor approximation of u. We have

$$u(x, t+\Delta t) = \sum_{n=0}^{\infty} \frac{\Delta t^n}{n!} \frac{\partial^n u(x,t)}{\partial t^n} = u(x,t) + \Delta t \frac{\partial u(x,t)}{\partial t} + \mathcal{O}(\Delta t). \tag{16}$$

We discard higher order terms and rearrange, yielding

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x, t+\Delta t) - u(x,t)}{\Delta t}, \tag{17}$$

with a truncation error that goes as $\mathcal{O}(\Delta t)$.

Centered in space

The centered difference scheme involves two second-order Taylor-aproximations in space:

$$u(x+\Delta x, t) = u(x,t) + \Delta x \frac{\partial u(x,t)}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 u(x,t)}{\partial x^2} + \mathcal{O}(\Delta x^2) \tag{18}$$

$$u(x-\Delta x, t) = u(x,t) - \Delta x \frac{\partial u(x,t)}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 u(x,t)}{\partial x^2} + \mathcal{O}(\Delta x^2). \tag{19}$$

Truncating, adding the equations to elimiate the first order derivatives, and solving for the second derivative yields:

$$\frac{\partial^2 u(x,t)}{\partial x^2} \approx \frac{u(x+\Delta x, t) - 2u(x,t) + u(x-\Delta x, t)}{\Delta x^2} \tag{20}$$

with a truncation order of $\mathcal{O}(\Delta x^2)$.

The full scheme

Plugging Equations (17) and (20) into the heat equation given in Equation (1) yields

$$\frac{u(x, t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x+\Delta x, t) - 2u(x,t) + u(x-\Delta x, t)}{\Delta x^2}. \tag{21}$$

We are interesting in solving the equation forwards in time, so we solve for the term $u(x, t+\Delta t)$, yielding:

$$u(x, t+\Delta t) = \big(u(x+\Delta x, t) + u(x-\Delta x, t)\big) \frac{\Delta t}{\Delta x^2} + \left(1 - 2\frac{\Delta t}{\Delta x^2}\right) u(x,t). \tag{22}$$

Note that the solution at the next time step requires three spatial solutions at the previous time-step. Hence, we are reliant on the boundary conditions to keep the algorithm running.

While this scheme is easy to implement, it suffers from stability issues, which can be combated by tweaking the ratio $\Delta t/\Delta x^2$. By ensuring that

$$\Delta t \leqslant \frac{\Delta x^2}{2}, \tag{23}$$

the method behaves nicely.

2.3 Neural network in the context of PDEs

Consider a neural network $f : \Omega \to \mathbb{R}$ parameterized by θ. The question is now, how do we embed the partial differential equation in the neural network? The answer lies in how we engineer our loss function.

Criterion

From Equation (1) we can see that we wish to minimize the following expression:

$$\min_{\theta} \left[\frac{\partial^2 f(x,t;\theta)}{\partial x^2} - \frac{\partial f(x,t;\theta)}{\partial t} \right] \tag{24}$$

for $x \in [0, L]$ and $t > 0$. Thus, we can simply use the mean squared error of the residuals in the right hand side of Equation (24). We therefore define our cost function $\mathcal{C}$ as the mean squared residual error over all (assume n) data-points in $\Omega = [0, L] \times [0, T]$.

$$\mathcal{C}(f) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\partial^2 f(x_i, t_i)}{\partial x^2} - \frac{\partial f(x_i, t_i)}{\partial t} \right)^2 \tag{25}$$

Embedding boundary conditions

Currently, we are not imposing any boundary conditions. Thus, by naively running the network as is, we may find any function whose second partial derivative in space is equal to its partial derivative in time. There are several approaches to solving this problem. In [1] they train the network according to a multi-task loss following the same trick as in Equation (24) but for data-points lying on the boundary.

We instead opt for the method of constructing a *trial function* that acts as a mediator between the neural network and the loss-function which ensures that the network learns to satisfy the boundary conditions.

Recall from our boundary and initial conditions that we require $u(0,t) = u(1,t) = 0$ and $u(x,0) = \sin(\pi x)$. The function g defined by

$$g(x,t;\theta) = (1-t)\sin(\pi x) + x(1-x)tf(x,t;\theta), \tag{26}$$

satisfies these conditions. It is therefore this function we pass into our cost function $\mathcal{C}$.

Neural network architecture

As our architecture we wanted to reuse the GENERICNN implemented in [4], which is a feed-forward neural network with RELU-activations at each hidden layer. However, after initial testing, using RELU yielding wildly oscilating loss, hence we chose the more stable SIGMOID-activation. This led to slower learning rates, due to the saturation of gradients, but allowed the network to get a tighter approximation.

