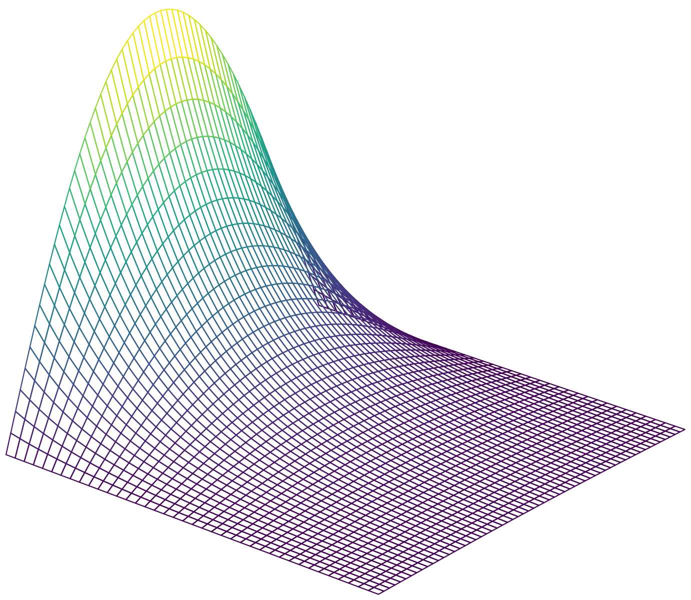
Neural networks as methods for solving differential equations.
— Project 3 —
FYS-STK4155

Heine Aabø Halvard Sutterud

December 2018



Abstract

In this project we study the solution of partial differential equations (PDE) with neural networks, and compare it to a traditional explicit scheme. The PDE in question is the diffusion equation with Dirichlet boundary-conditions. Using the hyper-parameter optimization library `hyperopt` we study different architectures, activation functions and optimizers for a network implemented in `tensorflow`. The minimal loss was obtained by a network with three hidden layers with a total of 302 nodes and tanh as activation, optimized using Adam optimization [4] with a learning rate of 10^{-3}. This gave a mean squared error of the loss function as 0.95×10^{-5} and a MSE of 1.5×10^{-8}, compared to the forward euler method which gave a MSE of 5.2×10^{-4} for a similar number of lattice points. However this is at the cost of performance, with 10000 epochs used during training for the network to converge to the given results.

Contents

1 Introduction

Partial differential equations are of huge interest in a variety of fields in science and has proved particularly useful as models for physical problems. The equations are often complex, demanding large amounts of calculations and efficient algorithms to solve. With neural networks being proved successful in many applications, an interesting subject would be their performance in solving partial differential equations compared to other well-documented algorithms.

In this project we will approach the problem of solving one specific partial differential equations, the diffusion equation in one dimension, using two different methods. First we are going to find the explicit solution using the forward euler method, then we will study how a neural network perform to comparison, with varying activation functions and optimizers. We will implement our own code for the explicit scheme, and use the functionality of `tensorflow` to create the neural network, which can be reproduced from the supplementary material found at https://github.com/halvarsu/FYS-STK4155.

We begin with an overview of the theory behind the methods used, covering some of the mathematics behind the functions implemented in the code. After this we will present the results, before a more detailed discussion and conclusion.

2 Theory and methods

2.1 PDEs and Neural networks

In this project we will be studying n'th order partial differential equations (PDEs) of N variables $x_n \in \vec{X}$ on the form

$$D(x_1, x_2, \dots, x_N)u(x_1, \dots, x_N) = f(x_1, \dots, x_N), \qquad (1)$$

subject to boundary conditions, where D is a differential operator involving derivatives of all the variables x_i up to order n, f is a function on the specified domain, and u is the function we want to solve for. Defining $F(\vec{X}) \equiv D(\vec{X})u(\vec{X}) - f(\vec{X})$, then the differential equation eq. (1) becomes

$$F\left(X, \frac{\partial u(\vec{X}))}{\partial x_1}, \cdots, \frac{\partial u(\vec{X})}{\partial x_N}, \frac{\partial^2 u(\vec{X})}{\partial x_1 \partial x_2}, \cdots, \frac{\partial^n u(\vec{X})}{\partial x_N^n}\right) = 0. \qquad (2)$$

Note that F is generally a non-linear function of all possible derivatives up to order N, as it involves the differential operator D. To avoid completely intractable equations, we will be dropping the arguments for some functions (looking at you, F) from here on.

