

# Solving Partial Differential Equations Using Physics-Informed Neural Networks

## A Case Study on the One-Dimensional Heat Equation

Simen L. Wærstad\*  
*University of Oslo*  
 (Dated: December 18, 2025)

Artificial Neural networks (ANNs) have in recent years become an increasingly popular tool for solving differential equation. - - - -

### I. INTRODUCTION

Humankind's desire to understand and describe the world around us often leads to differential equations. They thus appear in a vast array of fields, ranging from physics to biology, and finance. However, being notoriously difficult to solve and often without any analytical solution to be found, the solutions of these systems often rely on numerical approximations. The study of such methods and their solutions are therefore of great importance in addressing many of our most fundamental questions.

Advancements in computational power and theoretical work such as the Universal Approximation Theorem by Cybenko [1] suggested that ANNs could be used in solving differential equations.

We will here consider a FFNN

### II. METHODS

#### A. Theory

##### 1. The Heat Equation

The one-dimensional heat equation can be written as

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

for  $t > 0$  and  $x \in [0, L]$ . We assume the Dirichlet boundary conditions, that is, we assume

$$u(0, t) = 0 \quad \text{and} \quad u(L, t) = 0 \quad t \geq 0.$$

For the given boundary conditions and the initial condition

$$u(x, 0) = \sin(\pi x),$$

it can be show that the analytical solution is given by:

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

##### 2. The Forward Euler Scheme

The forward Euler scheme, also known as the, explicit scheme is a traditional method for solving partial differential equations. By first dividing the spatial and temporal dimensions into discrete points separated by a constant step size, the forward Euler scheme uses the spatial values at time step  $n$  to compute an approximation to the solution at time step  $n + 1$ .

Consider the case of the heat equation. Then, the discretization is carried out as follows: pick a spatial step size  $\Delta x$  and a temporal step size  $\Delta t$ . Then we can discretize the continuous variables  $x$  and  $t$  by defining  $x_i = i\Delta x$  and  $t_i = i\Delta t$ . This splits the spatial interval  $(0, L)$  into  $L/\Delta x + 1$  points and similarly the temporal interval  $(0, T)$  is split into  $T/\Delta t + 1$  points. We can now discretize  $u(x, t)$  by defining  $u_i^j = u(x_i, t_j)$ .

We now need to discretize the heat equation itself. To do this we must first find discretized expressions for each partial derivative. Using standard approximations of derivatives we have that

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{j+1} - u_i^j}{\Delta t},$$

and

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{\Delta x^2}.$$

Substituting these approximations into equation 1 and isolating  $u_i^{j+1}$  we get

$$u_i^{j+1} = ru_{i-1}^j + (1 - 2r)u_i^j + ru_{i+1}^j.$$

Keeping in mind our boundary conditions and defining  $r = \Delta t/\Delta x^2$  we can rewrite this as a matrix equation

$$\mathbf{u}^{j+1} = A\mathbf{u}^j$$

, where  $\mathbf{u}^j$  is the column vector containing  $u_i^j$  for all  $i$ , and  $A$  is the tridiagonal matrix with  $1 - 2r$  on the diagonal and  $r$  on the super- and subdiagonal. Thus, solving our partial differential equation reduces to iteratively multiplying  $\mathbf{u}^i$  by the matrix  $A$ .

It can be shown that in order for this scheme to be stable it is required that  $\Delta t/\Delta x^2 \leq 1/2$ . Practically, this means that if we chose a desired spatial resolution, then we are forced to pick significantly greater temporal resolution, making the computation notably more expensive. For example,  $\Delta x = 1/100$  forces  $\Delta t \leq 1/20000$ .

---

\*<https://github.com/Waerstad/FYS-STK4155-Project-3>

Due to the approximations chosen for the first and second partial derivatives of  $u$  the scheme has global truncation error  $\mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta t)$ .

### 3. Solving PDEs with Neural Networks

Neural networks can be used to solve partial differential equations by altering the loss function to include information about the equation at hand.

The network is structured as follows: The input layer consists of one node per variable we are interested in. In the heat equation case, this would be one node for  $x$  and one for  $t$ . The network then has some amount of hidden layers with a chosen width. The output layer contains only one node. The output value  $N$  of the network is then fed into a trial solution  $\hat{u}(x, t, N)$  which is constructed so that it forces the neural network output to obey the boundary and initial conditions. In the case of the heat equation we have that

$$\hat{u}(x, t, N) = (1 - t) \sin(\pi x) + x(1 - x)N.$$

If we rearrange partial differential equation we want to solve to the form

$$F(u, x, t) = 0$$

where  $F$  is a function of  $x, t$  and  $u$  as well as the partial derivatives of  $u$  with respect to  $x$  and  $t$ . Hence, finding the  $u$  such that  $F = 0$  is equivalent to finding the  $u$  that minimizes  $F(u, x, t)^2$ . Thus, for  $N$  input pairs  $(x_i, t_i)$  the loss function becomes

$$\mathcal{L}(\hat{u}, \mathbf{x}, \mathbf{t}) = \frac{1}{N} \sum_{i=0}^N F(\hat{u}, x_i, t_i)^2.$$

