Exploring advantages/disadvantages of solving simple PDEs with Neural Networks

Tor-Andreas S.Bjone

December 18, 2022

Abstract

We looked at the mean squared error of several neural networks and an explicit solver against a closed-form solution. We found that the neural network that performed best was using the ReLu activation function, the RMSprop otimizer and had 3 hidden layers. But this had a MSE of 2 orders of magnitude higher than the explicit solver after 20 epochs. We then timed the explicit solver against a already trained network for different matrix sizes and found that the explicit solver was about two orders of magnitude faster than the neural network. For such a simplistic problem a traditional solver performs better than a neural network.

Contents

1 Introduction

In recent years the applications of Neural Networks have been expanded massively. There has been done a lot of research into solving PDEs with Neural Networks, such as the paper (Zobeiry & Humfeld, 2020)[**2**]. Solving/approximating such systems as the Schrödinger equation or the Navier-Stokes equation with no closed-form solutions can be very beneficial.

In this report we will tackle a simple problem, the heat equation in 1D, to see how a Neural Network performs against a conventional numerical method, the Forward Time Central Space (FTCS) scheme. To evaluate the methods we choose a boundary condition with a closed-form solution and look at the difference between this and our numerical solutions. And to set up the Neural Network we use Keras, which is a TensorFlow API.

The method section first goes trough through the closed-form solution and then the Neural Network. And the results start by comparison of the mean squared error (MSE) before looking at the time it takes to solve using the explicit method versus the Neural Network.

2 Theory and Methods

2.1 Heat equation

We will look at the heat equation in one dimension.

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, t \geq 0, x \in [0, L] \tag{1}$$

or

$$u_{xx} = u_t, \tag{2}$$

with initial conditions

$$u(x,0) = \sin(\pi x) \quad 0 \leq x \leq L, \tag{3}$$

and with $L = 1$ being the length of the x-region of interest. The boundary conditions are

$$u(0,t) = 0 \quad t \geq 0, \tag{4}$$

$$u(L,t) = 0 \quad t \geq 0. \tag{5}$$

2.1.1 Closed-form solution

The analytical solution to this problem is derived in (Tveito and Winther, 2005, pp. 90-92)[**1**], so we will just present a small summary of the method here. First we assume that $u(x,t)$ is linear and homogeneous and that the equation is separable in the form of $u_k(x,t) = X_k(x)T_k(t)$, where k refers to it being a particular solution. We can then solve the equation by separation of variables to find

$$u(x,t)_k = e^{-(k\pi/L)^2 t} \sin\left(\frac{k\pi}{L}x\right), \ k = 1, 2, 3, ..$$

This will then give the family u_k of particular solutions. We assume then that $f(x)$ can be written as a linear combination of the eigenfunctions of $X(x)$, so that

$$f(x) = \sum_{k=1}^{N} c_k \sin\left(\frac{k\pi}{L}x\right),$$

where c_k is some constant. Then it follows by linearity that

$$u(x,t) = \sum_{k=1}^{N} c_k e^{-(k\pi/L)^2 t} \sin\left(\frac{k\pi}{L}x\right).$$

Now inserting that $f(x) = \sin(\pi x)$ and $L = 1$ it is easy to see that this gives $c_1 = 1$ and all other constants $c_k = 0$. This gives us the **closed-form solution** of

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

2.1.2 Numerical approximation

We will solve this by so called FTCS-scheme (Forward Time Central Space) and use the Forward-Euler method for moving in time. This results in

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}.$$

And to simplify notation, let

$$u(x_i,t_j+\Delta t) \to u_{i,j+1},$$

and

$$u(x_i+\Delta x,t_j) \to u_{i+1,j}.$$

Then we re-write the equation as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{\Delta x^2}.$$

Now we can define $\alpha = \Delta t/\Delta x^2$ to get

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}, \tag{6}$$

Where we have discretized x and t so that

$$x_i = i\Delta x,\ i = 0,1,2,...,n,$$

and

$$t_j = j\Delta t,\ j = 0,1,2,...,m.$$

And this scheme is only numerically stable when the **Von Neumann stability criterion** is met:

$$\alpha \leq 1/2,$$

as described in (Tveito and Winther, 2005, pp. 132-133)[**1**]. And it has a **truncation error** of $O(\Delta x^2)$ (Tveito and Winther, 2005, p. 64)[**1**].

