Project 3 FYS-STK-4155

Ingebrigt Kjæreng

18.12.2023

Abstract

I explored the use of a Standard Explicit Scheme Algorithm, a Recurrent Neural Network and a Convolutional Neural Network to predict temperatures along a grid given by a Partial Differential Equation. I also found the analytical solution of The Differential Equation. To make the PDE solvable by a Convolutional Neural Network, it was transformed to a time- and spatial-discretized form that was interpreted as patches of an image. This transformation was performed both using the Euler's Forward Method and the Implicit Euler's Method, comparing the performance of the two. I used the MSE cost function, and a range of Activation Functions for the Recurrent Neural Network. A common problem with Machine Learning methods is that they are incurring a high computational cost and for instance exploding gradients. To resolve this, Weight-Sharing, along with Dropout and Gradient Clipping was implemented, in addition to momentum and l2-regularization. For the Recurrent Neural Network, I used a technique called Backpropagation in Time to update the weights. I measured the accuracy of the predicted values vs. the real values by the Root Mean Squared Error (RMSE) for the Recurrent Neural Network and the Loss for the Convolutional Neural Network. The Recurrent Neural Network outperformed the Standard Explicit Scheme, and it significantly outperformed the best parameter combination of the Convolutional Neural Network by 5 decimal points, yielding an RMSE=0.00009 for the best parameters. However the Recurrent Neural Network was significantly more computationally expensive than the Standard Explicit Scheme, while the Implicit Euler's Method outperformed Euler's Forward Method by a factor of 10^-4 for the Recurrent Neural Network. The parameter combination yielding the best result was also cross-validated to verify the result. The results showed the difference in ability, strength and flexibility across the 3 models and the importance of choosing the right model for solving specific problems. It also showed how choosing the optimal parameters for the Neural Networks are essential to getting the most accurate results, and a big part of the analysis was raising the question whether the Neural Networks are a good choice with a certain computational cost.

1 Introduction

Machine Learning methods is a huge, ongoing field of research, and is used to solve many different types of problems. The Neural Networks have proven to be more accurate in many cases than easier solutions. The Neural Networks learn information from data by passing information to smaller units, neurons, and updating internal parameters based on previous performances, and the Neural Networks are statistical models based on iteratively learning patterns from fed-in data sets by passing information to neurons and updating parameters like weights and biases associated with each neuron based on previous performance. Two specific types of Neural Networks are the Convolutional Neural Network and the Recurrent Neural Network. The Recurrent Neural Network is often a somewhat more computationally expensive variant of a Neural Network. A differential equation was solved for the temperature gradient of a rod, predicting a critical temperature. This was solved using both the Recurrent Neural Network and the Convolutional Neural Network, where the temperature prediction was evaluated using MSE performance metric. The motivation for using both the Recurrent and Convolutional Neural Network stemmed from a desire to know whether the Convolutional Neural Network, which recures more modification than an explicit solution, or the assumed computationally more expensive Recurrent Neural Network was necessary in this case with possible complex temporal dependencies or high-dimensional data. Both networks were also implemented using both the Forward Euler's Method and Implicit Euler's Method to derive slightly different expressions interpretable to the Neural Networks, and seeing if this difference would make a difference to the predicted Loss/RMSE.

In the next section, I lay out the differential equation, the basic assumptions and elements of the recurrent and convolutional neural networks, the way the diferential equation was spatially and time-discretized, the cost function and activation functions used, along with weight-sharing and gradient clipping, techniques used to mitigate the cost of using the Neural Networks. Section 3 begins laying out the results from solving the Differential Equation using the Standard Explicit Scheme, then it is solved by Recurrent Neural Networks with different configurations, showing a certain combination of parameters yields a great result and finally the Convolutional Neural Network, attaining somewhat sub-optimal results for the computational costs incurred. Then Section 4 analyzes the results and compares them, discussing the computational costs incurred and whether they are necessary.

2 Mathematical Models

2.1 The Differential Equation

The Differential Equation was solved by using first a standard explicit scheme, and then using the Recurrent Neural Network and the Convolutional Neural Network. When solving the equation using the standard explicit scheme, there was also done tests of solutions for $\delta(x) = 1/10$, $\delta(x) = 1/100$ using $\delta(t)/\delta(x)^2 leq 1/2$, monitoring the solution over the two time points $t1$ and $t2$ with a smooth $u(x1, t)$ and a linear $u(x2, t)$. The differential equation solved is the physical problem of the temperature gradient in a rod of length L = 1 at x=0 and x=1. This is a one-dimensional problem

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

$$u_{xx} = u_t \tag{2}$$

$$u(x, 0) = sin(\pi * x) 0 < x < L \tag{3}$$

. The boundary conditions are

$$u(0, t) = 0, t_{>}0$$

,

$$u(L, t) = 0, t_{>}0$$

. The function $u(x, t)$ is the temperature gradient of a rod. As time increases, the velocity approaches a linear variation with x. The explicit forward Euler algorithm was implemented with discretized versions of time given by a forward formula and a centered difference in space resulting in

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} = \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t} \tag{4}$$

$$u_{xx} \approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2} \tag{5}$$

$$u_{xx} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2} \tag{6}$$

The differential equation was first solved using Euler's Forward Method for the standard explicit scheme, and then it was solved using a Recurrent Neural Network and a Convolutional Neural Network. The mathematics explained here lays out the methods used for the Forward Euler's Equation. Then the same was done for the Implicit Euler's Method. The mathematics for laying out interpretable forms using the Implicit Euler's Method are described in the Appendix. Converting it to a form interpretable to the Neural Networks meant knowing the ansatz for the solution, the information about the initial conditions, and information about the boundary conditions. I call the aforementioned boundary conditions $g(x)$, and the initial condition $f(x = 0) = a$. With the boundary conditions, the neural network changed iteration by iteration. The cost function could be the difference between the left-hand side and the right-hand side squared, which looks like a MSE. An ansatz for the solution

was first set up, and then the information was fed in about the initial and boundary conditions, then the difference between the left hand side and the right hand side with my starting guess was calculated. The equation was discretized, so $f(x)$ moved over to a discretized equation, $x(y) = f(y)$. x is discretized, and the discretization means I have $x, x0- > x1$, the final value, so i have $n+1$ points. $f(x=0) = f(x0) = f0$, $xi = x0 + i\Delta * x$, $\Delta * x = (xn - x0)/n$. The no. of integration or discretization points are defined as

$$f(x + \Delta * x) = f(x) + \Delta * x - f'(x) + (\Delta * x^2/2!)f'' + 0(\Delta * x)^3) \tag{7}$$

