Project 3

FYS-STK4155 — Applied data analysis and machine learning

Anders Johansson
17th December 2018

Abstract

This report deals with the numerical solution of the one-dimensional diffusion equation. The traditional Forward Euler scheme is compared with the novel approach of using neural networks. Tensorflow is used for the neural network due to its ability to handle cost functions defined in terms of the derivatives of the neural network's output with respect to its input.

While Forward Euler performed better both in terms of accuracy and runtime, the neural networks also showed promising results, giving a mean squared deviation between the two sides of the diffusion equation on the order of 10^{-6} after 10 000 epochs. A combination of the hyperbolic tangent as activation function and the Adam optimiser performed best.

Finally, two different transformations of the neural network output into a solution fulfilling the initial and boundary conditions showed that it is advantageous to use as much information about the expected solution as possible when constructing such a transformation. When using a more naive approach, the number of nodes per layer proved more important than the number of layers, as shown by the three best architectures being $(100, 100, 100)$, $(100, 100)$ and $(10, 10, 10, 10, 10)$ nodes in the hidden layers.

Contents

1 Introduction

Differential equations are an important tool in physics for understanding the world around us. Classical motion is determined by Newton's laws of motion, which are ordinary differential equations, while relativistic motion and quantum mechanics are summarised in partial differential equations. Unfortunately, differential equations are hard to solve analytically, and numerical techniques are therefore employed. Historically, finite difference and finite element schemes have been used with great success. This report utilises a relatively new approach, namely artificial neural networks, and compares it with a simple finite difference scheme.

Artificial neural networks are used to solve the diffusion equation, which models such important phenomena as heat dissipation and diffusion. Tensorflow, Google's leading framework for machine learning and artificial intelligence, is used for a high-level, intuitive, "black box" approach. The results are compared to the simplest finite difference scheme, namely Forward Euler.

Since the main theory of neural networks was derived in an earlier project, this report mainly revolves around their application to neural networks. First, however, the report starts off with a derivation of the finite difference scheme with which the neural network is compared, before going through the details of how to use neural networks for solving differential equations. Both methods are then applied to the diffusion equation and their results are compared.

2 Theory and methods

2.1 The diffusion equation

The standard diffusion equation is

$$\frac{\partial u}{\partial t} = \nabla^2 u, \quad \text{or, in one dimension,} \quad \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \tag{2.1}$$

for $x \in [0,1]$ and $t \in [0,T]$. In this project, the initial and boundary conditions are

$$u(x,0) = \sin(\pi x), \qquad u(0,t) = u(1,t) = 0. \tag{2.2}$$

The analytical solution is obviously

$$u(x,t) = \mathrm{e}^{-\pi^2 t}\sin(\pi x). \tag{2.3}$$

2.2 Finite difference methods

Differential equations are equations where the unknown is a function, and the equation consists of terms containing the function and its derivatives. Finite difference methods simply replace the derivatives with numerical approximations based on Taylor expansions, and solve for the unknown function values.

2.2.1 The Forward Euler scheme

The simplest approximations to numerical derivatives which can be inserted in the diffusion equation are

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t}, \qquad \frac{\partial^2 u(x,t)}{\partial x^2} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2}. \tag{2.4}$$

Discretising with $x_i = i\Delta x$ for $i = 0,\ldots,N_x$ with $\Delta x = 1/N_x$, $t^j = j\Delta t$ for $j = 0,\ldots,N_t$ with $\Delta t = T/N_t$ and $u_i^j = u(x_i,t^j)$, the diffusion equation with these approximations is

$$\frac{u_i^{j+1}-u_i^j}{\Delta t} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2} \tag{2.5}$$

with boundary conditions

$$u_i^0 = \sin(i\pi\Delta x), \quad u_0^j = u_{N_x}^j = 0. \tag{2.6}$$

Since $\{u_i^0\}$ is known, the function values at the subsequent timesteps can be found recursively by

$$u_i^{j+1} = u_i^j + \frac{\Delta t}{\Delta x^2}\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right) \tag{2.7}$$

for $i = 1,\ldots,N_x-1$. The error terms are proportional to Δt and Δx^2 due to a first order approximation to the temporal derivative and a second order approximation to the spatial derivative. Stability is ensured[4] when

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}, \tag{2.8}$$

so $\Delta t = \frac{1}{2}\Delta x^2$ is used.