We note that this problem can be reduced to solving a matrix system, where

$$A = \begin{bmatrix} 1-2\alpha & \alpha & 0 & 0 & ... & 0 \\ \alpha & 1-2\alpha & \alpha & 0... & 0 & \\ 0 & \alpha & 1-2\alpha & \alpha & 0 & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & 0 & \alpha & 1-2\alpha \end{bmatrix}$$

is a tri-diagonal Topelitz matrix. And we can define a column vector for the time step as

$$V_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ ... \\ u_{n,j} \end{bmatrix}$$

so that

$$V_{j+1} = AV_j. \tag{7}$$

And this can be Forward solved using the Toeplitz forward solver algorithm 1.

Algorithm 1 Toeplitz Forward solver algorithm

Require: Spacial step-size Δx
Require: Timespan T and lengthspan L.
Require: Stability criterion parameter α.
Require: Function f(x)=u(x,0).
Calculate diagonal elements: $a = \alpha$, $b = 1 - 2\alpha$.
Calculate first timestep $u(x, 0) = f(x)$ and set boundary conditions.
while Time is less than T **do**
$u(x, t + \Delta t) = a \cdot u(x - \Delta x, t) + b \cdot u(x, t) + a \cdot u(x + \Delta x, t)$.
Set boundary conditions.
end while

This is a method of solving a tri-diagonal matrix system without having to do the matrix multiplication.

For our program we have chosen to hold the entire solution in space and time in memory. But it is also possible to iterate in time without holding in memory and just save the time-steps you want to look at. But for the 1D case this is not a concern on a modern computer as we will look at arrays with a maximum of $10^4 \times 10^4$, which will be under one Megabyte with 32-bit numbers.

2.2 Solving with Neural Networks

We will use the **keras API for TensorFlow** to approximate the heat equation with a NN. To train the network we will use the analytical solution with the mean squared error as the cost function. This is done by creating a matrix X that holds all x and t values and a matrix u with the corresponding analytical values. We will let x be of length $n = 100$ and t be of length $m = 100$ for the training of the network. Then for the testing we will double the size of n and m. We do not need to shuffle the dataset or do a train/test split when testing in this way, although some of the training set will be included in the test set when doing it this way. This will not matter as we increase the size of the dataset for the testing and will therefore be able to spot overfitting if it should happen.

We will set up the network with 3 hidden layers, with 16 neurons in the first layer, then 32 neurons in the following two layers. This will be tested using the Adam and RMSprop optimizers, and ReLu and Sigmoid activation functions. Then we will take the best performing of these and increase the complexity of the network by adding new layers of 32 neurons. These will all be trained using a batch size of 30 and 20 epochs.

Then we will look at the time the trained network uses for giving out a prediction, or feed forwarding, for a two layer network versus the explicit solver. Where this network has been trained using RMSprop and has ReLu activation functions.

3 Results and Discussion

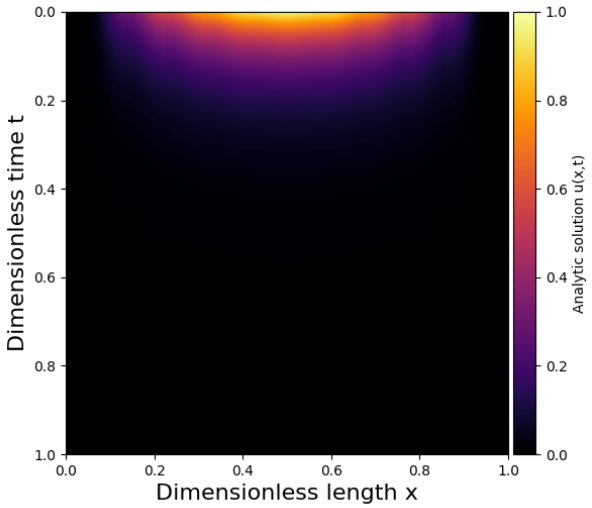


Figure 1: Analytic solution for the heat equation with initial conditions $u(x, 0) = \sin(\pi x)$ and boundary conditions $u(0, t) = u(1, t) = 0$.

All networks are initialized with weights drawn from a Xavier uniform initializer and bias=0.