. Euler's forward formula is given as

$$f'(x) = f'(i) = f(xi + \Delta * x) - f(xi)/\Delta * x = fi + i - fi/\Delta * x \tag{8}$$

,

$$f''i = fi + 1 + fi - 1 - 2fi/(\Delta * x)^2 \tag{9}$$

, by manipulating 2 Taylor-expressions, $(x + \Delta(x))$ and $(x - \Delta(x))$,

$$f'i = fi + 1 - fi - 1/2\Delta(x) * (0(\Delta(x)^2)), df/dx = g(x), fi + 1 - fi/\Delta(x) = gi => fi + \Delta(x) \tag{10}$$

,

$$d^2f/dx^2 = -\alpha(x), f(x) = A * cos(\alpha(x)) + B * sin(\alpha(x)) \tag{11}$$

, and the Implicit Euler's Formula applied to the above equation:

$$\Delta t \frac{u(x, t_{j+1}) - u(x, t_j)}{\Delta t} = \frac{\partial^2 u(x, t_{j+1})}{\partial x^2} \tag{12}$$

Finding the solution was done when saying that f, at $(x = 0) = f(x0) = f0 = 0 => A = 0$. Initial conditions were implemented in the Neural Network, and it began by guessing $h(x) = h0$, where $h0$ obeyed the initial conditions.

2.2 The Analytical Solution

The differential equation has the following analytical solution using the Forward Euler's Method, also found in Python at ...., where the complete Mathematical Solutions with all steps can be found in the Appendix. Using the Forward Euler's Method, the Fourier series solution for the given problem is:

$$u(x,t) = \sum_{n=1}^{\infty} \frac{1}{n\pi} \cdot \frac{1}{2} \sin(n\pi x) \cos(n\pi t) \tag{13}$$

Using the Implicit Euler's Method: the Fourier series solution

$$u(x,t) = \sum_{n=1}^{\infty} \frac{n\pi}{1} \sin(n\pi x) \cos(n\pi t) \tag{14}$$

, slightly different than the one found with the Forward Euler's Method.

2.3 The Recurrent Neural Network

The differential equation was first solved using a Recurrent Neural Network. The number of features are given by the columns of the design matrix, the number of features connecting to the hidden nodes in what is called the hidden layer, which is not a layer in the sense of the FFNN, but really capturing information about the sequence's context at each time step, the evolving hidden states over time, where each time step corresponds to a different instance in time. The hidden layer at each time step contains information about the sequence up to that point. The setup of a recurrent neural network in general involves a function receiving input from a previous function:

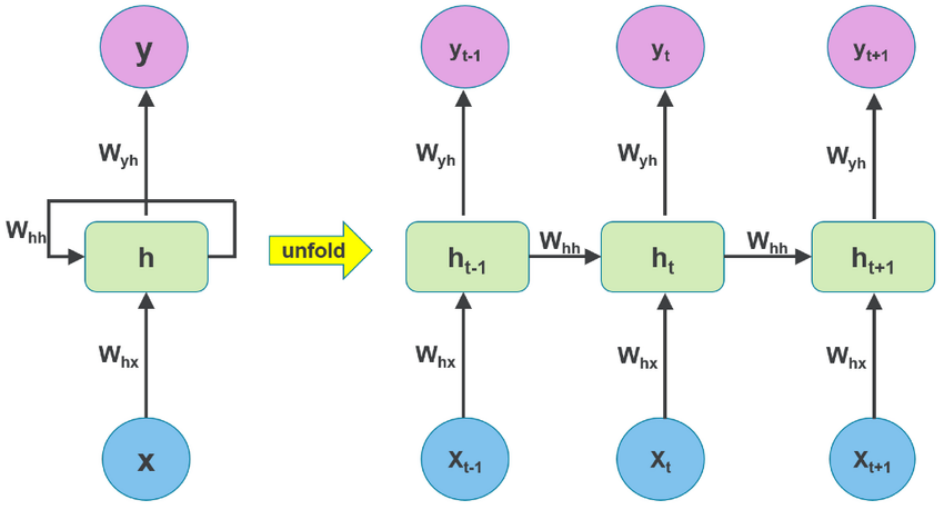

Figure 1: **A Recurrent Neural Network**

[1]

,where hi are the hidden layers, wi are the weights applied to each layer, and xi are the inputs, yi are the outputs, all at a given time step t. The hidden state at a given time step is computed using the current input and the previous hidden state, expressed as:

$$ht = f(W[ht-1, xt]) \tag{15}$$

, where f is the activation function. Important here is that the order in which the computations are performed corresponds to the order of the layers. If the input is a sequence, as is the case here, every input determines the internal state of the recurrent neural network for the following inputs. I also deployed the Back-Propagation through Time, leading to a sequential algorithm, illustrated below:

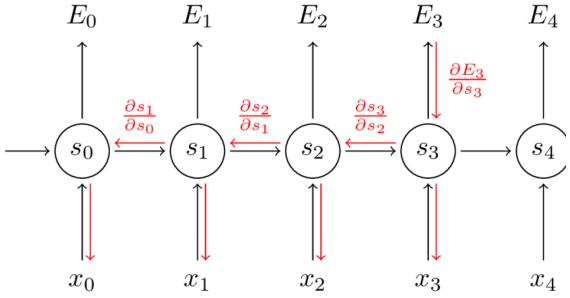

Figure 2: **Back-Propagation Through Time**

[2]

, where the key difference between this and what is the case for the backpropagation in a regular Neural Network is the summing up of the gradients for W at each time step. The Network was set up from scratch and inspired by the notes in [1].

2.4 The Convolutional Neural Network

The Differential Equation was then solved using the Convolutional Neural Network. The Convolutional Neural Network uses convolutional layers to capture spatial hierarchies and patterns in input data. The CNN consists of the input layer, containing the Raw data input, the convolutional layers applying filters to the input data to extract local patterns and features, where each filter scans across the input data, performing element-wise multiplications and aggregating the results to produce a feature map, then multiple features are used to capture different features in parallel. A convolution layer defines a window by which we examine a subset of the image, and subsequently scans the entire image looking

through this window. As you'll see below, I can parameterize the window to look for specific features (e.g. edges) within an image. Layers use parametrized filters to transform the input into useful representations. This window is also sometimes called a filter, since it produces an output image which focuses solely on the regions of the image which exhibited the feature it was searching for. The output of a convolution is the feature map. When going into deeper layers of the network, the channel depth will usually increase as feature maps become more specialized one needs more feature maps to represent the input.