The Universal Approximation Theorem [3] states that any function can be approximated by the output of a neural network with a single hidden layer, $N(\vec{X}, P)$, where P is the weights of the network. However, such an output will in general not respect the boundary conditions of our problem. While this will be solved by enough training, it is better to use the information at hand when designing the network. As such, we define a trial solution u_t to be a function of the network output

$$u_t(\vec{X}) = h_1(\vec{X}) + h_2(\vec{X}, N(\vec{X}, P)), \qquad (3)$$

where h_1 and h_2 are chosen such that the trial solution will fulfill the boundary conditions. We define the cost function to be mean square of F evaluated at the trial function over each sampled point $\vec{X}_i$

$$C = \frac{1}{N}\sum_i \Big(F(\vec{X}_i, u_t(\vec{X}_i), \cdots)\Big)^2. \qquad (4)$$

2.2 Diffusion equation

2.2.1 Analytic

The dimensionless diffusion equation for temperature in a one-dimensional rod of length $L = 1$ can be given by

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \quad t > 0, \quad x \in [0,1]. \qquad (5)$$

We use Dirichlet boundary-conditions, with the edges held at $u = 0$,

$$u(0,t) = u(L,t) = 0, \quad t \geqslant 0. \qquad (6)$$

To fully specify the problem, we also provide an initial distribution at $t = 0$, given by

$$u(x,0) = \sin(\pi x), \quad 0 < x < L. \qquad (7)$$

The general solution can be found by first separating eq. (5) by assuming that the general solution can be written as a sum of separable solutions on the form $\tilde{u}(x,t) = X(x)T(t)$, giving

$$T(t)\frac{\partial^2 X(x)}{\partial x^2} = X(x)\frac{\partial T(t)}{\partial t}$$
$$\Rightarrow \frac{X''(x)}{X(x)} = \frac{\dot{T}(t)}{T(t)}. \tag{8}$$

Observe that varying x in the last equation keeps the right hand side constant, and vice versa for the left hand side when varying t. In other words, both sides are equal to a constant, which we name $-\omega^2$ by reasons soon to become apparent. We then have two ordinary differential equations,

$$X''(x) = -\omega^2 X(x), \tag{9}$$
$$\dot{T}(t) = -\omega^2 T(t). \tag{10}$$

The first has the solution $X(x) = C\cos\omega x + D\sin\omega x$, which given the boundary conditions in eq. (6) have the discrete solutions $X_n(x) = \sin\omega_n x$ with $\omega_n = n\pi$ for $n = 1, 2, \cdots$ (hence the negative constant, as solutions have to tend to zero at two points, $x = 0$ and $x = 1$). This is up to a constant, which will be included later in $T(t)$. Solving the second equation gives the solutions

$$T_n(t) = K_n e^{-\omega_n^2 t} = K_n e^{-n^2\pi^2 t}, \quad \text{for all } n. \tag{11}$$

The eigenfunctions of the diffusion equation are then

$$u_n(x,t) = X_n(x)T_n(t) = K_n e^{-n^2\pi^2 t}\sin n\pi x, \tag{12}$$

and any real function fulfilling the boundary conditions can be written as a sum of these,

$$u(x,t) = \sum_n u_n(x,t) = \sum_n K_n e^{-n^2\pi^2 t}\sin n\pi x. \tag{13}$$

The K_n are then found by imposing some initial conditions and using Fouriers trick. In our case, the initial conditions are $u(x,0) = \sin\pi x$, and we trivially see that $K_1 = 1$ and $K_n = 0$ for $n \neq 1$. The analytic solution is

$$u(x,t) = e^{-\pi^2 t}\sin\pi x. \tag{14}$$

2.2.2 Explicit scheme

A common explicit method for solving PDEs numerically is the forward Euler method. Here x and t in eq. (5) is discretized and the derivatives are approximated with finite differences, where we use first-order forward differences

$$\frac{\partial f(a)}{\partial a} \approx \frac{f(a+\Delta a) - f(a)}{\Delta a}, \tag{15}$$

and second-order central differences

$$\frac{\partial^2 f(a)}{\partial a^2} \approx \frac{f(a+\Delta a) - 2f(a) + f(a-\Delta a)}{\Delta a^2}. \tag{16}$$

This gives eq. (5) on the form

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}, \tag{17}$$

where $u(x_j, t_n) = u_j^n$. Finally, rewriting it on an iterative form

$$u_j^{n+1} = r(u_{j+1}^n + u_{j-1}^n) + (1-2r)u_j^n, \tag{18}$$

where $r = \Delta t/\Delta x^2$ have to satisfy the stability constraint $r \leqslant \frac{1}{2}$.

