FYS-STK4155 - Project 3

Kari Lovise Lodsby

December 17, 2021

Abstract

In this project, I look at how neural networks perform for solving partial differential equations. First I looked at the heat equation, which is a well-known PDE, and compared the accuracy and efficiency of a neural network method to the standard forward Euler scheme for finding approximate solutions.

I also discuss the possibility of solving a specific non-linear differential equation that can find eigenvectors and eigenvalues of a symmetric, real matrix. While I was not able to get the eigenvalue finder to work, I speculate a bit on its utility and how it compares to a standard linear algebra numerical diagonalization scheme.

Explicit numerical methods seem to outperform neural networks both in terms of accuracy and computational performance. They also have the advantage of being easier to implement. However, there are some aspects of neural networks that I did not examine (scouting for optimal parameters, saving the network parameters after training) that could make them perform better.

1 Introduction

Partial differential equations (PDEs) describe problems in multiple sciences, such as mathematics, statistics and physics. Many of them are difficult or impossible to solve analytically, so we have to use some kind of algorithm to find an approximate solution. Numerical methods are commonly used, but they often have problems such as scaling poorly, or being unstable. In this project, I will use a numerical method and a neural network to solve PDEs to see whether neural networks perform better.

2 Theory

2.1 Analytic solution of the PDE

First I will look at the one-dimensional heat equation, the PDE I will solve. It has the form

$$\frac{\partial^2 u(x,t)}{\partial x^2} - \frac{\partial u(x,t)}{\partial t} = u_{xx} - u_t = 0$$

and I will use the boundary conditions $u(x,0) = \sin(\pi x)$, $u(0,t) = u(L,t) = 0$, where $L = 1$ is the length of the region.

To find the analytic solution,[1] I will make the ansatz that the solution is on the form

$$u(x,t) = X(x)T(t)$$

and insert it into the equation:

$$(X(x)T(t))_{xx} = (X(x)T(t))_t$$

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

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

Since each side of the equation only depends on one variable, they can only be equal if they are both constant. Therefore

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

If $k = 0$, the individual ODEs are trivial: $X(x) = Ax + b$ and $T(t) = C$.

Otherwise, we have:

$$X(x) = Ae^{\mu x} + Be^{-\mu x} (k = \mu^2 > 0)$$
$$X(x) = Ae^{\mu xi} + Be^{-\mu xi} (k = -\mu^2 < 0)$$
$$T(t) = Ce^{\mu^2 x}$$

Using the boundary conditions:

The boundaries $x = 0$ and $x = L = 1$ vanish. The initial spatial condition is on the form $u(x, 0) = \sin(\pi x)$. If k vanishes or $k = -\mu^2 > 0$, both A and B will also vanish. However, if $k = -\mu^2 < 0$, we find

$$X(0) = A\cos(\mu x) + B\sin(\mu x) = 0$$

so $A = 0$, and

$$X(1) = B\sin(\mu x) = 0$$

so $B = 0$ too.

Now we have an infinite set of equations that determine the Fourier coefficients of the initial condition:

$$u(x, 0) = \sum_{n=1}^{\infty} B_n \sin(n\pi x)$$

$$B_n = 2\int_0^1 dx u(x, 0)\sin(n\pi x)$$

Using the value of $\mu = n\pi$

$$T(t) = e^{-n^2\pi^2 t}$$

Applying the initial condition $u(x, 0) = \sin(\pi x)$: Combine the solutions into

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

Since $\{\sin(n\pi x)\}$ is an orthogonal basis, we know that

$$B_n = 2\int_0^1 \sin(\pi x)\sin(n\pi x) = \delta_{1n}$$

So $B_1 = 1$ and the rest of the B_ns are equal to 0, and the solution is

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

2.2 Numeric approximation of the PDE

I will now explain the explicit forward Euler scheme.