2.3 Neural networks for differential equations

A neural network can approximate any function, and it should therefore also be able to approximate the solution of a differential equation such as the diffusion equation. The neural network approximates by minimising a cost function, and the challenge is therefore to find a suitable cost function.

The simplest approach is simply to use the average square of the difference between the two sides of the diffusion equation as the cost function,

$$Q = \frac{1}{N_x N_t}\left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)^2. \tag{2.9}$$

It seems difficult to differentiate this cost function with respect to the weights and biases of the neural network. I have therefore chosen to use Tensorflow[2] for this project, since it contains functionality for calculating automatic derivatives (something my otherwise brilliant Fortran network lacks).

Another difficulty is to fulfill the initial and boundary conditions. One approach may be to add terms to the cost function which penalise approximations that do not fulfill the initial and boundary conditions. Another approach, which is found more often in literature[1] and therefore employed in this report, is to not use the output of the neural network directly, but rather to combine a neural

network with other functions in such a way that the initial and boundary conditions are guaranteed to be fulfilled.

In the case of the diffusion equation with our set of conditions, a natural choice is

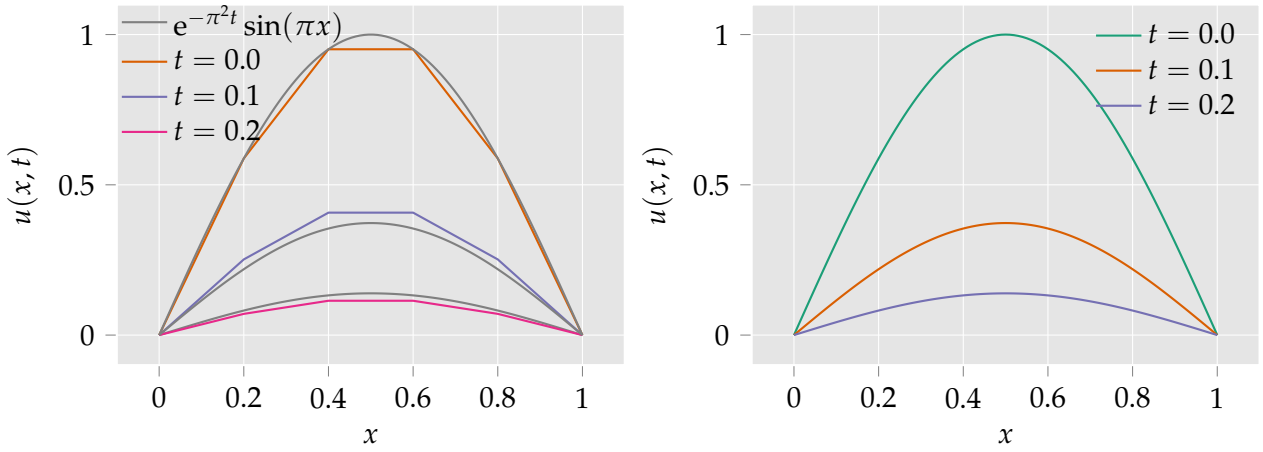
$$u(x,t) = \sin(\pi x) + tx(1-x)n(x,t), \tag{2.10}$$

where $n(x,t)$ is the output from the neural network. A possible issue with this choice is that the neural network has to turn u into a "dying sine", which requires $tx(1-x)n(x,t)$ to have a sine shape while also causing $\sin(\pi x)$ to die out. It may be better to use the knowledge of the sine shape and instead guess at