In figure 1 we see the analytic solution for our given boundary and initial conditions. This will be used to test our numerical solvers using the absolute error between this and the numerical approximation. This can be seen in figure 6. In the subfigures (a)-(d) we see the NNs prediction for different activation functions and optimizers. We can see that ReLu with RMSprop has the lowest error. And we note that all of the approximations have the highest loss at the boundaries. We can especially see that Adam with ReLu has a very high error at the $t = 0$ boundary. The reason our network is not handling the boundaries well is because the loss function has no special penalizing of wrong boundaries. This could be fixed by creating a new loss function by looking at the minimization of the NNs prediction

$$\min_{\hat{u}} \left\{ \|\hat{u}_{xx} - \hat{u}_t\|^2 + \|\hat{u}(0,t)\|^2 + \|\hat{u}(1,t)\|^2 + \|\hat{u}(x,0) - \sin(\pi x)\|^2 \right\},$$

where this norm $\|\cdot\|$ is to be read as the grand sum of the matrix. We then define our cost/loss-function as

$$C(\hat{u}) = \frac{1}{2} \left[\|\hat{u}_{xx} - \hat{u}_t\|^2 + \|\hat{u}(0,t)\|^2 + \|\hat{u}(1,t)\|^2 + \|\hat{u}(x,0) - \sin(\pi x)\|^2 \right].$$

We can then add parameters in front of the norms to decide how costly it would be for the network to not prioritize the boundaries. This is called a physics informed neural network and has been explored in a previous paper (Zobeiry & Humfeld, 2020)[**2**]. Getting back to figure 6, we look at the explicit solver in (e). Note here that the limits on the colorbar is changed because the error is so small compared to that of the NNs. We see that now our boundaries is forced to be correct. And we can see that as we move in time the middle of the rod gets a larger and larger error until it starts cooling down and of course the error gets smaller because the values get closer and closer to zero because of the $u(0,t) = u(1,t) = 0$ boundaries. We look more at the explicit solver in 2. We can see a big drop in MSE as half α from the criterion 1/2 to 1/4. By doing this we decrease the MSE by about 4 orders of magnitude. While decreasing Δx by 10 we only get a decrease in MSE of about one order of magnitude. This means that decreasing α will have the biggest effect on the accuracy of the solution. Now looking at 3 we can see loss versus epochs for different activation functions and optimizers. We see that the MSE is higher for all of these compared to the explicit solver. And we see that Adam with ReLu looks constant. This could be because the network died or it reaches a local minima in the gradient decent.

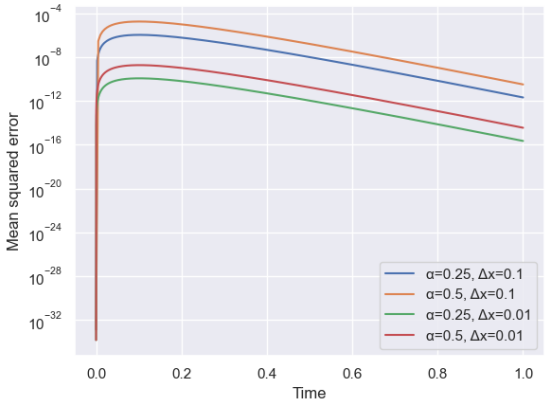


Figure 2: MSE of explicit solver for different $\alpha = \Delta t/\Delta x^2$ and Δx values.

We further explore RMSprop with ReLu since this has the lowest MSE. In figure 4 we see the MSE as a function of epochs for different number of hidden layers. We see that for 5 hidden layers we have a constant loss. This could be because the network is too complex for our problem and dies. This may be solved with different initializations of weights and biases. But it seems like for 3 and 4 hidden layers we do not get a big difference in MSE so 3 hidden layers may be enough for such a simple problem.

We chose our 3 layer network with RMSprop activation function and Adam optimizer and train this. Then in figure 5 we see the time it takes to predict depending on the matrix size against

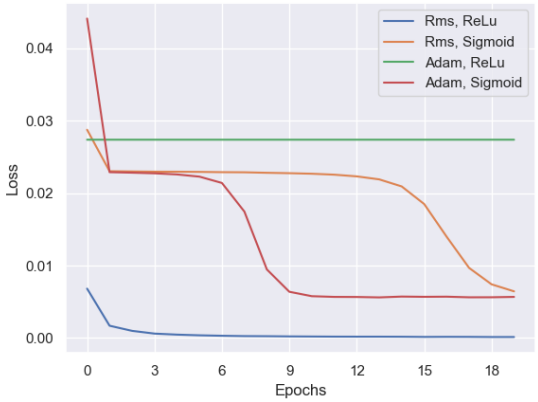


Figure 3: MSE loss function for different optimizers and activation functions. All are run with a batch size of 30 and trained on a $(n = 100) \times (m = 100)$ matrix against the analytical solution. The networks have 3 hidden layers, one of 16 neurons and two of 32 neurons.