The t derivative is given as the forward difference

$$\frac{\partial u(x, t)}{\partial t} \approx \frac{u(x_i, t_j + \Delta t) - u(x, t)}{\Delta t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t}$$

Similarly, the double x derivative is given by the central difference

$$\frac{\partial^2 u(x, t)}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2}$$

Inserting this into the PDE:

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

Note that the only unknown value here is the $u_{i,j+1}$. Setting $\alpha = \frac{\Delta t}{(\Delta x)^2}$ and solving for this $u_{i,j+1}$:

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

This scheme has the stability criterion $\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}$.[?] This can be a problem for large resolutions in x, as it requires a high resolution of t too.

2.3 Neural networks and PDEs

It is also possible train a neural network to solve PDEs. For this project, I will use a fully-connected, feed-forward neural network that uses back-propagation for the weights and biases. Then we must define the input layer, output layer and cost function, and discuss activation functions, learning rates, depths of the hidden layers and the sizes of each layer.

The input layer consists of two notes, corresponding to x and t. The output layer corresponds to $u(x,t)$ and has one node.

Since the heat equation is $u_{xx} - u_t = 0$, we wish to minimize $u_{xx} - u_t$ and choose the cost function

$$C(x,t,P) = (u_{xx} - u_t)^2$$

P is the parameters (weights and biases) of the neural network. Note that calculated u values will depend on P, even though the notation only mentions the two original variables. We also denote the optimal set of parameters as $\hat{P}$, which is the value that minimizes the cost function.

$$C(x,t,\hat{P}) = 0 = \min_P (u_{xx} - u_t)^2$$

We need some way to find the derivatives in the cost function, so we define the following trial function:

$$u_{trial} = u_1(x,t) + x(1-x)tN(x,t,P)$$

$N(x,t,P)$ is the output from the neural network. u_1 is some function that satisfies the boundary and initial conditions. We insert them and choose

$$u_1(x,t) = (1-t)(u(x,0) - ((1-x)x(0,t) + xu(L,t)))$$

This yields the following trial function:

$$u_{trial} = (1-t)\sin(\pi x) + x(1-x)tN(x,t,P)$$

This allows us to find the desired derivatives numerically. Then we can define the new parameters P with the following gradient method:

$$P_i = P_{i-1} - \lambda \frac{\partial C}{\partial P_{i-1}}$$

I will train the model over many iterations, and if the learning rate parameter λ is chosen well, this should converge towards a local minimum of the cost function. We hope that is also a global minimum.

Once the model has been trained using this scheme, it should be able to find the value of $u(x,t)$ if the two parameters are input.

2.4 Expanding this method to solve matrix eigenvalue problems

Look at the following differential equation:

$$\frac{d\mathbf{x}(t)}{dt} = \dot{\mathbf{x}} = -\mathbf{x}(t) + f(\mathbf{x}(t))$$

Define f like this:

$$f(\mathbf{x}) = [\mathbf{x}^T\mathbf{x}\mathbf{A} + (1 - \mathbf{x}^T\mathbf{A}\mathbf{x})\mathbf{I}]\mathbf{x}$$

Here, $\mathbf{x}$ is a real vector that represents a state in a system, and $\mathbf{A}$ is a symmetric real matrix, and $\mathbf{I}$ is the identity matrix. Its equilibrium points correspond to the eigenvalues of $\mathbf{A}$.

This suggests the cost function

$$C = [\dot{\mathbf{x}}(t) + \mathbf{x}(t) - f(\mathbf{x}(t))]^2$$

To calculate the derivative numerically, we use the following trial function:

$$\mathbf{x}_{trial}(t, P) = e^{-t}\mathbf{x}_0 + (1 - e^{-t})N(t, P)$$

The first term is there to make sure the initial conditions are satisfied. The second term follows the structure of the differential equation.

After some iterations of training, $\mathbf{x}$ should correspond to an eigenvalue of $\mathbf{A}$. As shown in the paper by Yi et al., this eigenvector belongs to the largest eigenvalue. We can also use the neural network on $-\mathbf{A}$, which makes $\mathbf{x}$ converge to the eigenvector corresponding to the minimum eigenvalue.

Once the eigenvectors have been calculated, the eigenvalues can be calculated from them:

$$\lambda = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}$$

3 Descriptions of algorithms

In this section, I will give a brief discussion of the implementation of the algorithms.