$$u(x,t) = \sin(\pi x)(1 + tn(x,t)). \tag{2.11}$$

3 Results

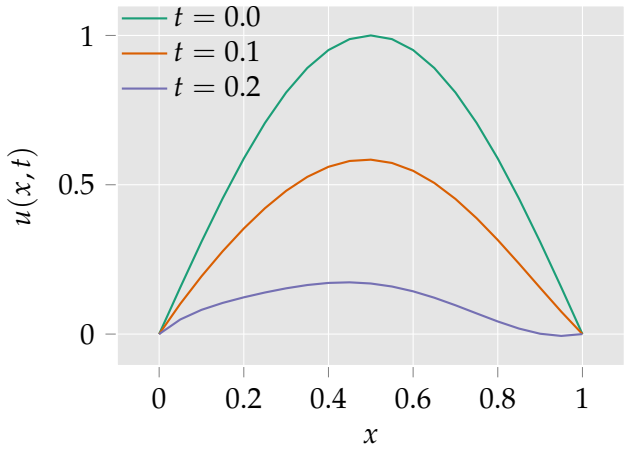
3.1 Forward Euler



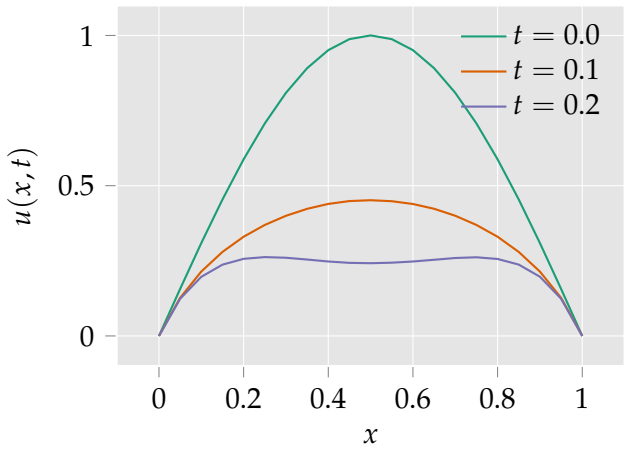
(a) $\Delta x = 0.2$ (b) $\Delta x = 0.01$

Figure 1: Results from solving the diffusion equation with the Forward Euler finite difference scheme for two different values of Δx and $\Delta t = \frac{1}{2}\Delta x^2$. While the solution for $\Delta x = 0.2$ deviates visibly from the analytic solution, $\Delta x = 0.01$ gives a good approximation.

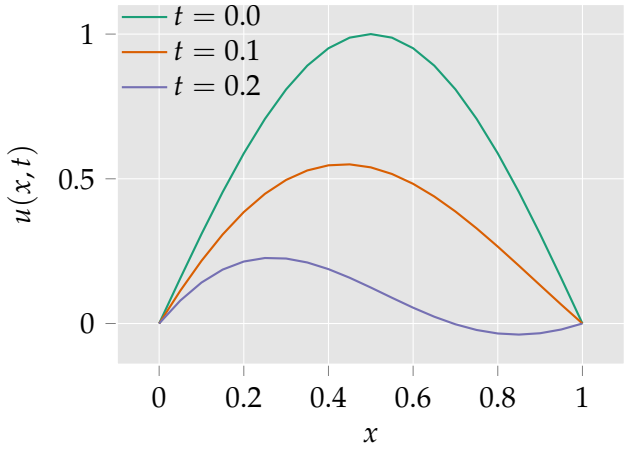
3.2 Neural networks



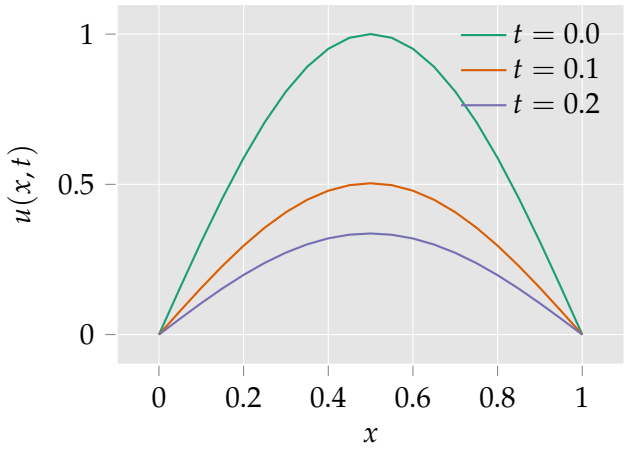
(a) $u(x,t) = \sin(\pi x) + tx(1-x)n(x,t)$, 100 epochs.



(b) $u(x,t) = \sin(\pi x) + tx(1-x)n(x,t)$, 10 000 epochs.