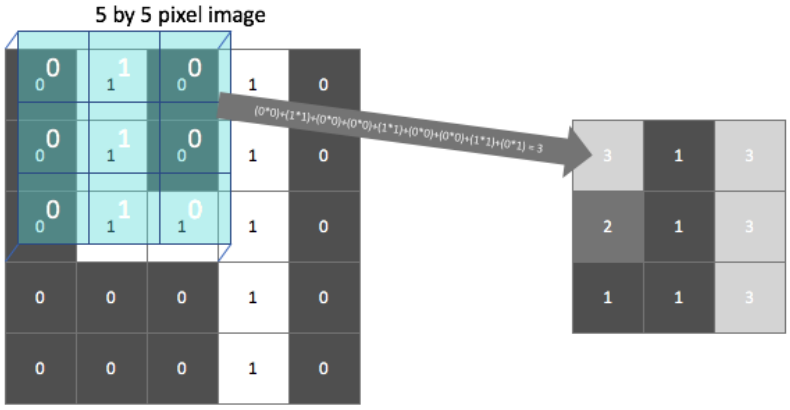

Figure 3: **The Convolutional Neural Network Layers**

[3]

This filter was overlayed onto the input, and linearly combined, then the pixel and filter values were activated. The output, shown on the right, identifies regions of the image in which a vertical line was present. A similar process could be done using a filter designed to find horizontal edges to properly characterize all of the features of a "4". Then it scanned each filter across the image calculating the linear combination (and subsequent activation) at each step. The network further consists of the activation function and pooling layers. A pooling layer compresses spatial information rather than extracting certain features, with the most important goal of preventing overfitting and reducing computational complexity.

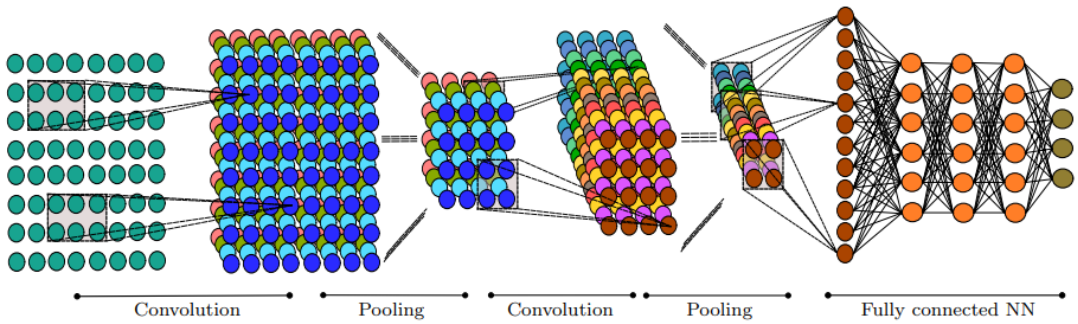

Figure 4: **The Convolutional Neural Network, Pooling**

[4]

The pooling takes on one of the different techniques Max pooling, average pooling, minimum pooling or global pooling, w. max pooling the most common. I applied the Max pooling here, which takes place on each feature map independently. When $A(i, j)$ represents the element at the i-th row

and j-th column of the input matrix, and B is the resulting matrix, the operation is given as

$$B(i,j) = max^p = 0k - 1max^q = 0k - 1A(i * S + p, j * S + q) \tag{16}$$

, where S is the stride, and the double max operation is used to find the maximum value within each pooling window. The network then proceeds to flattening, fully connected layers processing the flattened vector, similar to the hidden layers in a Feed Forward Neural Network. Then it has the output layer, the loss function and optimization and training. The convolutional Neural Network looks at local regions of the input. For instance, a 2 by 2 filter with a stride of 2 is scanned across the input to output 4 nodes, each containing localized information about the image. The 4 nodes will then be combined to form the "pixels" of a feature map, where the feature extracted is dependent on the parameters of the filter:

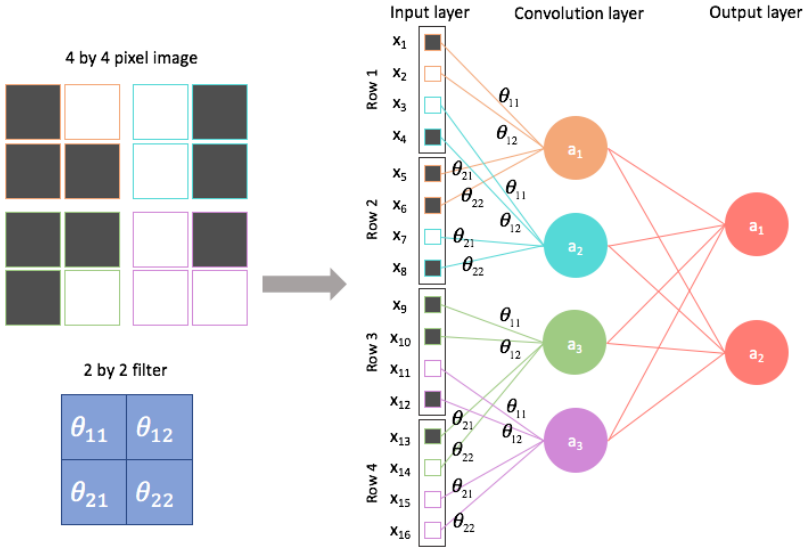

Figure 5: **The Convolutional Neural Network**

[5]

, where θ contains all the parameters in the network, including the weights and biases. The Convolutional Neural Network was built from scratch, with inspiration taken from [2].

2.5 Cost Function

When solving the differential equation, the MSE cost function was deployed because of it penalizing the difference between predicted and true values, and because the problem is framed as a regression problem, where the Neural Network is trained to predict the next state or derivative given the current state. The function then measures the difference between the predicted and actual values.