2.2.3 Neural Network

The cost function for the diffusion equation neural network becomes

$$C = \frac{1}{N_x N_t}\sum_i \left(\frac{\partial u_t}{\partial t} - \frac{\partial^2 u_t}{\partial x^2}\right)^2, \tag{19}$$

where our trial function fulfilling the Dirichlet boundary condition is given as

$$g_t(x,t) = (1-t)\sin(\pi x) + x(1-x)tN(x,t,P). \tag{20}$$

This is implemented in `pde_nn.py` at the repository using `tensorflow` as backend. We used $N_t = N_x = 20$, and a time of $t \in [0,1]$.

2.3 Cost function optimization

To update the weights we would normally calculate the derivative of the cost function with respect to the weights. This becomes very complicated because the cost function is also dependent on the input. To save some trouble, we instead used the inbuilt gradient methods from Tensorflow to define eq. (19) in terms of tensors, which can then be magically differentiated with respect to anything.

The algorithms used to train the neural network are momentum optimization and the highly successful Adam optimization, both stochastic gradient methods. Momentum optimization makes a moving average of the past gradients as the directional vector to accelerate the gradient descent. Potentially this will help it gain speed where the gradients are small, and can dampen oscillations in high-curvature directions which in some cases lead to problems for regular SGD. Another approach to momentum is the Nesterov Accelerated Gradient, where the parameters used in the update is their expected value with the current momentum, instead of their actual value. Adam is a first-order gradient-based algorithm aimed at non-convex optimization problems gaining its name due to the use of adaptive learning rates from estimates of first and second moments of the gradient [4].

2.4 Hyperparameter optimization

Tuning of hyperparameters is one of the challenges when dealing with neural networks. For large parameter spaces, a grid search can become inefficient, as many of the architectures are bad and give poor results. Instead, we use a Python library called `Hyperopt` [1] to search the parameter space more cleverly. Using `Hyperopt` it is then trivial to define parameter searches, either discrete or continuous. The method used was a so called tree-structured Parzen estimator [2]. We decided to run separate runs with 200 evaluations of different architectures for the Adam and Momentum optimizers. Each evaluation of the network involved 10000 epochs of training. For the momentum optimizer we also added a optimization of momentum between 0 and 1 with steps of 0.1, and a choice of using Nesterov momentum or normal momentum. Our hyperopt grids can be seen in tables 1 and 2 for Adam and Momentum optimization respectively.

Table 1: The parameters spaces of our hyperparameter optimization with the Adam optimizer from Tensorflow. Here λ is the learning rate, the layer sizes are given logarithmically linear between S_0 for the first layer and $S_L = 10 + \gamma_S(S_0 - 10)$ for the last layer with γ_S as a factor, and N_h is the number of hidden layers. The distributions are from the `Hyperopt` package and are described at the package wiki (link).

Parameter	Distribution	Parameter space
$\log_{10}\lambda$	quniform	$-5, -4, -3, -2, -1$
S_0	quniform	$10, 12, \cdots, 128$
γ_S	quniform	$[0, 1]$, step 0.01
N_h	randint	$1, 2, 3, 4$
Activation	choice	{Sigmoid, tanh, RELU}

Table 2: The parameters spaces of our hyperparameter optimization with the Momentum optimizer from Tensorflow. See table 1 for description of parameters.

Parameter	Distribution	Parameter space
$\log_{10}\lambda$	quniform	$-5, -4, -3, -2, -1$
S_0	quniform	$10, 12, \cdots, 128$
γ_S	quniform	$[0, 1]$, step 0.01
N_h	randint	$1, 2, 3, 4$
Activation	choice	{Sigmoid, tanh, RELU}
Momentum	quniform	$[0, 1]$, step 0.1
Nesterov?	choice	{True, False}

3 Results

3.1 Forward Euler method

In fig. 1 and fig. 2 the solutions of the forward euler method are plotted at two different times, for two different values of Δx, along with the analytical solution for comparison. With $\Delta x = \frac{1}{10}$ we got a gave a mean squared error equal to 5.2×10^{-4}, while $\Delta x = \frac{1}{100}$ gave a mean squared error equal to 5.1×10^{-6}. The time points are chosen close to $t = 0$, where the solution are significantly curved, and close to $t = 1$ where the solution is relatively linear compared to the first time point.

