Solving Partial Differential Equations Using Neural Networks: A Case Study on the 1-Dimensional Heat Equation

Simen L. Wærstad*
University of Oslo
(Dated: December 19, 2025)

Artificial neural networks have become a useful tool in solving PDEs. This has wide reaching applications as PDEs are ubiquitous in the sciences. One method of building neural networks for this purpose involves encoding the PDE residual into the loss function [1]. This yields a flexible PDE solver that once trained, can predict the solution at new points at little extra computational cost. Here we present a case study of a neural network based PDE solver applied to the heat equation. We use this as a basis for comparison with the classical forward Euler scheme, also known as the explicit scheme. We show that the neural network based solver does not outperform the forward Euler scheme. The best case neural network model roughly equaled the performance of the worst performing forward Euler scheme, both achieving an MSE on the order of 10^{-7}. In addition, we investigate the effect of different methods of generating the training data. Hence, we demonstrate a practical example of a neural network based PDE solver and highlight some benefits and downsides compared to traditional numerical schemes.

I. INTRODUCTION

Partial differential equations (PDEs) are central in every field of science. In applied mathematics numerical solutions to partial differential equations have been a central area of study with wide reaching applications. In the past decades neural networks have proven to be useful tools in solving partial differential equations.

One way of solving partial differential equations using neural networks is by encoding the residual of the equation into the loss function that is to be minimized by the training process. [1]. Once trained, the network can give numerical solutions at new points for little extra computational cost. Thus, this create a flexible PDE solver at the expense of the upfront computation cost associated with training.

We implement our own neural network PDE solver and evaluate its performance. In order to evaluate the network's performance we also implement the explicit scheme, a simple numerical scheme for solving PDEs, and compare its performance to that of the network. We apply these methods to solve the Heat Equation for a given initial condition and with Dirichlet boundary conditions.

In subsection II A we give the necessary background to the heat equation and its analytical solution under the considered initial and boundary conditions. We also go through the theory behind the forward Euler scheme and how it is used to solve the heat equation. Lastly, in this section we discuss the theory behind solving PDEs with neural networks. In subsection II B we go through details of the implementations of the forward Euler scheme and the neural network. In section III we discuss the results of doing a series of grid searches to find well-performing hyperparameter choices. We also discuss the effect of different methods of sampling the points used for training. Using these hyperparameters we compare the performance of the neural network with that of the forward Euler scheme. Lastly, in section III we present our conclusions and suggestions for future work.

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

II. METHODS

A. Theory

1. *The Heat Equation*

The one-dimensional heat equation can be written as [2]

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

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

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

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:

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

2. *The Forward Euler Scheme*

This section is based on notes by M. Hjorth-Jensen [3] and A. Kvellestad [4]

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 Δx and a temporal step size Δt. 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

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

, where $\boldsymbol{u}^j$ 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 $\boldsymbol{u}^i$ 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 $\Delta t \leq 1/20000$.

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. We will do this as presented in [1] and [2].

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 as some amount of hidden layers with a chosen width. The output layer contains only one node. The output value N of the network is 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, the for N input pairs (x_i, t_i) the loss function becomes

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

In the case of the heat equation we have

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

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

Algorithm 1 Explicit Scheme

```
procedure SolvePDE($\Delta x, \Delta t$, duration $T$)
    $\boldsymbol{u} \leftarrow$ InitalizeState($\Delta x$)
    $A \leftarrow$ InitalizeSolverMat($\Delta x, \Delta t$)
    $N = T/\Delta t + 1$
    for $i \in \{1, \ldots, N-1\}$ do
        $\boldsymbol{u} \leftarrow A\boldsymbol{u}$
        Save($\boldsymbol{u}$)
```

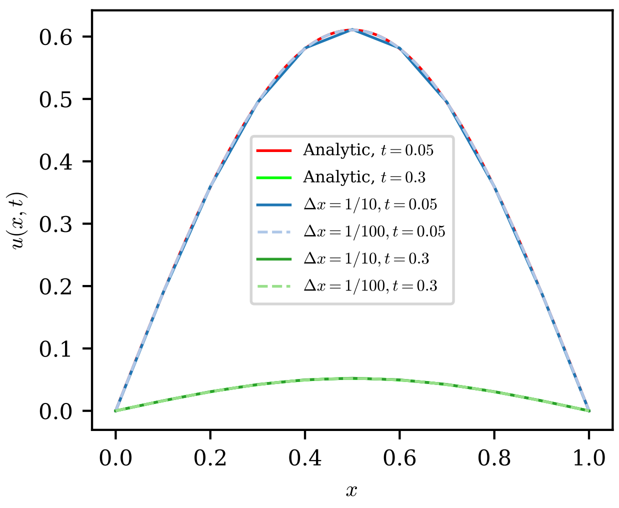

Figure 1: Two numerical solutions to the heat equation computed by the explicit scheme compared to the analytical solutions at $t = 0.05$ and 0.30. The time step was given by $\Delta t = \Delta x^2/10$.