2.6 Solving the Differential equation with the Recurrent Neural Network

The motivation of using a Recurrent Neural Network to solve the problem was a desire to know whether the temporal evolution of temperature over time was essential, or if it had complex temporal dependencies. Solving the equation means knowing the ansatz for the solution, the information about the initial conditions, and information about the boundary conditions. I call the aforementioned boundary conditions $g(x)$, and the initial condition $f(x = 0) = a$. With the boundary conditions, the neural network changed iteration by iteration. The cost function could be the difference between the left-hand side and the right-hand side squared, which looks like a MSE. An ansatz for the solution was first set up, and then the information was fed in about the initial and boundary conditions, then the difference between the left hand side and the right hand side with my starting guess was calculated. The equation was discretized, so $f(x)$ moved over to a discretized equation, $x(y) = f(y)$. x is discretized, and the discretization means I have $x, x0 - > x1$, the final value, so i have $n + 1$ points.

$f(x=0) = f(x0) = f0$, $xi = x0 + i\Delta * x$, $\Delta * x = (xn - x0)/n$. The no. of integration or discretization points are defined as

$$f(x + \Delta * x) = f(x) + \Delta * x - f'(x) + (\Delta * x^2/2!)f'' + 0(\Delta * x)^3) \tag{17}$$

. Finding the solution was done when saying that f, at $(x = 0) = f(x0) = f0 = 0 => A = 0$. The ansatz for the solution was:

$$u(x,t) = f(t) \cdot g(x) \tag{18}$$

, where $f(t)$ captures the temporal behavior, and $g(x)$ captures the spatial behavior. In addition to implementing the initial conditions in the Neural Network, which began by guessing $h(x) = h0$, where $h0$ obeyed the initial conditions, the spatial and temporal discretization transformed equations w.r.t. the time and spatial relationships using the Forward Euler's Formula ready to use in the Neural Network was substituted back in the differential equation:

$$\Delta t\alpha \left(u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)\right) \approx \frac{\Delta x^2}{2} \left(u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)\right) \tag{19}$$

, and this is the form in which it was implemented.

2.7 Solving the Differential Equation with the Convolutional Neural Network

The Differential Equation solved by the Convolutional Neural Network has ansatz

$$u(x, t + \Delta t) - u(x,t) \approx \alpha \cdot (u(x + \Delta x, t) - 2u(x,t) + u(x - \Delta x, t)) \tag{20}$$

, and converting it to an interpretable form for the CNN means in the end we have

$$\approx \frac{\Delta x^2}{2} \left(u(x + \Delta x, t) - 2u(x,t) + u(x - \Delta x, t)\right) \tag{21}$$

, using the Forward Euler's Method.

2.8 Activation Functions

Based on the performance and comparisons of [4], I chose a number of activation function deemed appropriate for analysis and implemented the activation functions Sine, ReLU, Sigmoid, TanH and Swish in the Recurrent Neural Network. I had initial values for the weights and biases, that produced an initial weight matrix, and these were weighted by the output of the layer $ht - 1$. Then this was set up to an output, but this was modulated by a new weight matrix, that produces a $y, y1$. The ht, which the activation function is deployed upon, produced a quantity, $R(t)$, which will be given in the new matrix $V(h(t)) + C$, where C is the Bias. The parameters needed to be trained were the weights, 3 matrices, which link everything, and then, the bias is b and c. Because doing this at every step required a lot of memory, I used weight-sharing, and to avoid the problem of the exploding gradient, I deployed the Gradient Clipping technique.

2.9 Weight-Sharing

Weight-Sharing was applied to reduce the computational cost. It uses weights and biases for multiple parts of the network. Instead of learning distinct patterns for each connection or neuron, weight-sharing allows the network to share the same set of parameters across different regions. Applied in a network layer, it looked like this:

$$u_{\text{new}}(i,j) = \theta \left(\sum_{p=0}^{k-1} \sum_{q=0}^{k-1} w(p,q) u(i+p, j+q) + b \right) \tag{22}$$

where $u_{\text{new}}(i,j)$ is the new temperature value at spatial location (i,j), $w(p,q)$ are shared weights, $u(i+p, j+q)$ are input values from the previous layer, b is the bias term, and θ is the activation function. The input to the Convolutional Neural Network is the aforementioned interpretable variant of the differential equation, structured as a grid. Convolutional layers in the network then capture local patterns and relationships. Weight-sharing is applied by using the same set of filters across different spatial locations. Each filter learns to detect specific spatial patterns or features. Weight-sharing is applied across different layers of the network. It learns the spatiotemporal patterns in the evolving temperature distribution and applies the same set of filters across different spatial locations, learning to recognize spatial features relevant to the evolution of temperature. The network stacks multiple layers to capture the evolution of the temperature over time and shares weights across these layers to enable the network to generalize the temporal dynamics.

2.10 Gradient Clipping

To avoid gradients becoming too large during backpropagation, gradient clipping was deployed, setting a threshold, and if the gradients exceed this threshold, they were scaled down to keep them within a reasonable range. Here I set a global norm threshold. The exploding gradients arise during the calculation of the gradient with respect to the model parameters in the backpropagation through time algorithm through the update equation

$$ht = activation(Whh * ht - 1 + Wxh * xt + bh) \tag{23}$$

, or $ht = activation(temporalterm + spatialterm + b0)$, where ht is the hidden state of the Recurrent Neural Network at time t, analogous to the temperature field $u(xi, tj)$ in the heat equation, Whh is the recurrent weight matrix, capturing the temporal evolution and spatial information of the hidden state, Wxh is the weight matrix for the input xt, which represents the spatial information, bh is the bias term. When the spectral radius of Whh is greater than 1, and the value is maintained or increased over multiple time steps, the gradient w.r.t. ht can become very large. The gradient clipping takes the form of

$$grads_clipped = clip_value * (grads/max(1, ||grads||/clip_value) \tag{24}$$

, where $||grads||$ is the L2 norm of the gradient vector, and the division ensures the scaling factor is applied only when the norm exceeds the threshold, preventing zero division. In addition, dropout was applied to prevent overfitting.