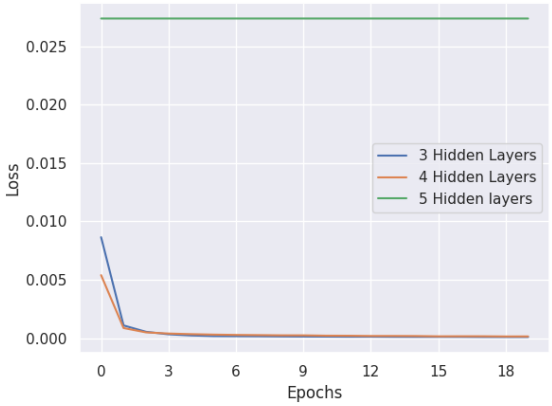


Figure 4: MSE loss function for different number of hidden layers. First hidden layer is 16 neurons and the rest are 32. All are run with a batch size of 30 and trained on a $(n = 100) \times (m = 100)$ matrix against the analytical solution. Activation functions and ReLu and the optimizer is RMSprop.

the time used by the explicit solver. We can see that in logspace both graphs are linear, meaning they are taking exponentially more time as we increase the matrix size. This is expected for the explicit solver. For the Neural Network we disregard the low matrix size points because this is only run one time and is therefore very dependent on small variations by the processes already running on the computer. These effects will be smaller and smaller the bigger the matrix size as such effects will average out. So then we can also say that the predictions of the neural networks grow linearly in logspace. And it is about two orders of magnitude bigger than the explicit solver.

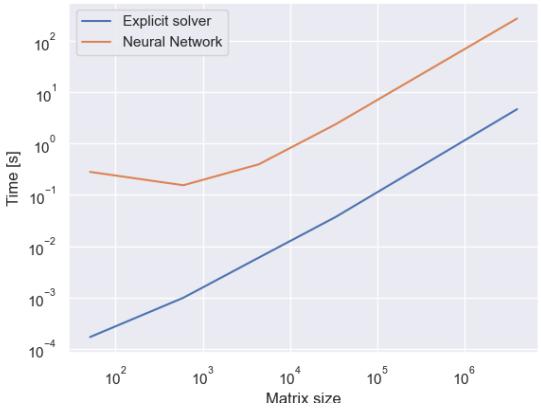


Figure 5: Time vs matrix size for a neural network with three hidden layers, one of 16 neurons and two of 32 neurons. The network is already trained using a batch size of 30 on a $(n = 100) \times (m = 100)$ matrix against the analytical solution. Activation functions and ReLu and the optimizer is RMSprop. And the explicit solver has $\alpha = 1/4$ and uses different Δx to calculate the matrix size that the NN gets to solve. Run on a 1,8 GHz dual-core Intel Core i5.

4 Conclusion

For the Neural Network we found that the simpler networks performed better on this problem. Specifically a 3 layer network trained using RMSprop, with ReLu activation functions performed best in terms om the MSE against the analytical solution. But we found that the explicit solver performed better, with about two-three orders of magnitude lower MSE. This is mostly because of the boundary conditions, as there is no implementation in the networks to penalize high MSE on the boundaries. For future work we could create a cost function for the network as described in the previous section, by looking at the minimization problem of the heat equation and the boundary conditions.

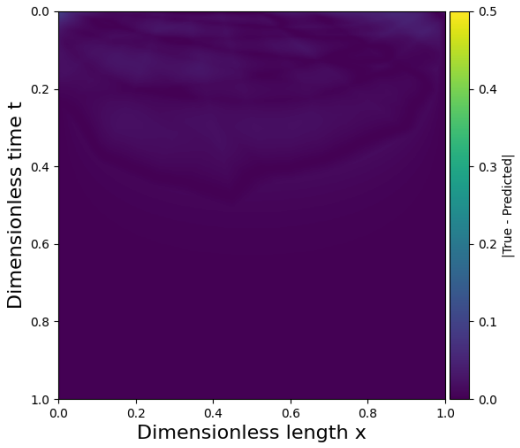
We also found that the time the already trained network used to predict a solution was about two orders of magnitude slower than the time used for the explicit solver. This indicates that for such a simple problem there is no benefits to using a Neural Network. For future work one could also look at implicit schemes such as Crank-Nicolson, which is stable for all Δx and Δt and could possibly solve even faster.