3.1 Explicit forward Euler

The relation I found can be used to calculate the discretized points $u_{i,j}$. The boundary terms are determined by the original boundary conditions, while the values for $t = 0$ are determined by the initial condition. This ensures that we can calculate every point.

I tested it with $\Delta x = 0.01$. Note that this requires $\Delta t \leq 0.00005$ respectively, which suggests that the method does not scale well because so many calculations will be made.

3.2 Neural networks

The neural network has 2 hidden layers. Each of the hidden layers has 100 nodes. The training period and learning rate are 200 epochs and $\lambda = 0.01$. I chose not to look for the ideal λ value because even a single run of the program takes a very long time because of the heavy computational load.

The activation function is the sigmoid function, as it seems like a good choice when the expected values are in $[0, 1]$.

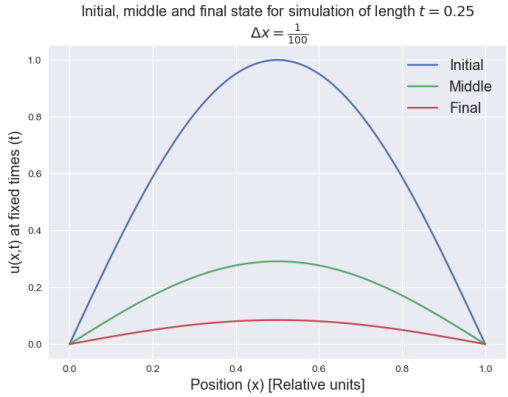
3.3 Eigenvalues

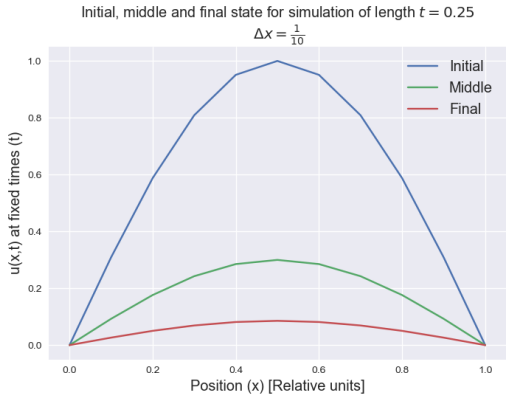
I generate a symmetric matrix $\mathbf{A}$ by generating a random square matrix and adding it to its transpose. This neural network will also have 2 hidden layers with 100 fully-connected neurons each. The input and output layers are both of size N, where N is the length of the vectors.

Both the vector and the parameters of the neural network (weights and biases) will be updated when the neural network runs. The vector should converge to an eigenvector of $\mathbf{A}$.

4 Results

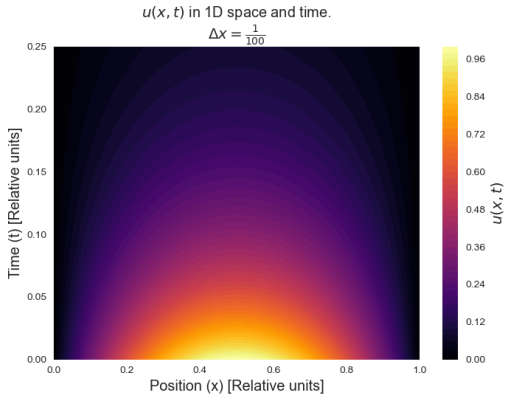
4.1 The heat equation



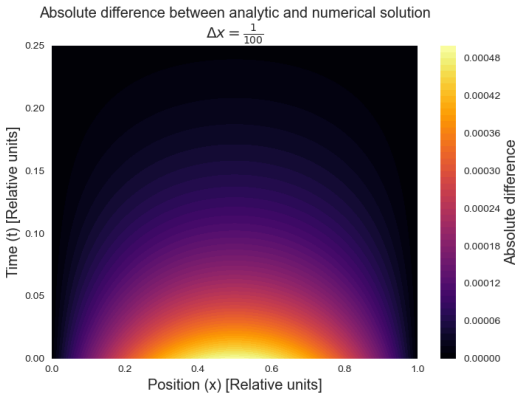


These graphs show the results of the explicit forward Euler scheme for a simulation of length 0.25 for three different t values: 0, 0.25 and approximately 0.125. Additionally, they respectively have $\Delta t = \frac{1}{20000}$ with $\Delta x = \frac{1}{100}$ and $\Delta t = \frac{1}{200}$ with $\Delta x = \frac{1}{10}$. Even the low-resolution approximation seems reasonably accurate.