3 Results

3.1 Explicit Scheme Algorithm

The explicit scheme algorithm consisted of predicting the temperature gradient for $x \in (0,1)$ at $t_1 = 0$ and $t_2 = 0.02$ using Δt as dictated by the stability limit of the explicit scheme, where the stability criterion for the explicit scheme requires that $\Delta(t)/\Delta(x)\ 2 \leq 1/2$. Tests were performed for $\Delta(x) = 1/10$ and $\Delta(x) = 1/100$:

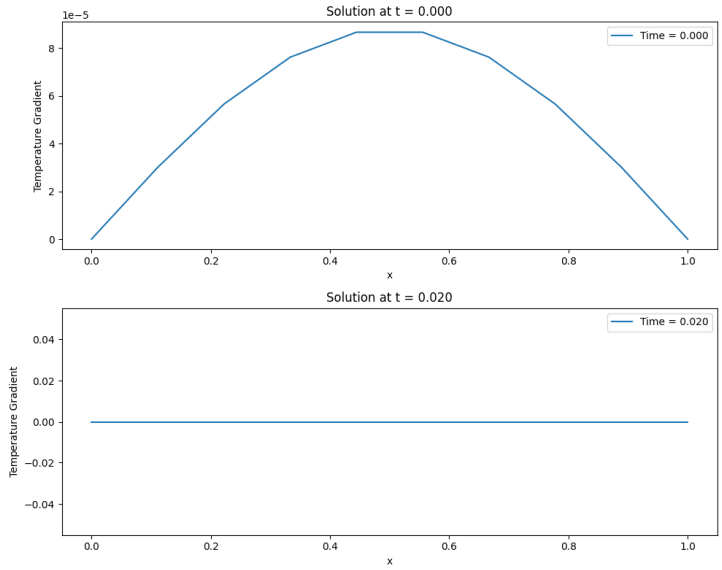

Figure 6: **The Explicit Scheme Algorithm, for** $\Delta(x) = 1/10$
6

At $t = 0$, the curve is not very smooth.

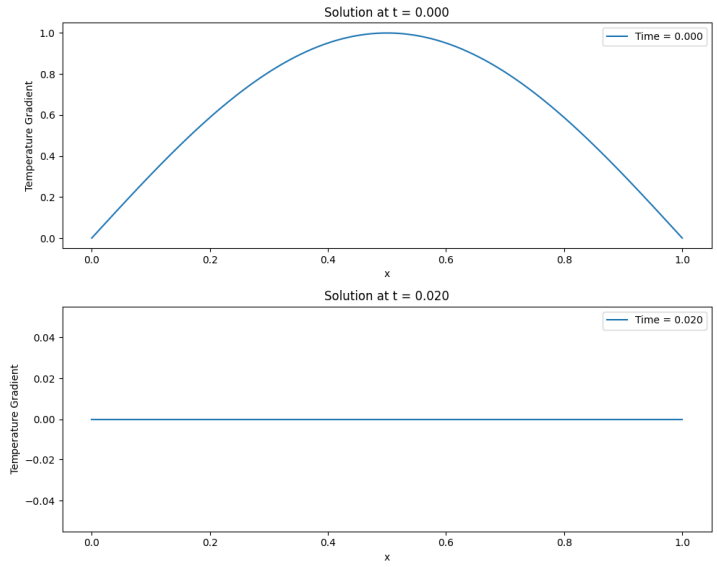

Figure 7: **The Explicit Scheme Algorithm, for** $\Delta(x) = 1/100$
7

Expectedly, with a smaller $\Delta(x)$, the curve became smoother. When deploying the Implicit Euler's Method, both sizes of $\Delta(x)$ yielded the same smooth curve as in 7, meaning it did not show the lack of smooth convergence at a larger $\Delta(x)$. The only difference in the Euler's Forward Method and Implicit Euler's Method implementation was the fact that the Implicit Euler's Method constructs a matrix system and solves it at each time step, where as the Forward method iterates over time and spatial steps using the forward difference scheme. The full solution of the explicit scheme using Euler's Forward Method is shown on the left in 8, where using the Implicit Euler's Method is shown on the right:

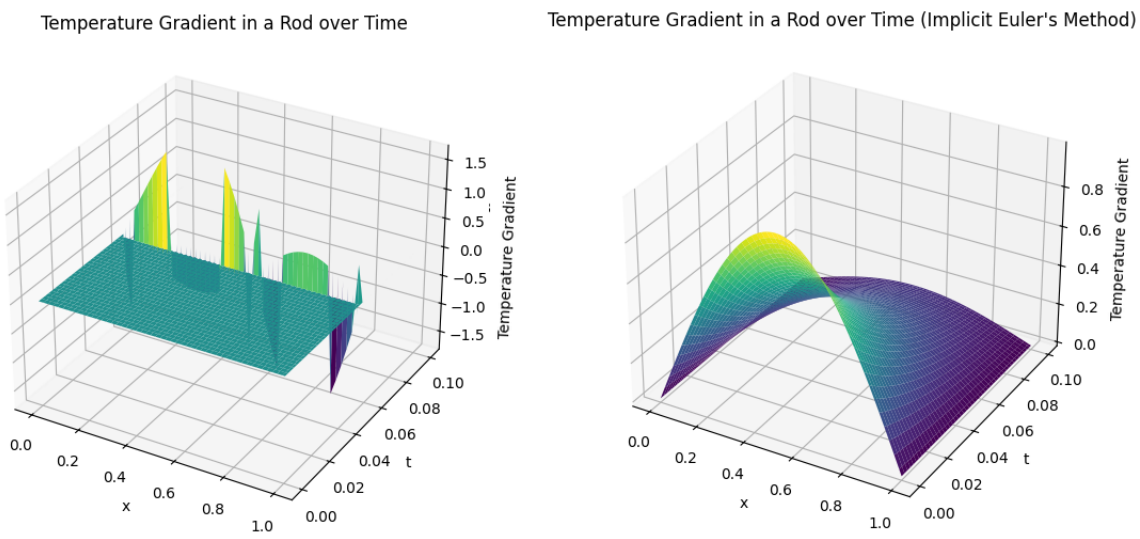

Figure 8: **The Explicit Scheme Algorithm using Euler's Forward Method, left, Implicit Euler's Method, right**
8

3.2 Recurrent Neural Network

I wanted to see whether the difference between the Forward Euler's Method and the Implicit Euler's Method was also present after implementing the Neural Networks. I began by implementing the Recurrent Neural Network using the Forward Euler's Method, using the 5 different activation functions Sigmoid, Sine, ReLU, TanH and Swish, training the left hand side and the right hand side of the PDE separately, beginning with the Sigmoid activation function. I then modified the no. of hidden layers, neurons, momentum and learning rate and gradually tested the other four activation functions. I then ran 5-fold cross-validation over the combination giving the best results, yielding an RMSE of 0.003456. Then I did the same, using the interpretable form of the equation with the Implicit Euler's Method. I made a prediction of the test set of the temperature for the selected RNN, comparing the predicted and true temperature, figure 11. I then created a table of all 5 activation functions combined w. different layer sizes, learning rates and momentum, comparing RMSE over the 120 combinations. Using MCMC sampling rather than Batches greatly sped up the computation time.