One could imagine that as the complexity of the problem increased the Neural Network would perform better compared to the explicit scheme. Therefore it would be beneficial to look at the heat equation in 2D and 3D. And if we have a cost function that is not based on an analytical solution, we could train the network with ICs as parameters and see if the network can predict for ICs that it has not been trained on. In that case one could solve problems without closed-form solutions.

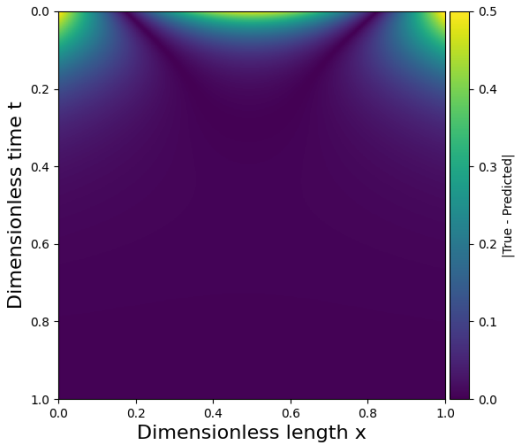
References

[1] Tveito, A. & Winther, R. (2005) Introduction to Partial Differential Equations: A computational approach. 2nd edn. New York, NY, USA: Springer.

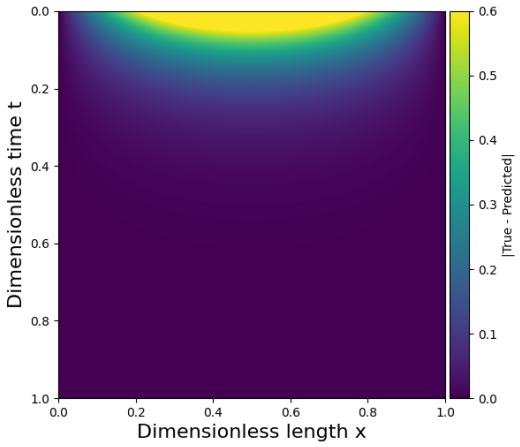
[2] Humfeld, K & Zobeiry, N. (2020) Physics-Informed Machine Learning Approach for Solving Heat Transfer Equation in Advanced Manufacturing and Engineering Applications. Washington, Seattle, WA: University of Washington



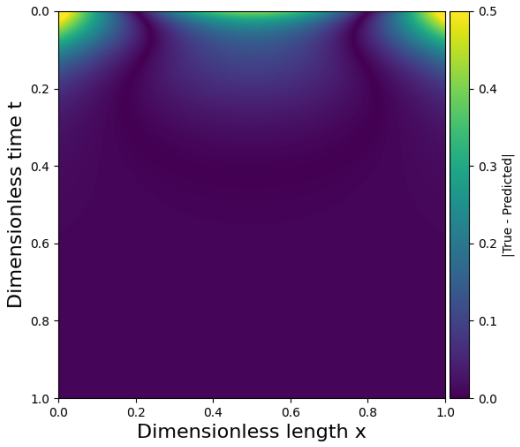
(a) Rms, ReLu



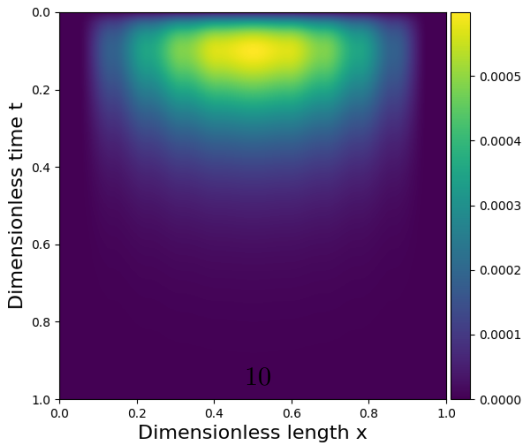
(b) Rms, Sigmoid



(c) Adam, ReLu



(d) Adam, Sigmoid



(e) Explicit solver

Figure 6: Absolute error for NN in (a)-(d) and explicit solver in (e). NN is trained with 20 epochs and a batch size of 30 with $(n = 100) \times (m = 100)$ data points of x and t respectively, using the analytical solution for MSE as the loss function. The networks have three hidden layers, one of 16 neurons and two of 32. The explicit solver in (e) is using $\Delta x = 0.1$ and $\alpha = 1/4$.