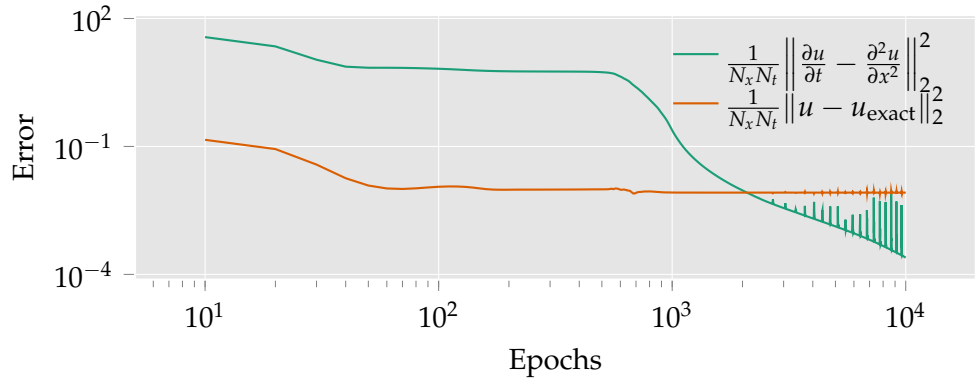


(c) $u(x,t) = \sin(\pi x)(1 - tn(x,t))$, 100 epochs.

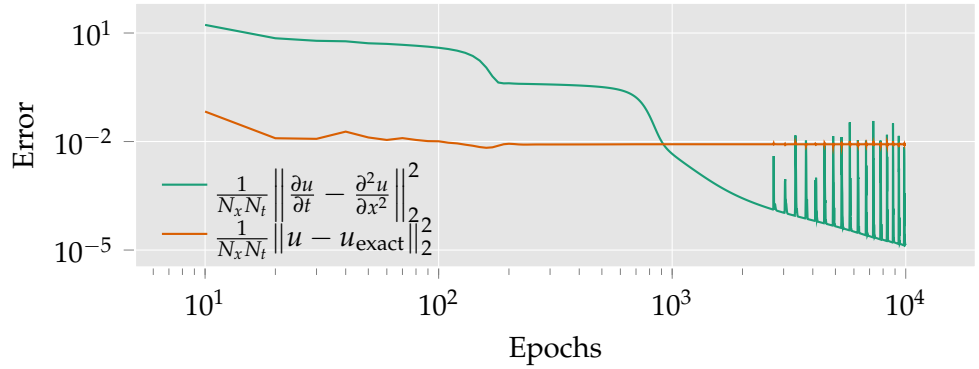


(d) $u(x,t) = \sin(\pi x)(1 - tn(x,t))$, 10 000 epochs.

Figure 2: Approximation to the solution of the diffusion equation provided by a neural network with 3 hidden layers with 100 nodes each. While the network gives horrible results after 100 epochs, the solution is seen to approach the analytic solution, especially for the second choice of approximation. Evidently, it is important to choose a clever transformation of $n(x,t)$ into a solution which fulfills the boundary and initial conditions.



(a) $u(x,t) = \sin(\pi x) + tx(1-x)n(x,t)$



(b) $u(x,t) = \sin(\pi x)(1 + tn(x,t))$

Figure 3: The error in the neural network's approximation, both in terms of the cost function defined as the squared difference between the two sides of the diffusion equation and the deviation from the known, exact solution. While the cost function is eventually reduced, it fluctuates, and the deviation from the exact solution does not follow the same trend. It is therefore clearly advantageous to do a parameter search and try different architectures and minimisers.

3.2.1 Parameter sweep

Simulations similar to the ones above were run for a variety of combinations of activation functions, optimisers and network architectures. The best results (defined as having the smallest cost) are shown below. The Adam optimiser, with adaptive learning rates and momenta, clearly outperforms batch gradient descent, while the rectified linear unit and hyperbolic tangent give better results than the sigmoid function. Furthermore, when using an "uninformed" $n(x,t) \mapsto u(x,t)$ (table 1 on the following page) it is clear that the breadth of the neural network is more important than the depth as long as the network is sufficiently deep, as shown by the networks with $(100, 100, 100)$ and $(100, 100)$ nodes in the hidden layers generally performing better than networks with $(10, 10, 10, 10, 10, 10, 10, 10)$ or (1000) nodes.

1170 parameter combinations were tested for each u. Adam denotes the `tf.train.AdamOptimizer` with default parameters, which uses an adaptive learning rate α and momentum γ, thereby outperforming batch gradient descent with fixed learning rate and momentum. The architecture gives the numbers of nodes in the hidden layers. More results are available at github.com/anjohan/ml3/data.

Table 1: $u(x,t) = \sin(\pi x) + tx(1-x)u(x,t)$.