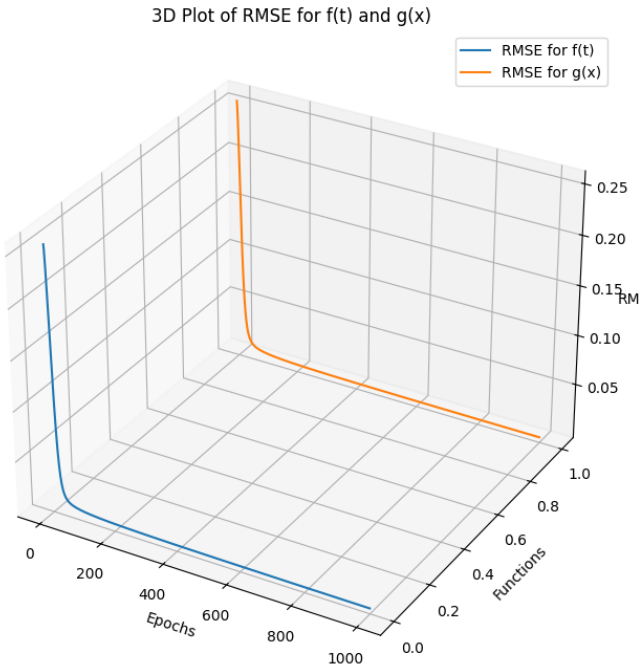

Figure 9: **The best parameters of the Recurrent Neural Network using the ReLU activation function, momentum=0.7, learning rate=10^-2, 8 hidden layers, 2 neurons per layer, cross-validated** 9

Using the Implicit Euler's Method rather than Euler's Forward Method for the same combination, yielded an improvement in the RMSE of 10^-4:

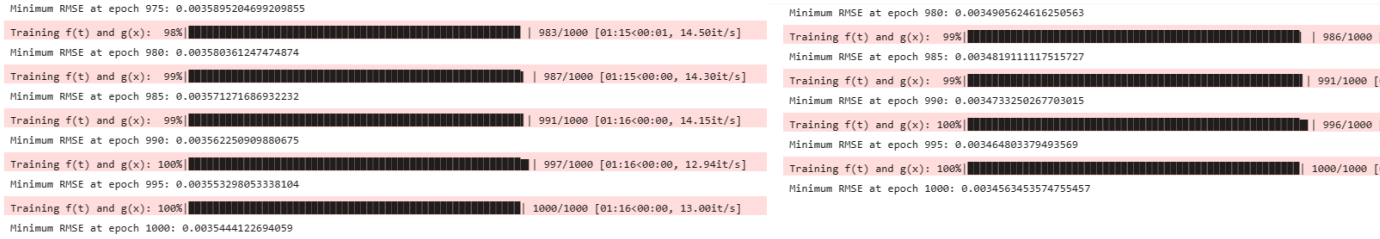

Figure 10: **Euler's Forward Method, left, and Implicit Euler's Method, right** 10

A prediction of the test set of the temperature for the selected RNN, comparing the predicted and true temperature, using Spins for solving the PDE, which the RNN relies on:

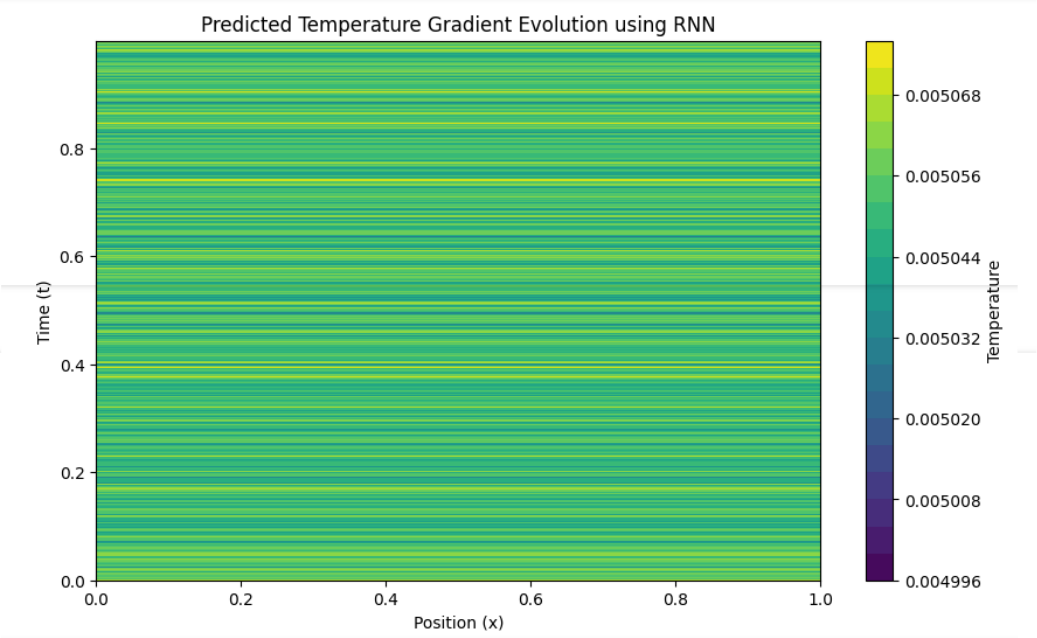

Figure 11: **Following the development of the predicted RMSE(Temperature Gradient Evolution of the Rod) over time using the RNN based on Spins for solving the PDE.**
11

After running a comparison of 120 combinations of activation functions, momentum coefficients, hidden layer sizes and learning rates, applying both dropout, weight-sharing and gradient clipping the results were presented in a table:

	Hidden Layers	Neurons per Layer	Activation Function	Momentum		Learning Rate	Dropout Rate	Min Training Loss
0	1	[3]	sigmoid	0.7	0	1.000000e-06	1	0.000009
1	1	[3]	sigmoid	0.7	1	1.000000e-07	1	0.000009
2	1	[3]	sigmoid	1.0	2	1.000000e-06	1	0.000009
3	1	[3]	sigmoid	1.0	3	1.000000e-07	1	0.000009
4	1	[3]	sine	0.7	4	1.000000e-06	1	0.000009
..	...	...	...	...	..	...	...	...
115	2	[25, 25]	tanh	1.0	115	1.000000e-07	1	0.000009
116	2	[25, 25]	swish	0.7	116	1.000000e-06	1	0.000009
117	2	[25, 25]	swish	0.7	117	1.000000e-07	1	0.000009
118	2	[25, 25]	swish	1.0	118	1.000000e-06	1	0.000009
119	2	[25, 25]	swish	1.0	119	1.000000e-07	1	0.000009