2.4 Performance metrics

In order to evaluate our numerical approximations, whether they are computed using finite-difference or they are trained neural networks, we have several metrics we can consider. Calling our numerical approximation to u for $\tilde{u}$, we define our errors as

$$e_p = \|u - \tilde{u}\|_p, \tag{27}$$

where $\|\cdot\|_p$ denotes the matrix p-norm. We consider in particular the ∞-norm, coinciding with the maximum absolute error, and the 2-norm, which is the standard euclidean (Frobenius) norm.

3 Implementation

The finite difference solver is implemented using a naive for-loop approach in `numpy` and can be seen in Listing 2. In order to satisfy the stability criterion, we choose our desired Δx and compute a corresponding Δt by

$$\Delta t = \Delta x^2/2. \tag{28}$$

The neural network is implemented in `tensorflow`, which takes care of the backpropagation of gradients through the network using the `AutoGrad`-package. Ideally, we wanted to implement this in `PyTorch`, however, taking element-wise gradients of the network output with the respect to the network input, as is required to evaluate Equation (26), turned out to be hard to do in `PyTorch`. Due to my lacking experience in `Tensorflow`, the implementation is a modified version of the one used to solve the wave equation in [5]. This was implemented in `Tensorflow V1`, thus we had to add the compatibility modifications shown in Listing 1 in order for this to run using the newer `Tensorflow V2` API.

4 Numerical experiments

We start by running the forward in time–central in space scheme. The errors are reported in Table 1. The 2-norm steadily decreases, which means that on average the approximation becomes

```python
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()
```

Listing 1: Compatibility modification needed to use the deprecated functionality of `Tensorflow` V1 in the V2 API.

```python
def ftcs(space_resolution, time_resolution, space_min_max=[0, 1],
                time_max=1, boundary_conditions=[0, 0], initial_condition=None):
    """
    Solves the 1D-heat equation using a forward-in-time
    centered-in-space discretization scheme.

    :param space_resolution: number of points in spatial direction
    :param time_resolution: number of points in temporal direction
    :param space_min_max: the spatial boundary values
    :param time_max: the time to run the simulation for
    :param boundary_conditions: the boundary conditions at the space_min_max-values
    :param initial_conditions: the initial conditions at time = 0, callable.
    """

    if initial_condition is None:
        initial_condition: lambda x: 0

    dt = time_max / time_resolution
    u0, dx = np.linspace(*space_min_max, num=space_resolution, retstep=True)
    u = np.zeros((time_resolution, space_resolution))
    u[:, [0, -1]] = boundary_conditions
    u[0] = initial_condition(u0)

    F = dt / dx ** 2
    for step in tqdm.trange(time_resolution - 1):
        for i in range(1, space_resolution - 1):
            u[step + 1, i] = u[step, i] +\
                    F * (u[step, i - 1] - 2 * u[step, i] + u[step, i + 1])

    return u

@np.vectorize
def initial_condition(x):
    return np.sin(np.pi * x)
```

Listing 2: A `numpy`-based implementation of the *forward in time—centered in space* scheme. The initial conditions are passed in as a callable, and the function returns the solution at all time-steps in form of a `time_resolution` × `space_resolution` matrix.

better with decreased grid-size. Also, the ∞-norm decreases steadily towards zero. With this in mind, the approximations are fairly good at even low spatial resolutions. Due to the stability criterion, running this method for even smaller grid-sizes in space quickly becomes infeasible.

Table 1: Solving the heat equation for varying spatial resolution, with the temporal resolution chosen to satisfy the stability criterion. We report both the e_1 and e_2 errors.

FTCS	e_∞	e_2
$\Delta x = 1/10$	0.00357	0.04674
$\Delta x = 1/20$	0.00098	0.03688
$\Delta x = 1/30$	0.00044	0.03113
$\Delta x = 1/40$	0.00025	0.02743
$\Delta x = 1/100$	0.00004	0.01787

We now run the neural network. During testing we realized that the neural network also needs a balance between the number of data-points in the temporal domain and the number of data-points in the spatial domain. We therefore run the network with both spatial and temporal resolution increasing at the same rate, following that of Table 1. We evaluate the same errors. The network is run for 10000 epochs (which in hindsight might be on the fewer end of things), and the results are shown in Table 2. Note in particular how the error not neccesarily decreases with higher resolution. This might indicate that a different architecture is to be prefered. Maybe a shallower network with larger width. This will also reduce the run-time of the network, as the current architecture is quite sluggish.