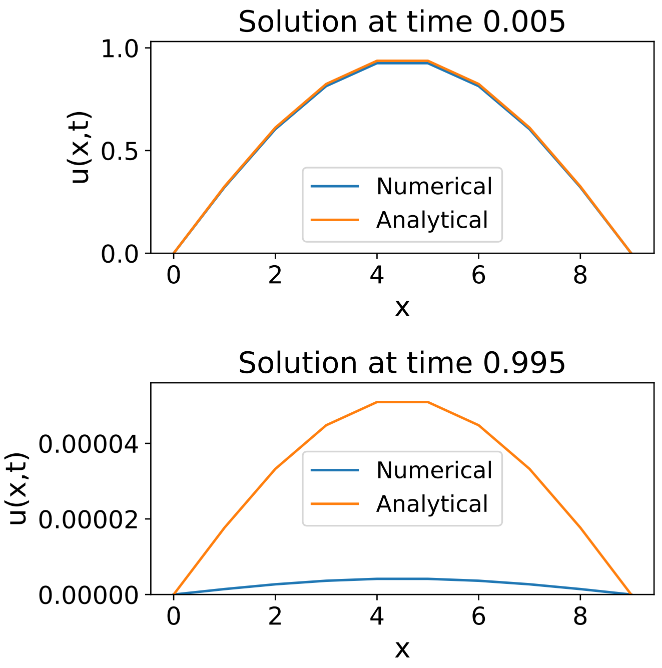


Figure 1: Numerical and analytical solution of the diffusion equation at two different times, with $\Delta x = \frac{1}{10}$

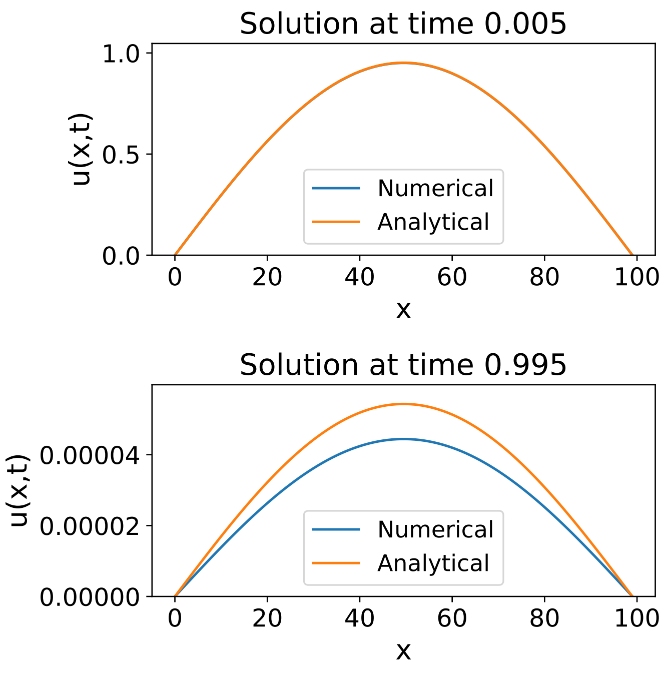


Figure 2: Numerical and analytical solution of the diffusion equation at two different times, with $\Delta x = \frac{1}{100}$.

Figure 3 show the solution $u(x,t)$ for a fixed value of x. The relative errors between the numerical solutions and the analytical solution are plotted in fig. 4.

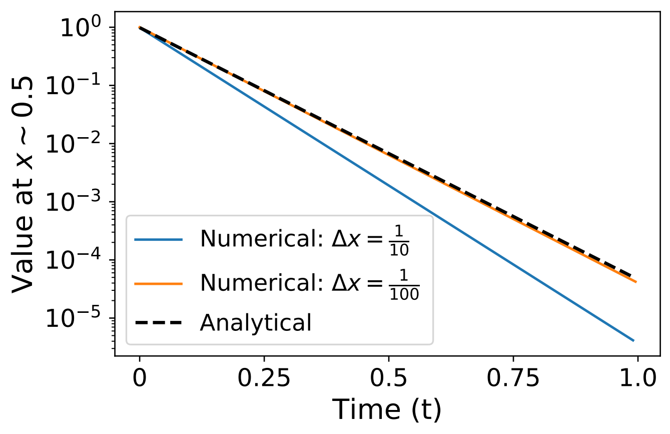


Figure 3: Solutions of the diffusion equation, with numerical solutions for $\Delta x = \frac{1}{10}$ and $\Delta x = \frac{1}{100}$, with fixed value $x \sim 0.5$.