Figure 12: **Table of combinations of parameters and RMSE.**
12

I then added l2-regularization to the Recurrent Neural Network, first and foremost to the parameter combination currently yielding the best result, resulting in the following for both very small and large values of l2, where a significant enough size on the l2-regularization parameter proved vital to generate a good RMSE:

```
Training f(t) and g(x) with L2 regularization (l2_penalty=1): 100%|████████████| 998/1000 [01:08<00:00, 14.91it/s]
Minimum RMSE at epoch 995: 0.008851101289794886
Training f(t) and g(x) with L2 regularization (l2_penalty=0.1): 100%|████████████| 1000/1000 [01:07<00:00, 14.88it/s]
Minimum RMSE at epoch 1000: 0.2770095427503684
Training f(t) and g(x) with L2 regularization (l2_penalty=0.01): 100%|████████████| 1000/1000 [01:05<00:00, 15.33it/s]
Minimum RMSE at epoch 1000: 0.1843866597557269
Training f(t) and g(x) with L2 regularization (l2_penalty=0.001): 100%|████████████| 1000/1000 [01:04<00:00, 15.45it/s]
Minimum RMSE at epoch 1000: 0.7319150307340814
Training f(t) and g(x) with L2 regularization (l2_penalty=0.0001): 100%|████████████| 1000/1000 [01:15<00:00, 13.25it/s]
Minimum RMSE at epoch 1000: 34.993061772809845
Training f(t) and g(x) with L2 regularization (l2_penalty=1e-05): 100%|████████████| 1000/1000 [01:14<00:00, 13.42it/s]
Minimum RMSE at epoch 1000: 2.5170558300427254e+18
```

Figure 13: **Table of different l2-terms and their corresponding RMSE.**

13

3.3 Convolutional Neural Network

I began by creaing a simple convolutional neural network with 1 convolutional layer and 1 dense layer, applying max pooling. The convolutional neural network takes the PDE in a discretized form, learned by the network as patches of an image input. Comparison of Loss w. different max pooling sizes:

1 Max pooling layer, Pool size = 2:

```
Epoch 1/25, Loss: 0.48729598241172123
Epoch 2/25, Loss: 0.5182457411377474
Epoch 3/25, Loss: 0.5509811652657421
Epoch 4/25, Loss: 0.5853691010152491
Epoch 5/25, Loss: 0.6212839100601316
Epoch 6/25, Loss: 0.6586068559128915
Epoch 7/25, Loss: 0.6972255515362321
Epoch 8/25, Loss: 0.7370334620912152
Epoch 9/25, Loss: 0.777929457317894
Epoch 10/25, Loss: 0.8198174085766332
Epoch 11/25, Loss: 0.862605826059851
Epoch 12/25, Loss: 0.9062075321189945
Epoch 13/25, Loss: 0.950539367047202
Epoch 14/25, Loss: 0.9955219240142179
Epoch 15/25, Loss: 1.0410793101737412
Epoch 16/25, Loss: 1.087138931255346
Epoch 17/25, Loss: 1.133631297217651
Epoch 18/25, Loss: 1.1804898467787222
Epoch 19/25, Loss: 1.2276507888551287
Epoch 20/25, Loss: 1.2750529591373092
Epoch 21/25, Loss: 1.3226376902054169
Epoch 22/25, Loss: 1.3703486937490608
Epoch 23/25, Loss: 1.4181319535988193
Epoch 24/25, Loss: 1.4659356284080354
Epoch 25/25, Loss: 1.5137099629404533
```

1 Max pooling layer, Pool size = 5:

```
Epoch 1/25, Loss: 0.749511290228364
Epoch 2/25, Loss: 0.7652322179043924
Epoch 3/25, Loss: 0.7807237224738086
Epoch 4/25, Loss: 0.79593361646902
Epoch 5/25, Loss: 0.8108162241539687
Epoch 6/25, Loss: 0.8253318487444063
Epoch 7/25, Loss: 0.8394462858655248
Epoch 8/25, Loss: 0.853130376964142
Epoch 9/25, Loss: 0.8663595976482916
Epoch 10/25, Loss: 0.8791136768961307
Epoch 11/25, Loss: 0.8913762438292213
Epoch 12/25, Loss: 0.9031344993298939
Epoch 13/25, Loss: 0.9143789102395556
Epoch 14/25, Loss: 0.9251029242324929
Epoch 15/25, Loss: 0.9353027037417223
Epoch 16/25, Loss: 0.9449768775359936
Epoch 17/25, Loss: 0.9541263087241204
Epoch 18/25, Loss: 0.9627538781048467
Epoch 19/25, Loss: 0.9708642818944061
Epoch 20/25, Loss: 0.9784638429577661
Epoch 21/25, Loss: 0.9855603347454017
Epoch 22/25, Loss: 0.9921628172016976
Epoch 23/25, Loss: 0.9982814839641161
Epoch 24/25, Loss: 1.0039275202192726
Epoch 25/25, Loss: 1.0091129706203046
```

1 Max pooling layer, Pool size = 1:

```
Epoch 1/25, Loss: 0.9296405022162872
Epoch 2/25, Loss: 0.9283072180730906
Epoch 3/25, Loss: 0.9269917233876318
Epoch 4/25, Loss: 0.9256923662786471
Epoch 5/25, Loss: 0.9244074752468087
Epoch 6/25, Loss: 0.9231353636812872
Epoch 7/25, Loss: 0.9218743341566021
Epoch 8/25, Loss: 0.9206226825235528
Epoch 9/25, Loss: 0.9193787017988466
Epoch 10/25, Loss: 0.918140685857332
Epoch 11/25, Loss: 0.9169069329316383
Epoch 12/25, Loss: 0.91567574892357
Epoch 13/25, Loss: 0.9144454505320273
Epoch 14/25, Loss: 0.9132143682021762
Epoch 15/25, Loss: 0.9119808489005875
Epoch 16/25, Loss: 0.9107432587211063
Epoch 17/25, Loss: 0.9094999853265869
Epoch 18/25, Loss: 0.9082494402310476
Epoch 19/25, Loss: 0.9069900609270625
Epoch 20/25, Loss: 0.9057203128636879
Epoch 21/25, Loss: 0.9044386912792378
Epoch 22/25, Loss: 0.9031437228940355
Epoch 23/25, Loss: 0.9018339674678587
Epoch 24/25, Loss: 0.9005080192267513
Epoch 25/25, Loss: 0.8991645081638854
```