To verify that our implementation was corrected we computed solutions for $t \in (0, 0.30)$ using $\Delta x = 1/10$ and 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 4.53×10^{-7} and 1.18×10^{-7} at $t = 0.05$ and $t = 0.30$ respectively. For $\Delta x = 1/100$ the MSE was 4.86×10^{-11} at $t = 0.05$ and 1.26×10^{-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` [5] and its high level API `keras` [6]. 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 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 [7] 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 101×101 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.

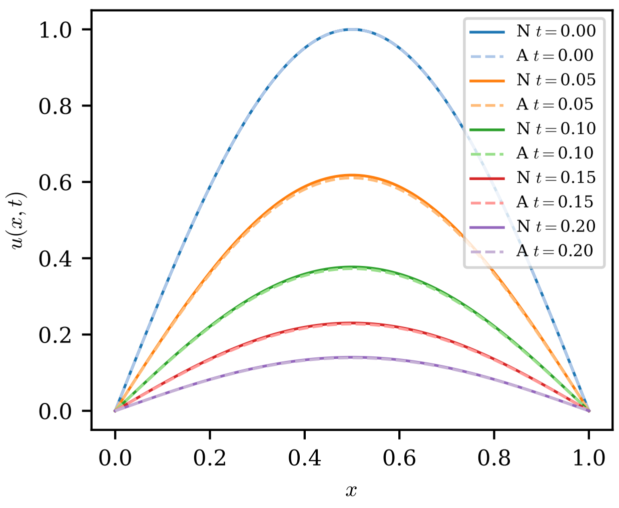

Figure 2: Comparison of neural network solutions N and compared to analytical solutions A and different times t.

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 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 $\mathrm{SiLu}(x) = x\,\mathrm{sigmoid(x)}$. Numbers of hidden layers form 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

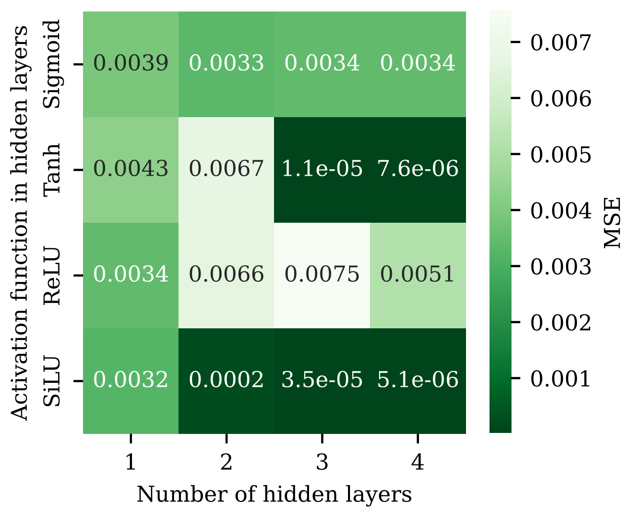

Figure 3: Results of a grid search where the activation functions and the number of hidden layers varied.

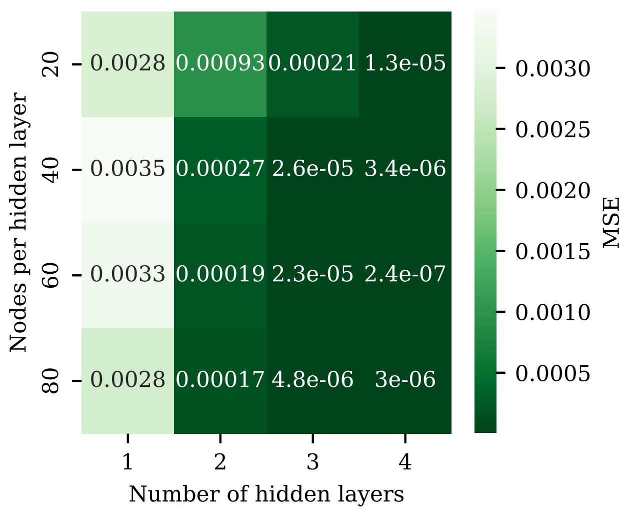

Figure 4: Results of a grid search where the activation functions and the number of hidden layers varied. The activation function in the hidden layers was SiLU.

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 actvation 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 L2 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 L1 and L2 regularization we observed the same behavior. For the two cases the results were identical up to rounding. The results for L2 regularization can be seen in Figure 5.