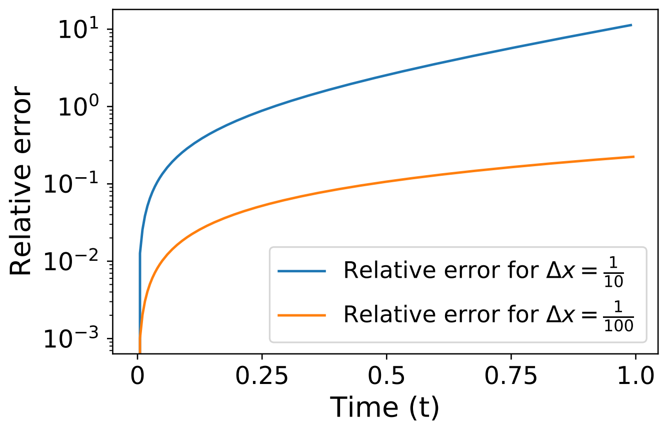


Figure 4: Relative error between numerical solutions for $\Delta x = \frac{1}{10}$ and $\Delta x = \frac{1}{100}$ and analytical solution.

3.2 Neural networks

The lowest cost function achieved was $C = 9.5 \times 10^{-5}$, using tanh as cost function, a learning rate of $\log_{10} \lambda = -3$. The lowest MSE on the other hand was 1.5×10^{-8}, also with Adam optimizer but with sigmoid as activation function and a learning rate of $\log_{10} \lambda = -4$.

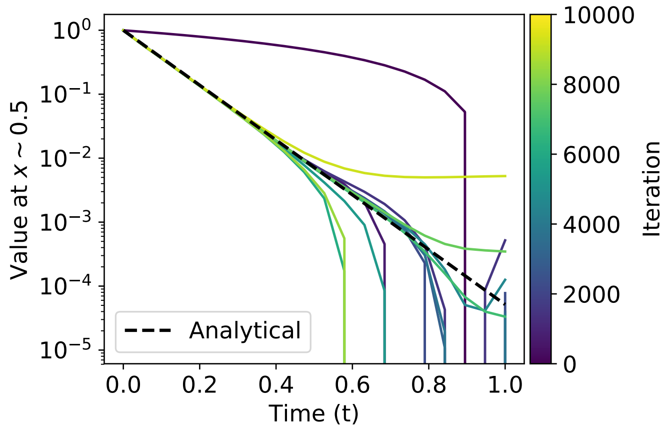
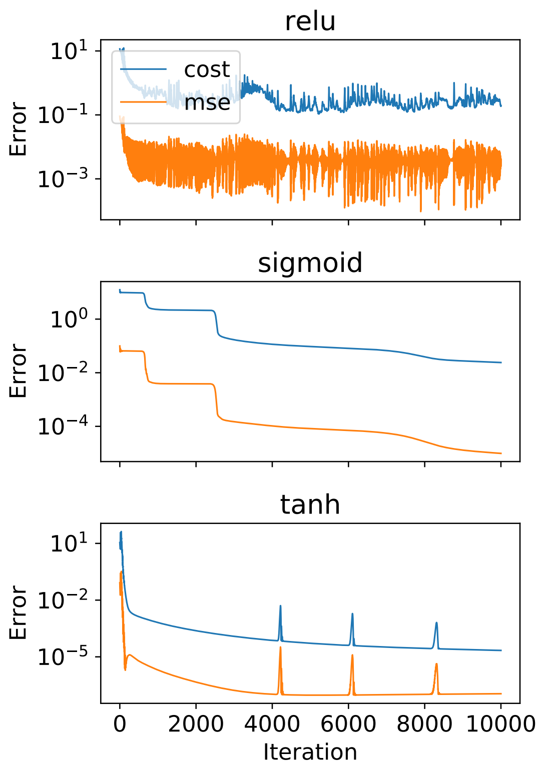