In the case of the heat equation we have

$$\mathcal{L}(\hat{u}, \mathbf{x}, \mathbf{t}) = \frac{1}{N} \sum_{i=0}^N \left( \frac{\partial \hat{u}(x, t)}{\partial t} - \frac{\partial^2 \hat{u}(x, t)}{\partial x^2} \right)^2. \quad (3)$$

Thus, minimizing the loss  $\mathcal{L}$  by using some form gradient descent with respect to the neural network parameters should yield a neural network that solves the partial differential equation given by  $F$ . Once trained, it can predict new points at little extra cost.

## B. Implementation

### 1. The Explicit Scheme

The explicit scheme was implemented in python and is relatively straight forward. A short summary of the algorithm is shown in algorithm1

To verify that our implementation was corrected we computed solutions for  $t \in (0, 0.30)$  using  $\Delta x = 1/10$  and

### Algorithm 1 Explicit Scheme

---

```

procedure SOLVEPDE( $\Delta x, \Delta t$ , duration  $T$ )
   $\mathbf{u} \leftarrow \text{INITIALIZESTATE}(\Delta x)$ 
   $A \leftarrow \text{INITIALIZESOLVERMAT}(\Delta x, \Delta t)$ 
   $N = T/\Delta t + 1$ 
  for  $i \in \{1, \dots, N - 1\}$  do
     $\mathbf{u} \leftarrow A\mathbf{u}$ 
  SAVE( $\mathbf{u}$ )

```

---

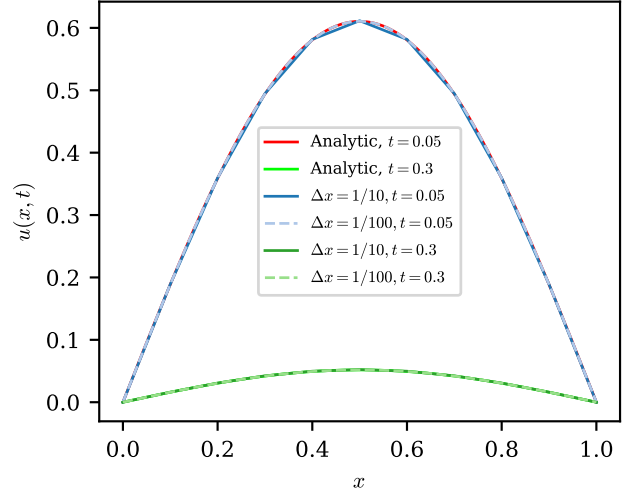


Figure 1: Two numerical solutions to the heat equation computed by the explicit scheme compared to the analytical solutions at  $t = 0.05$  and  $t = 0.30$ . The time step was given by  $\Delta t = \Delta x^2/10$ .

$1/100$  and using  $\Delta t = \Delta x^2/10$ . We chose  $\Delta t < \Delta x^2/2$  to avoid floating point errors pushing the step size above the stability limit. The results were then compared to the analytical solution at  $t = 0.05$  and  $t = 0.30$ . A plot of the numerical and analytical solutions can be seen in Figure 1. For  $\Delta x = 1/10$  the mean squared error was  $4.53 \times 10^{-7}$  and  $1.18 \times 10^{-7}$  at  $t = 0.05$  and  $t = 0.30$  respectively. For  $\Delta x = 1/100$  the MSE was  $4.86 \times 10^{-11}$  at  $t = 0.05$  and  $1.26 \times 10^{-11}$  at  $t = 0.30$ . Thus, the explicit scheme provides a good approximation to the analytical solution.

### 2. The Neural Network

The neural network was implemented using the neural network functionalities of the software library **tensorflow** and its high level API **keras**. Using *layers* and *Model* objects we constructed fully connected feed forward neural networks with the custom loss function given by equation 3. Our implementation only allows for hidden layers with the same number of nodes. For the output layer we used a linear activation function. The

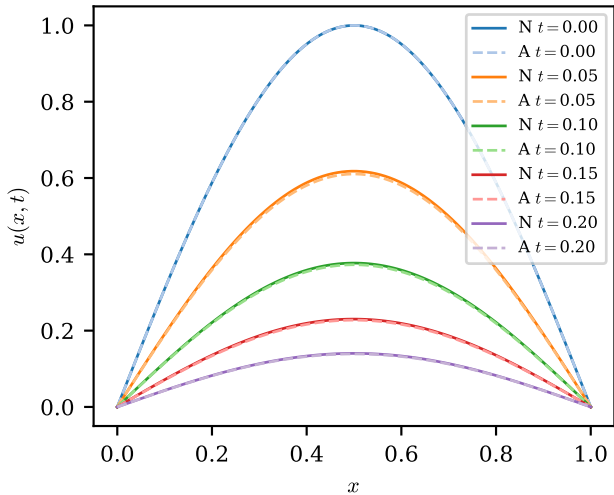


Figure 2: Comparison of neural network solutions  $N$  and compared to analytical solutions  $A$  and different times  $t$ .

neural network implementation is flexible and can easily be adapted to solve other PDEs. The code is available on our GitHub repository.