Here is a colour contour plot of function values calculated by the explicit scheme with the high resolutio. The low-resolution ones are similar, and can be found in the github folder.

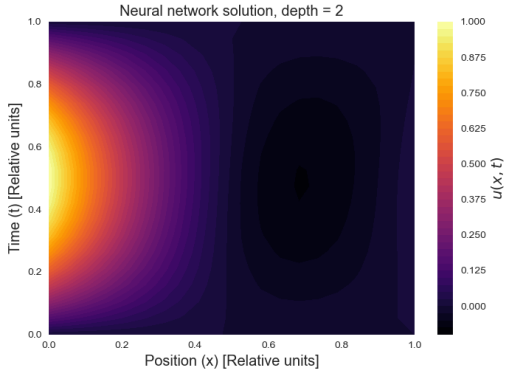


The following plot shows the difference between the analytic solution and the numerical solution:

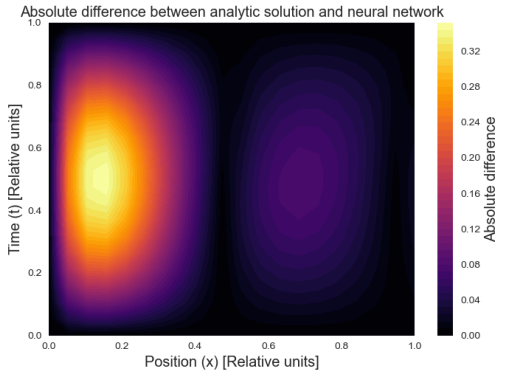


We observe that the largest absolute difference is approximately 0.0005, which is good, especially since the calculation only took about a second.

Here is the result for the neural network:



The following plot shows the difference between the analytic solution and the neural network solution:



Not only is the peak absolute value much higher at around 0.32 (which is worse than even the low-resolution forward Euler, whose heatmap suggests a peak error around 0.05), but the calculation took much longer: 32 minutes and 17 seconds. It is possible that this result would have been better with a different λ (or other aspects of network design), however. Perhaps increasing the depth or number of neurons would make it more accurate, but that would also make the learning time even longer. There is also the risk of overfitting. With that said, my high runtime does include the training time - if one wishes to solve multiple problems using the neural network, it is only necessary to train it once, after which the parameters can be saved and reused. At this point it may be fast and accurate.

4.2 Eigenvalue solver

I was not able to modify the PDE solver to work with the neural network method because adding 6 nodes made it too complex for me to handle. However, I have included my attempt at a solution.

I would have compared the NN approach to the numpy eigenvalue finder, which is fast and accurate, at least for small matrices. I would have tested how the methods scale for large matrices, sparse matrices and dense matrices. It would also have been interesting to see if the method Yi et al. would perform for matrices that are almost symmetric, and to test it with an asymmetric matrix to see what would happen.

5 Discussion

In this section, I will discuss the methods of solving the PDE, as well as my attempt to solve the eigenvalue problem.

5.1 PDE solvers

The main issue with the forward Euler scheme is scaling, thanks to the stability requirement quadratically decreasing the length of the time steps when the number of spatial steps increases. This can lead to issues, not only because of the sheer number of time steps, but because small numbers can slow down computations considerably. While I did not have problems with it for this specific problem, it is important to keep in mind if one wishes to implement the scheme. However, there are other numerical schemes that can solve PDEs without having problematic stability requirements, e.g. the Crank–Nicolson method (which is often unconditionally stable) and the backwards Euler methods (which is less accurate, but both stable and immune to oscillations). See for instance [3] for more details on such methods.