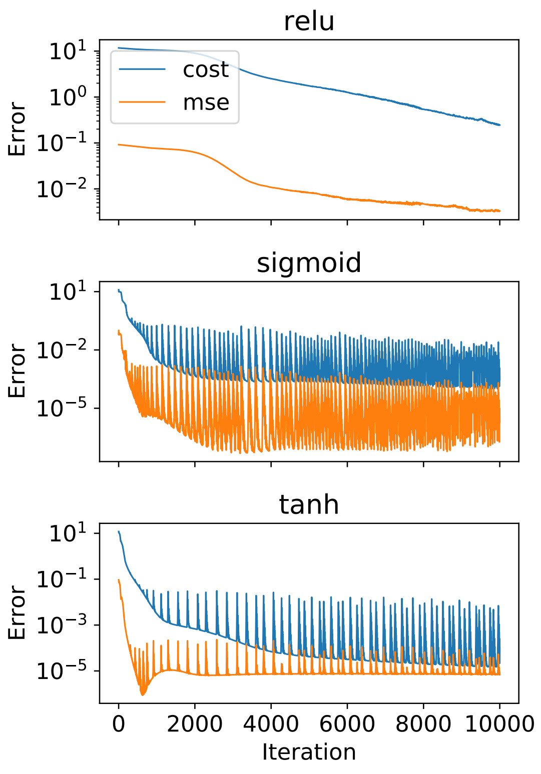
Figure 5: Convergence of the trial function towards the analytical solution. Some of iterations are better than the final solution due to the jittering away from the optimum seen clearly in fig. 6a

.

The development of cost and MSE as function of iteration for the different activation function can be seen in figs. 6a and 6b for Adam and Momentum optimizers respectively, where the best solution for each method is plotted. Their parameters are listed in table 3.



(a) Adam optimizer.



(b) Momentum optimizer

Figure 6: The cost function for each iteration for the best architecture with the two optimizers, for all three activation functions

Table 3: The architecture of the best run during the hyper search, for each of the activation functions.

	tanh	sigmoid	RELU
		ADAM	
N_h	2	3	3
Accumulated size	302	427	431
λ	10^{-3}	10^{-2}	10^{-5}
C	9.5×10^{-6}	6.8×10^{-5}	0.098
MSE	4.2×10^{-6}	3.6×10^{-7}	0.0037
	tanh	sigmoid	RELU
		Momentum	
N_h	3	3	2
Accumulated size	370	302	194
λ	10^{-2}	10^{-2}	10^{-2}
C	2.2×10^{-5}	0.02	0.2
MSE	1.1×10^{-7}	9.8×10^{-6}	0.004
Momentum	0.9	0.9	0.2
Nesterov?	False	False	True

For an overview over the five best solutions found by the hyper optimization in terms of cost function and mse, the reader is directed towards tables 4 and 5 respectively for the Adam optimizer, and in tables 6 and 7 for the Momentum optimizer.

Further, one can find histograms over the loss per iteration for the three different activation functions, for both optimizers, in fig. 7.

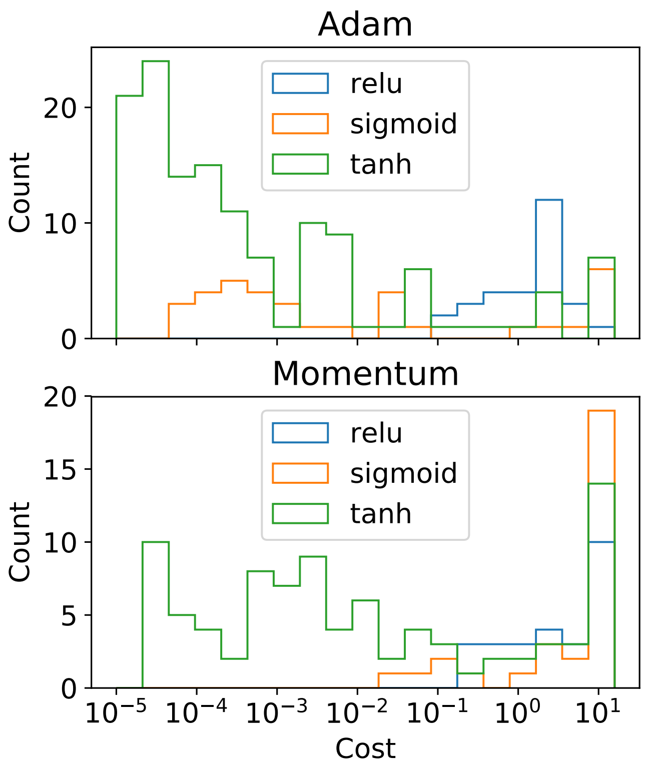


Figure 7: Performance of the three different architectures during the hyperparameter optimization for both optimizers. Tanh is clearly superior for both optimizers.