To initially test the neural network implementation we trained a neural network with 4 hidden layers, each consisting of 40 nodes, with hyperbolic tangent as the activation function. The Adam optimizer [2] with learning rate  $\lambda = 0.001$  was used to train the network. No regularization was added. To train the network, a linearly spaced grid of  $101 \times 101$  points in the region  $[0, 1] \times [0, 0.3]$  of the  $xt$ -plane was used. A plot comparing the numerical solutions of the neural network and the analytical solutions can be seen in Figure 2. For predicting 100 evenly spaced  $x$ -values on  $[0, 1]$  was used for each time  $t$ . The plot shows a good agreement between the neural network's solutions and the analytical solutions.

### III. RESULTS AND DISCUSSION

We performed different grid searches to tune the network. In all runs the Adam optimizer was used for 1000 epochs. For each neural network configuration in the grid searches the mean squared error was computed by comparing the solution at 101 linearly spaced  $x$  values over the interval  $(0, 1)$  and  $t = 0.3$ . Due to limited computational resources each grid search was only performed once. While the runs were seeded, factors such as favorable parameter initialization might make what is on average a worse performing network configuration perform better than what is on average a superior network configuration.

We first carried out a grid search where the activation function and number of hidden layers changed. This combination was used as the activation functions react differently to increasing the number of layers as some

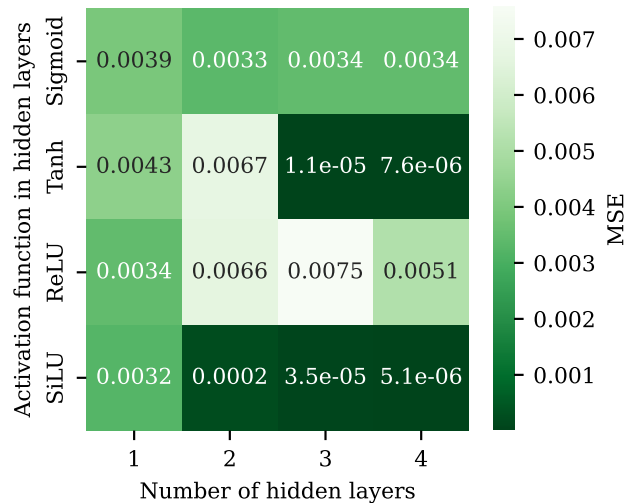


Figure 3: Results of a grid search where the activation functions and the number of hidden layers varied.

have greater problems with vanishing or exploding gradients. The activation functions considered were sigmoid, tanh, ReLU and SiLU. Here SiLU refers to the Sigmoid Linear Unit given by  $\text{SiLU}(x) = x \cdot \text{sigmoid}(x)$ . Numbers of hidden layers from 1 to 4 were tested. In all cases the layer size was kept constant at 40 and the learning rate at 0.001. No regularization was done on the model parameters. The resulting heatmap can be seen in Figure 3.

From Figure 3 we see that sigmoid and ReLU performs particularly bad. In the case of ReLU is expected as it is well known to be prone to problems with vanishing gradients. In addition, it is not continuous, which could cause issues when evaluating the partial derivatives in the loss function. The possible presence of these issues are further supported by the fact that the error increases with the number of layers. For input values away from zero, the derivative of the sigmoid function also becomes zero, which could be contributing to the bad performance.

We see that both hyperbolic tangent and SiLU perform the best, with SiLU slightly outperforming tanh. These are both smooth functions, this could partly be why they perform well. In both cases we see the performance increase as the number of layers increase, indicating that the models do not have issues with vanishing gradients.

We performed another grid search, varying the number of nodes and the number of layers. The activation function was fixed to SiLU as that was best performing. The besides the hyperparameters varied in the grid search and the activation function, the same neural network configuration was used as in the previous search. The result of the grid search can be seen in Figure 4. We see that overall 4 layers performed the best. There also seems to be a general trend of higher number of nodes performing better. However, the best performing combination was 4 layers with 60 nodes each.