While I found many disadvantages with the neural network, it is worth mentioning that it has the advantage of being more flexible than the explicit methods. Once trained, the neural network can calculate any value of $u(x, t)$. The explicit method only calculates approximate values at certain points. For the specific problem I solved in this project, the interpolation errors would likely be small, but that is likely not the case for every problem.

If the neural network had been allowed to run for a longer amount of time, and I had more time to experiment, I would likely have ended up with a solution that is at least as accurate as the explicit scheme, and at that point the parameter values can be saved so that the network does not have to be trained again. Additionally, it is possible to optimize a neural network by changing its design, such as changing the activation function on some neurons, or changing the depth and/or width. Higher complexity would give the model more degrees of freedom, which could improve accuracy, but it would also further increase the computational cost. If the data is noisy, additional complexity might lead to overfitting instead of greater accuracy.

The learning rate λ is the most important parameter. If it is too low, the network will be even slower. If it is too high, it might make the model worse. I was unable to experiment with learning rates, which might have been a way to get a considerably better model. It is also possible to optimize other aspects of the network, such that the hidden layers. However, this would further increase a computational cost that is already extremely high.

5.2 Eigenvalue solver

This discussion is mostly hypothetical, based on what would have happened if the solver worked.

This eigenvalue solver suffers from being difficult to implement, which is a common issue for neural network-based approaches. It is also reasonable to believe that this deep learning-based solver would have many of the usual downsides associated with this kind of approach, such as being "black boxes" and requiring a lot of time to train and optimize.

Compared to standard linearization methods, the neural network method also has the obvious drawback of only being able to find the maximum and minimum eigenvalues, which renders the method unusable for problems that require all of them. However, there are still many situations where one only cares about the maximum and/or minimum eigenvalues.

Another potential problem is what happens if the initial vector happens to be orthogonal to one of the desired eigenvectors. If this happens, it is possible that the method will not converge. That is also something I could have tested.

The method discussed in this paper only works for symmetrical matrices. As the paper mentions, it is unknown if such a network-based approach might work for non-symmetric matrices.[4]

6 Conclusion

For solving the heat equation, the simple forward Euler scheme was better in every way. It was much faster and more accurate, in addition to being easier and more straight-forward to understand and implement. With that said, it is likely that the neural network would have performed better if I had been able to scout for better parameters. Additionally, once the time-consuming training of the network is finished, the parameters can be saved, which should make every subsequent use of the algorithm fast and accurate. Even then, I have to wonder if using a neural network is worth the hassle for this kind of problem, especially considering that there are other linear schemes that avoid the problems of the forward Euler scheme.

The complexity of the eigenvalue solver meant that I was not able to implement it correctly. It would have been interesting to see how it compares to the standard linear algebra methods (especially for large matrices), although I fear that it would have similar problems as the PDE solver. Additionally, this method is held back by limitations that standard eigenvalue solvers do not suffer from.

Neural networks appear to not work particularly well for the systems I looked at in this project. To solve PDEs, training and using a neural network was much slower and yielded considerably worse results than just using a simple forward Euler scheme. For the eigenvalue finder, I was not able to get the neural network to work at all, and considering my experiences with the PDE solver, I suspect that the standard methods for finding eigenvectors would be more practical for most cases.

Neural networks tend to scale well while handling complex structures. However, as this project illustrates, they are not impressive for simple structures, where simpler methods may be both more efficient and easier to implement.

References

[1] Lawrence C. Evans, *Partial Differential Equations Second Edition* 2010.

[2] M. Hjorth-Jensen, *Project 3 on Machine Learning, deadline December 17, 2021* https://compphysics.github.io/MachineLearning/doc/Projects/2021/Project3/html/Project3.html 2021.

[3] Leif Rune Hellevik, *Numerical Methods for Engineers* https://folk.ntnu.no/leifh/teaching/tkt4140/._main065.html 2020.

[4] Z. Yi, Y. Fu, and H. J. Tang, *Neural Networks Based Approach for Computing Eigenvectors and Eigenvalues of Symmetric Matrix*, Computers and Mathematic with Applications 47, (2004), 1155-1164