Taking the reported errors at face-value, the classical finite difference method is to prefer, despite its stability issues.

Table 2: Solving the heat equation for identical spatial and teporal resolution. In addition to the errors not neccesarily decreasing on increased resolution, the network complexity quickly increases, which incurs significant run-time increase. This might indicate that a different architecture is to be prefered.

FTCS	e_∞	e_2
$\Delta t = \Delta x = 1/10$	0.03684	0.14905
$\Delta t = \Delta x = 1/20$	0.14728	1.24897
$\Delta t = \Delta x = 1/30$	0.03979	0.39177
$\Delta t = \Delta x = 1/40$	0.04963	0.55036
$\Delta t = \Delta x = 1/100$	0.05103	1.35165

5 Conclusion

In this project we have discussed the solution of the heat-equation, both using finite-difference methods, and a neural network based approach. Based on the results, the *forward in time—centered* *in space* difference scheme outperformed the neural net both in accuracy and in runtime, despite its stability issues.

The benefits of using a neural network for this problem is hard to see at face value. One benefit is however that the neural network does not need an underlying domain discretization. This does not constitute any advantage in such a simple case as our 1D diffusion equation, but in higher dimensions, this can prove to be a great advantage.

For instance in the finite element method, the numerical properties of the approximation greatly hinges on the quality of the underlying mesh.

The poor results of the neural network sparked an interest in trying out different architectures, however I spent too much time trying to make this work using `PyTorch`, and realized too late that I had to migrate to `Tensorflow`, where my experience is lacking.

In conclusion, for simple tasks like this, it seems like the neural network works well, comparatively, for very coarse domain discretizations, however falls behind finer resolutions.

Outside of the scope of this project would be the solution of the 2D wave equation using a neural network over a non-trivial domain, which might be able to highlight some of the benefits of not requiring domain discretization.

A Finding eigenvalues

In this appendix we briefly discuss the idea behind how you can estimate eigenvalues and eigevectors of a real symmetric matrix using the neural network framework discussed in this project. Unfortunately, I was unable to make the network converge, and due to time-constraints this was relegated to an appendix.

Following [6] we can also use our neural network to find the eigenvalues of a real symmetric matrix A. In particular, let Q be a 6×6 random matrix and define

$$A = \frac{Q^T + Q}{2}, \tag{29}$$

which then is a symmetric real matrix.

In [6] they show that any non-zero equilibrium point of the system

$$\frac{\partial u(t)}{\partial t} = -u(t) + f(u(t)), \tag{30}$$

where $f(u)$ is defined as

$$f(u) = \left[u^T u A + \left(1 - u^T A u\right) I\right] u \tag{31}$$

is an eigenvalue of the matrix A. Here u is the network output.

To recast this into our framework, we wish to minimize the function

$$\frac{\partial u(t)}{\partial t} + u(t) - f(u(t)) \tag{32}$$

with respect to our network parameters. The network is shown to converge to *an* eigenvalue of the matrix A. However, you can make the network converge to the largest eigenvalue by choosing an initial condition for the network non-orthogonal to the eigenspace of the largest eigenvalue. This eigenspace is not known a-priori, so a random initialization is made with the motivation that the probability is rather large that the random vector is not orthogonal to the eigenspace.

References

[1] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, "DeepXDE: A deep learning library for solving differential equations," Jul. 10, 2019. arXiv: `1907.04502 [physics, stat]`. [Online]. Available: `http://arxiv.org/abs/1907.04502` (visited on 12/18/2019).

[2] A. Al-Aradi, A. Correia, D. Naiff, G. Jardim, F. G. Vargas, and Y. Saporito, "Solving Nonlinear and High-Dimensional Partial Differential Equations via Deep Learning," p. 76,

[3] M. J. Hancock, "Linear Partial Differential Equations," p. 44,

[4] I. Stangeby, *FYS-STK4155 - Project 2*, Dec. 17, 2019. [Online]. Available: `https://github.com/qTipTip/FYS-STK4155` (visited on 12/18/2019).

[5] K. Hein. (). Data Analysis and Machine Learning: Using Neural networks to solve ODEs and PDEs, [Online]. Available: `https://compphysics.github.io/MachineLearning/doc/pub/odenn/html/._odenn-bs039.html` (visited on 12/20/2019).

[6] Z. Yi, Y. Fu, and H. J. Tang, "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix," *Computers & Mathematics with Applications*, vol. 47, no. 8, pp. 1155–1164, Apr. 1, 2004, ISSN: 0898-1221. DOI: `10.1016/S0898-1221(04)90110-1`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0898122104901101` (visited on 12/20/2019).