In addition, we performed grid searches with L1 and

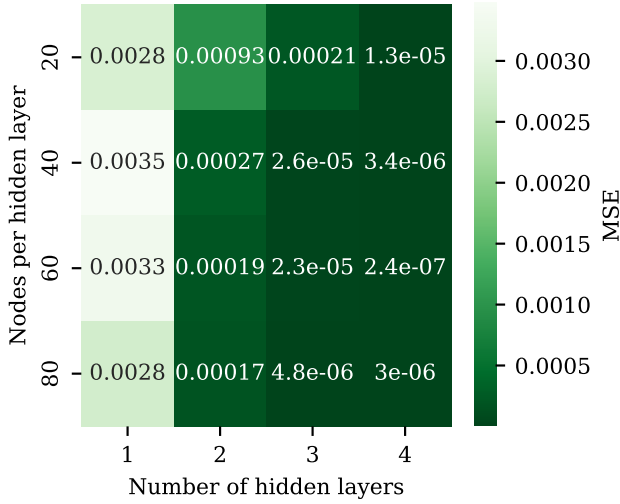


Figure 4: Results of a grid search where the activation functions and the number of hidden layers varied. The activation function in the hidden layers was SiLU.

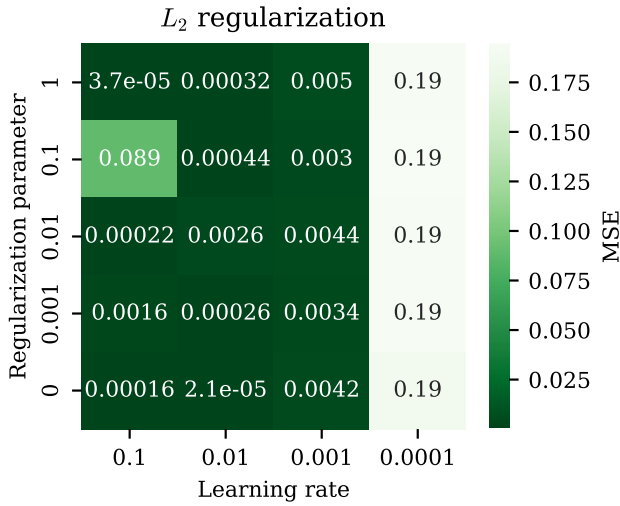


Figure 5: Results of a grid search where the activation functions and the number of hidden layers varied. The activation function in the hidden layers was SiLU.

$L_2$  regularization where the regularization parameter and learning rate were varied. For the search we used SiLU

as the activation function and 4 hidden layers with 60 nodes each. In both the case of  $L_1$  and  $L_2$  regularization we observed the same behavior. For the two cases the results were identical up to rounding. The results for  $L_2$  regularization can be seen in Figure 5.

The grid search shows that no regularization with learning rate 0.01 performs best. Generally there seems to be no clear trend in the impact of the regularization parameter. In particular, the combination of learning rate 0.1 with regularization parameter of  $\lambda = 0.1$  and

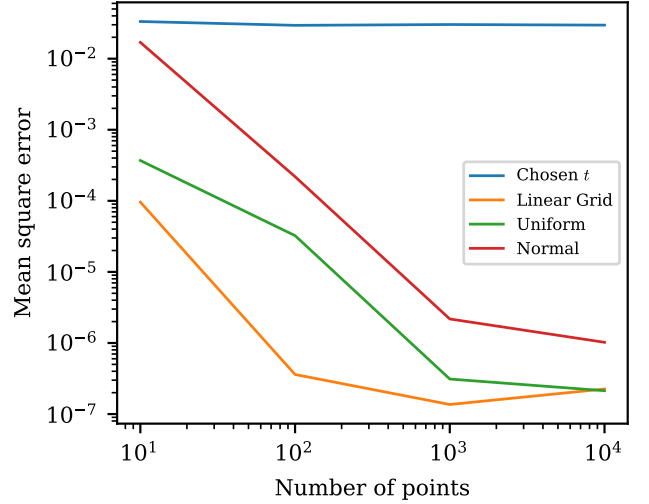


Figure 6: Caption

$\lambda = 1$  are outliers. The choice  $\lambda = 0.1$  gives significantly higher MSE compared to other values of  $\lambda$  at the same learning rate. In contrast,  $\lambda = 1$  gives significantly better results. Since there is no clear trends with regards to the regularization, the variation might be due to chance as a result of the chosen seed and the parameter initialization. In addition, since we are working with synthetic noise-free data and independent variables  $x$  and  $t$ , then it is a priori unlikely that regularization should have a positive effect. Therefore we do not add regularization for the rest of the paper.

#### IV. CONCLUSION

- 
- [1] G. Cybenko, Mathematics of Control, Signals and Systems **2**, 303 (1989).
  - [2] D. P. Kingma and J. Ba, *Adam: A method for stochastic*

*tic optimization* (2017), 1412.6980, URL <https://arxiv.org/abs/1412.6980>.

## Appendix A: Derivations