The grid search shows that no regularization with learning rate 0.01 performs best. Generally there seems

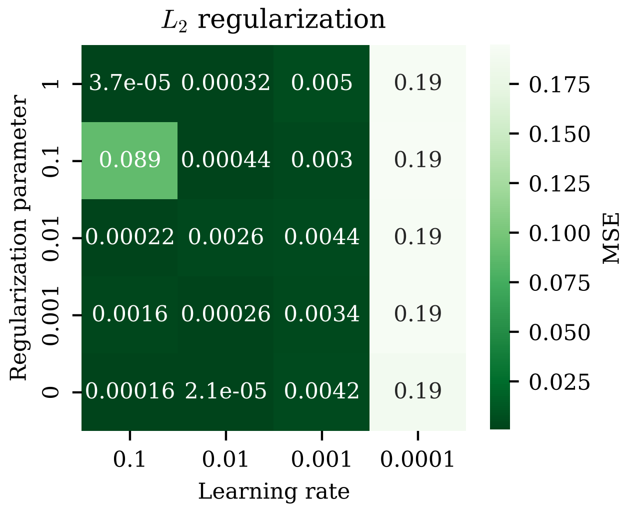

Figure 5: Results of a grid search where the activation functions and the number of hidden layers varied. The activation function in the hidden layers was SiLU.

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 $\lambda = 1$ are outliers. The choice $\lambda = 0.1$ gives significantly higher MSE compared to other values of λ 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

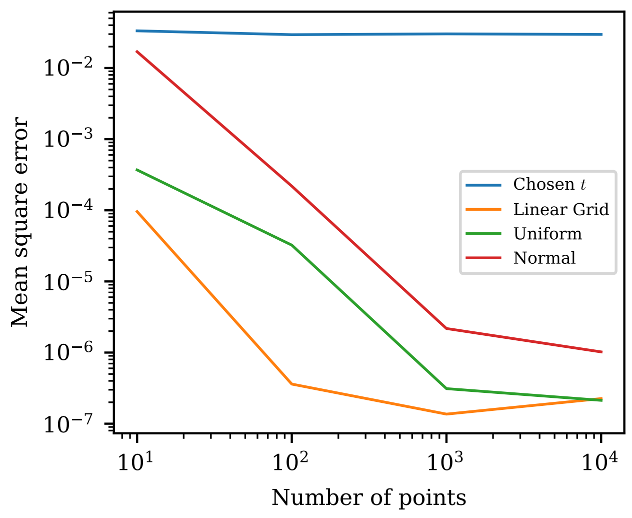

Figure 6: Mean squared error of neural networks with identical hyperparameters trained on differently generated points as a function of the number of points.

positive effect. Therefore we do not add regularization for the rest of the paper.

Since we are working with synthetic input data we are free to select points in any way. Therefore, we investigated the effect of training the neural network on differently generated points. For each way of generating points the MSE was computed for cases where the neural network was trained on approximately 10, 100, 1000, and 10,000 points. Since the methods did not always allow for an exact power of 10, we sometimes used an approximate amount. We looked at four different ways of generating data. Firstly, we simply generated (x, t) points along the lines $t = 0.05$ and $t = 0.30$ by linearly spacing the x values over $[0, 1]$. We call this "Chosen t" sampling. Secondly, we looked at a linearly spaced grid. This method did not always allow for a number of points exactly a power of 10. Instead we used $4 \times 3 = 12$, $10 \times 10 = 100$, $32 \times 32 = 1024$ and 100×100 points. Thirdly, we looked at uniformly distributed points, where x and t were independently sampled from $[0, 1)$ and $[0, 0.3)$. Lastly, we considered normally distributed points where x and t were independently sampled from a normal distribution where the mean was the midpoint of the intervals $[0, 1]$ and $[0, 0.3]$ for x and t respectively. The standard deviation was chosen to be the midpoint divided by 3, as this made it very likely that the points remain inside the intervals of interest. For all cases we used a neural network with SiLU as its activation function, 4 hidden layers with 60 nodes each and no regularization. Training was performed over 1000 epochs using Adam with learning rate 0.01. The MSE was computed by comparing the predictions to the analytical solution for 101 linearly spaced (x, t) points along $t = 0.05$ and $t = 0.3$ each. The results are shown in Figure 6.

From Figure 6 we see that the linear grid method performs best and that the "Chosen t" method performs worst. We would expect points used for training give the most information about the points close to it. Therefore, choosing only points with the t value we wish to find does presumably not give the neural network much data about the change in the time direction. Similarly, a linearly spaced grid would therefore under this view maximize the information. Hence, the superior performance of this method is not surprising. The uniformly distributed points perform slightly worse. While the points are spread out over the region the random nature risks having some points falling too close and some falling too far away. The normally distributed points perform worse again. While the points are more spread out than in the "Chosen t" method, they are still clustered around the center of the region $[0, 1] \times [0, 0.3]$.