Table 4: The five architectures giving the lowest loss function for the Adam optimizer.

Act.	Layers	$\log_{10}\lambda$	Loss	MSE
tanh	[103, 101, 98]	-3	1.0×10^{-5}	4.2×10^{-6}
tanh	[119, 112, 104]	-3	1.3×10^{-5}	6.0×10^{-6}
tanh	[111, 105, 98]	-3	1.4×10^{-5}	4.1×10^{-6}
tanh	[118, 114, 111]	-3	1.4×10^{-5}	5.1×10^{-6}
tanh	[119, 112, 104]	-3	1.5×10^{-5}	5.9×10^{-6}

Table 5: The five architectures giving the best MSE for the Adam optimizer.

Act.	Layers	$\log_{10}\lambda$	Loss	MSE
sigmoid	[77, 70, 64]	-3	2.7×10^{-4}	1.5×10^{-8}
sigmoid	[127, 113, 100]	-3	5.0×10^{-4}	5.2×10^{-8}
sigmoid	[79, 54, 37, 25]	-3	3.7×10^{-4}	6.7×10^{-8}
sigmoid	[111, 69, 42]	-3	1.6×10^{-4}	7.1×10^{-8}
tanh	[83]	-2	2.2×10^{-4}	9.7×10^{-8}

Table 6: The five architectures giving the lowest cost function for the Momentum optimizer.

Act.	Layers	$\log_{10}\lambda$	Loss	MSE
tanh	[118, 99, 83, 70]	-2	2.2×10^{-5}	1.1×10^{-7}
tanh	[103, 79, 60, 46]	-2	2.7×10^{-5}	9.2×10^{-7}
tanh	[89, 85, 80, 76]	-2	2.9×10^{-5}	8.9×10^{-7}
tanh	[92, 74, 60, 49]	-2	3.1×10^{-5}	2.8×10^{-6}
tanh	[88, 74, 63, 53]	-2	3.2×10^{-5}	1.7×10^{-7}

Table 7: The five architectures giving the best MSE for the Momentum optimizer.

Act.	Layers	$\log_{10}\lambda$	Loss	MSE
tanh	[42, 39, 36, 34]	-2	4.9×10^{-5}	2.9×10^{-8}
tanh	[118, 99, 83, 70]	-2	2.2×10^{-5}	1.1×10^{-7}
tanh	[88, 74, 63, 53]	-2	3.2×10^{-5}	1.7×10^{-7}
tanh	[40, 33, 27, 22]	-2	6.2×10^{-5}	2.7×10^{-7}
tanh	[54, 50, 46, 43]	-2	4.2×10^{-5}	4.9×10^{-7}

3.2.1 Momentum

The parameters of momentum for the Momentum optimizer in our hyperparameter search can be seen in fig. 8. The momentum optimization could probably be tuned a bit, with more higher resolution than the current 0.1 and restrictions on an interval closer to 1, as 0.9 is by far the best solution.

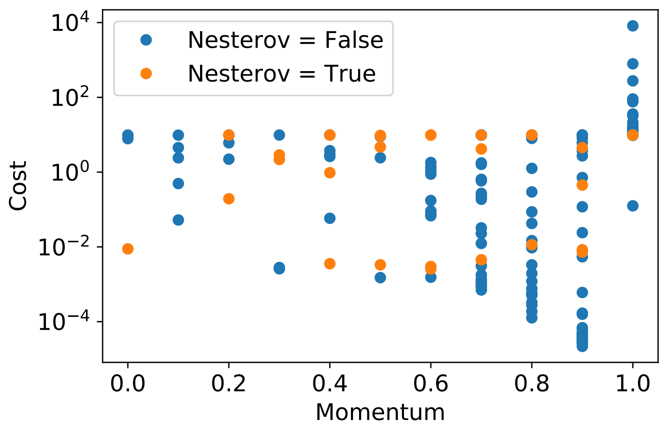


Figure 8: Cost as function of momentum in the sampled architectures found in our hyper parameter search.