Activation function	Optimiser	Architecture	Cost	Error
tanh	Adam	100,100,100	$2.3 \cdot 10^{-4}$	$8.3 \cdot 10^{-3}$
tanh	Adam	100,100	$1.6 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10,10	$2.8 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10,10,10	$2.9 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
relu	Adam	100,100,100	$3.0 \cdot 10^{-3}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,10,10	$3.2 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	10,10,100	$3.4 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	10,10,10	$3.7 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	100,10,10	$3.8 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	10,100,10	$4.6 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	$\alpha = 1 \cdot 10^{-1}, \gamma = 1 \cdot 10^{-1}$	100,100,100	$5.3 \cdot 10^{-3}$	$8.8 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10,10,10,10	$5.7 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10	$7.2 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
relu	Adam	100,100	$7.5 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$
tanh	$\alpha = 1 \cdot 10^{-1}, \gamma = 1 \cdot 10^{-2}$	100,100,100	$7.7 \cdot 10^{-3}$	$8.9 \cdot 10^{-3}$
tanh	$\alpha = 1 \cdot 10^{-1}, \gamma = 1 \cdot 10^{-4}$	100,100,100	$8.0 \cdot 10^{-3}$	$7.6 \cdot 10^{-3}$
relu	$\alpha = 1 \cdot 10^{-1}, \gamma = 1 \cdot 10^{-3}$	100,100,100	$8.5 \cdot 10^{-3}$	$7.7 \cdot 10^{-3}$

Table 2: $u(x,t) = \sin(\pi x)(1 + tu(x,t))$.

Activation function	Optimiser	Architecture	Cost	Error
tanh	Adam	10,10,10,10,10,10,10	$6.0 \cdot 10^{-6}$	$8.4 \cdot 10^{-3}$
tanh	Adam	100,100,100	$6.9 \cdot 10^{-6}$	$8.4 \cdot 10^{-3}$
sigmoid	Adam	100,100,100	$8.0 \cdot 10^{-6}$	$8.4 \cdot 10^{-3}$
tanh	Adam	100,10,10	$1.5 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10,10	$1.7 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
relu	Adam	1000	$2.0 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10,10,10,10	$3.4 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,10,10,10	$5.1 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
relu	Adam	100,100,100	$6.4 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
sigmoid	Adam	10,100,10	$7.0 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
relu	Adam	100,100	$7.9 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
sigmoid	Adam	100,100	$8.6 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,10,10	$9.8 \cdot 10^{-5}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,100	$1.1 \cdot 10^{-4}$	$8.4 \cdot 10^{-3}$
tanh	Adam	10,10,10	$1.1 \cdot 10^{-4}$	$8.5 \cdot 10^{-3}$
sigmoid	Adam	10,10,100	$1.1 \cdot 10^{-4}$	$8.4 \cdot 10^{-3}$
tanh	Adam	100,100	$1.2 \cdot 10^{-4}$	$8.4 \cdot 10^{-3}$

4 Summary and conclusion

The one-dimensional diffusion equation was solved with both the traditional Forward Euler scheme and the more novel approach of using neural networks. While the neural networks converged towards the analytic solution, they were clearly beaten by Euler's method, which performed better in terms of both of accuracy and runtime. The main conclusion is, as in the previous project, to use specialised methods whenever possible, while neural networks can offer an acceptable solution which does not require much knowledge from the user — especially when using a library as powerful as Tensorflow.

A parameter sweep was performed in order to find the optimal solution of activation functions, optimisers and network architectures. Results showed that networks with fewer layers with more nodes performed better than deeper networks with fewer nodes per layer. The networks performed best when using the hyperbolic tangent as activation function and the Adam optimiser, with its adaptive learning rate and momentum. Furthermore, the choice of transformation of the output from the neural network proved important for a good performance, and the choice that used the most information about the solution proved most effective, giving a mean squared deviation on the order of 10^{-6} after 10 000 epochs.

A simple extension of this project would be to try more parameter combinations, and deeper neural networks or networks with more nodes per layer. Performance issues will be mitigated if these simulations are run on a GPU. More substantial extensions include for example other network types, as the approximation of $u(x,t)$ is similar to an image problem and may therefore benefit from a convolutional neural network.

References

[1] Isaac E Lagaris, Aristidis Likas and Dimitrios I Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE transactions on neural networks* 9.5 (1998), pp. 987–1000.

[2] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[3] Pankaj Mehta et al. "A high-bias, low-variance introduction to machine learning for physicists". In: *arXiv preprint arXiv:1803.08823* (2018).

[4] Aslak Tveito and Ragnar Winther. *Introduction to partial differential equations: a computational approach*. Vol. 29. Springer Science & Business Media, 2004.