While this behavior clearly indicates that the linear grid performs best among these methods in the case of the heat equation, this might not be the case for other partial differential equations. In particular, the case of equations with very unstable regions, is interesting. Here one might expect that the optimal point sampling is to sample sparsely from the stable regions and more densely from the regions known to be unstable.

Combining what we have found to be optimal hyperparameters and sampling methods, we computed the MSE as a function of the number of iterations for a different learning rates. We used neural networks with 4 hidden layers, each of 60 nodes, SiLU activation functions and no regularization. The training was performed on a linear grid of 100 points using Adam. The resulting MSE curves can be seen in Figure 7. Here we see that while the neural network solutions gets low mean squared error values, only the lowest spike here performs as well as the explicit scheme, which in the worst case had an MSE on the order of 10^{-7}. While the neural network does perform worse, it is more flexible in that it does not need to redo the training to predict solutions for new points. For the explicit scheme the calculation needs to be rerun to predict new points.

IV. CONCLUSION

We find that even the best neural network model in our limited testing performs just on par with the explicit scheme in the low resolution case ($\Delta x = 1/10, \Delta t = 1/100$), where they both achieve MSE values on the order 10^{-7}. In the high resolution case ($\Delta x = 1/100, \Delta t = 10^{-5}$) the explicit scheme outperforms the neural network by achieving an MSE on the order of 10^{-11}. While the neural network gets a higher MSE, it is more flexible and permits predicting solutions at new points at little extra cost. On the other hand, the initial computational cost of training is high. We note that our results are limited by the low sample size, increasing the effect of the randomly initialized model parameters.

Notably, we found that regularization did not seem to

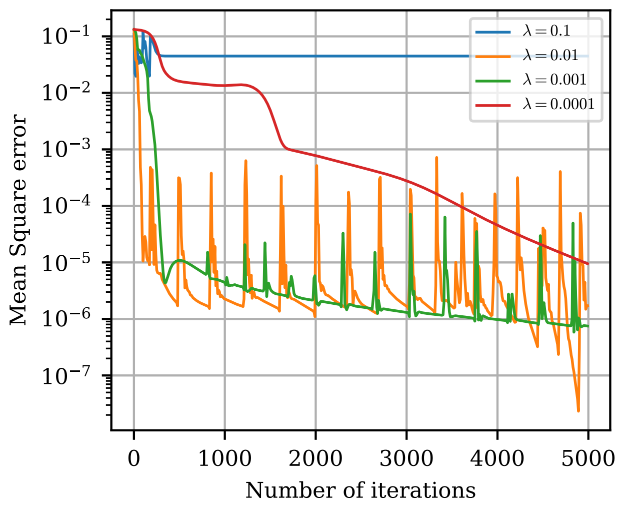

Figure 7: Mean square error as a function of the number of iterations for various learning rates. We used hyperparameters found to perform the best in our grid searches.

have a consistent positive effect. We hypothesize that this is partly due to the nature of the PDE problem, where the input coordinates are uncorrelated and noise free.

We also investigated the effect of different methods of sampling points for training on the MSE. We found that the naive method of sampling the points as a linear grid performed the best. However, we theorize that this does not apply for more unstable differential equations, where more unstable regions could benefit from denser sampling.

For future work we suggest revisiting the grid search with more computational resources to increase the number of runs, reducing the impact of randomness and thereby the accuracy of the results. Studying the behavior of different sampling methods in the case of more unstable differential equations would also be of interest. Lastly, we suggest a more thorough treatment of regularization in the case of neural networks used to solve PDEs.

[1] I. Lagaris, A. Likas, and D. Fotiadis, IEEE Transactions on Neural Networks **9**, 987–1000 (1998), ISSN 1045-9227, URL `http://dx.doi.org/10.1109/72.712178`.

[2] M. Hjorth-Jensen, *Applied Data Analysis and Machine Learning* (Department of Physics, 2025), URL `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`.

[3] M. Hjorth-Jensen, *Computational Physics Lecture Notes 2015* (Department of Physics, University of Oslo, Norway, 2015), URL `https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf`.

[4] A. Kvellestad, *Lecture Notes FYS3150*, `https://github.com/anderkve/FYS3150/blob/master/lecture_notes/2025/lecture_notes.pdf`, accessed on October 22, 2025, URL `https://github.com/anderkve/FYS3150/blob/master/lecture_notes/2025/lecture_notes.pdf`.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al., *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015), software available from tensorflow.org, URL `https://www.tensorflow.org/`.

[6] F. Chollet et al., *Keras*, `https://keras.io` (2015).

[7] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization* (2017), 1412.6980, URL `https://arxiv.org/abs/1412.6980`.