4 Discussion

4.1 Forward Euler method

As seen in section 3.1 the results were much better for $\Delta x = \frac{1}{100}$ than $\Delta x = \frac{1}{10}$, with lower relative error. This is also clear from looking at fig. 1 and fig. 2 where both numerical solutions decay too quickly compared to the analytical. It is apparent that the difference between the two numerical solutions are somewhat scaled, as a result of different Δx-values. The use of finite differences means that there will be local error with each iteration. First order forward differences for the time derivative gives local error proportional to Δt, which from the stability constraint is proportional to Δx^2, and the second order central differences also gives local error proportional to Δx^2. Smaller values of Δx will be desired. However this comes at the cost of computation time, where the number of iterations increases proportional to $\frac{2}{\Delta x^3}$.

4.2 Neural networks

The neural network solution gave better results than Forward Euler for the same grid size. The smallest grid size of the Forward Euler method was 2000, a factor 5 more than what the network had available. We can probably conclude that the strength of the neural network is its ability to find the correct solution given few data points. However it comes with a great computational cost. Especially the learning phase is very costly, and it is relevant to compare the whole process of training the network with solving the equation explicitly. Of the optimization methods, Adam produces faster convergence but more unstable solutions than a optimized Momentum optimizer with momentum of 0.9. Some more experimenting with Momentum could be useful, letting it take values of momentum between 0.9 and 1 might give better results.

We observe that all the best solutions for both optimizers were with tanh as activation function, as seen in fig. 7. The reason for this is unknown, but it might have something to do with the sharp exponential decline of the solution. Sigmoid performs well, but has the most jittering when using the Adam-optimizer. With momentum, the best sigmoid solution learns too slowly. Relu is unstable for higher learning rates, and does not converge in time for our tests with 10000 iterations. It would be interesting to see if relu also exhibits the same erratic behaviour as the other ones at lower values of the cost function (i.e even more iterations). We also tested elu, which has an exponential cutoff at zero, but it did not converge in our test cases so it was not kept as a valid solution. As a general comment on the solutions of this problem, most of the well performing networks were in the upper end with respect to size, and a learning rate of $\lambda = 10^{-3}$ is prevalent.

The jittering in the cost function is bad for our hyperoptimization runs because a good architecture can be discarded if the last iteration suddenly jumps. Keeping some memory of the weights in the network over the iterations might give better results, because we can then choose the best architecture from the last few iterations.

MSE and cost do not always follow perfectly, as seen in the start of tanh of fig. 6a or in fig. 5. There the best solution with the lowest MSE is actually found after less than 1000 iterations. This is probably because the numerical derivatives are more sensitive to perturbations than the squared error. In a way the current loss function also optimizes for smoothness in the trial function, and not only distance from correct result. Future work might be to test other methods better suited to finding this smoothness. Another potential improvement is

One next step could be to use other types of architectures. Given our limited number of evaluations, a larger hyperoptimization search could possibly lead to different results, as we only searched a small number of architectures. Even more potential improvement lies in experimenting with the form of the trial function, and a better mathematical background on the choice of trial function would be very welcome. Another idea for future work is to include more prior knowledge of the spatial topology of the problem. Our network considers all the points equally and disconnected, with no structure. As such one can argue that the network does not learn the derivatives of the problem, but only fits the solution. It might be interesting to test a convolutional network, as it provides local information for each convolutional layer. This would let the network directly represent the second and first derivatives.

5 Conclusion and future work

The neural network performed better than Forward Euler for lattice sizes of the same order of magnitude, with a resulting MSE ranging between 4.2×10^{-6} to 1.1×10^{-7}, compared to the forward Euler method with MSE equal to 5.1×10^{-6}. While Forward Euler is a very basic explicit method, this shows that the strength of the network approach is that it is able to adapt to the problem with litte prior knowledge. This might be utilized on harder problems where an analytical solution is not available. An interesting extension would be to find a problem where most numerical solutions have difficulties while the neural networks work, or on the other side find the limits where the neural network does not provide a good solution. The fact that the network was trained for a smaller lattice of 20×20 and outperformed the Euler method with a lattice equal to 100×20000 may suggest that a neural network can perform better where sparsity is necessary, i.e. in higher dimensional problems. Finally, depending on the problem, the best solution can also be dependent on the relative importance of precision versus runtime.

References

[1] Distributed Asynchronous Hyperparameter Optimization in Python: github.com/hyperopt/hyperopt, December 2018. original-date: 2011-09-06T22:24:59Z.

[2] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[3] Balázs Csanád Csáji. Approximation with artificial neural networks.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.