Figure 14: **1 Max Pooling Layer, with different sizes**

14

I then increased the no. of pooling layers. Comparison of Loss w. 3 rather than 1 max pooling layer. Interestingly, the loss now converges, and indicates it could become better by running even more epochs:

3 Max Pooling layers, Pool Size = 1:

```
Epoch 1/25, Loss: 0.7881098811151983
Epoch 2/25, Loss: 0.7849625595604869
Epoch 3/25, Loss: 0.7817806418434708
Epoch 4/25, Loss: 0.7785646223835878
Epoch 5/25, Loss: 0.7753150121987002
Epoch 6/25, Loss: 0.7720323381394119
Epoch 7/25, Loss: 0.7687171421328246
Epoch 8/25, Loss: 0.7653699804359482
Epoch 9/25, Loss: 0.7619914228991962
Epoch 10/25, Loss: 0.7585820522408153
Epoch 11/25, Loss: 0.7551424633319055
Epoch 12/25, Loss: 0.7516732624931253
Epoch 13/25, Loss: 0.7481750668028826
Epoch 14/25, Loss: 0.7446485034174408
Epoch 15/25, Loss: 0.7410942089032821
Epoch 16/25, Loss: 0.7375128285818885
Epoch 17/25, Loss: 0.7339050158870907
Epoch 18/25, Loss: 0.730271431735287
Epoch 19/25, Loss: 0.7266127439085055
Epoch 20/25, Loss: 0.7229296264507289
Epoch 21/25, Loss: 0.7192227590772454
Epoch 22/25, Loss: 0.7154928265973283
Epoch 23/25, Loss: 0.7117405183502856
Epoch 24/25, Loss: 0.7079665276549563
Epoch 25/25, Loss: 0.7041715512725328
```

3 Max Pooling layers, Pool Size = 2:

```
Epoch 1/25, Loss: 0.600433786756424
Epoch 2/25, Loss: 0.5963356805645813
Epoch 3/25, Loss: 0.5922357284472378
Epoch 4/25, Loss: 0.5881345516859566
Epoch 5/25, Loss: 0.5840327647775346
Epoch 6/25, Loss: 0.5799309752660632
Epoch 7/25, Loss: 0.5758297835840581
Epoch 8/25, Loss: 0.5717297829026802
Epoch 9/25, Loss: 0.5676315589908633
Epoch 10/25, Loss: 0.5635356900828887
Epoch 11/25, Loss: 0.559442746754543
Epoch 12/25, Loss: 0.5553532918073663
Epoch 13/25, Loss: 0.5512678801610933
Epoch 14/25, Loss: 0.5471870587537181
Epoch 15/25, Loss: 0.5431113664493721
Epoch 16/25, Loss: 0.5390413339535266
Epoch 17/25, Loss: 0.5349774837354803
Epoch 18/25, Loss: 0.5309203299579555
Epoch 19/25, Loss: 0.5268703784134005
Epoch 20/25, Loss: 0.522828126467152
Epoch 21/25, Loss: 0.5187940630069393
Epoch 22/25, Loss: 0.5147686683988082
Epoch 23/25, Loss: 0.5107524144490162
Epoch 24/25, Loss: 0.5067457643720673
Epoch 25/25, Loss: 0.5027491727642819
```

3 Max Pooling layers, Pool Size = 5:

```
Epoch 1/25, Loss: 0.8978021012193265
Epoch 2/25, Loss: 0.8964195033428416
Epoch 3/25, Loss: 0.8950154584447212
Epoch 4/25, Loss: 0.8935887502387676
Epoch 5/25, Loss: 0.8921382029818687
Epoch 6/25, Loss: 0.8906626821145316
Epoch 7/25, Loss: 0.8891610948064375
Epoch 8/25, Loss: 0.887632390411286
Epoch 9/25, Loss: 0.8860755608348317
Epoch 10/25, Loss: 0.8844896408202211
Epoch 11/25, Loss: 0.8828737081543533
Epoch 12/25, Loss: 0.8812268837990604
Epoch 13/25, Loss: 0.8795483319507874
Epoch 14/25, Loss: 0.8778372600324444
Epoch 15/25, Loss: 0.8760929186208126
Epoch 16/25, Loss: 0.8743146013128821
Epoch 17/25, Loss: 0.8725016445346048
Epoch 18/25, Loss: 0.8706534272950033
Epoch 19/25, Loss: 0.8687693708890623
Epoch 20/25, Loss: 0.8668489385521063
Epoch 21/25, Loss: 0.8648916350688457
Epoch 22/25, Loss: 0.8628970063397242
Epoch 23/25, Loss: 0.8608646389076079
Epoch 24/25, Loss: 0.8587941594470726
Epoch 25/25, Loss: 0.8566852342192108
```

Figure 15: **3 Max Pooling Layers, with different sizes**

15

Adding l2-regularization showed that, like in the case of the Recurrent Neural Network, the size of the regularization is detrimental to getting a good Loss. Setting an l2-regularization term of 10^-2 yielded a good Loss, while all other values of $l2$ yielded a Loss that was, at minimum, 10^3 times larger. Moreover, implementing the Implicit Euler's Method for the discretization also gave a much worse Loss result than with the Forward Euler's Method, so much worse that it suggested an issue with the model. Decreasing the learning rate improved the loss, but it was still very bad, something that can likely be attributed to the fact that the model might also need other ways of normalizing the data.

4 Discussion

Using the Standard Explicit Scheme was a very low-cost operation and setting up the equations was not mathematically difficult nor computationally expensive. From 6 we see that the prediction of the teperature gradient for $\delta(x) = 1/10$ has a curve that is not very smooth. Predicting the Temperature Gradient using the Recurrent Neural Network resulted in a temperature that took on much lower values across the board and would in a figurative sense give a much smoother curve. However the aim is to strike a balance between what is a good model and what one is willing to spend when using that model. From the off it became apparent that the Recurrent Neural Network was more than capable of handling the PDE. The question then became whether it was computationally necessary to do so. With regard to the Recurrent Neural Network, I begin by noticing from 10 that the Implicit Euler's Method produced a better RMSE by a marginal factor, namely one of 10^-4. This is a smaller difference between the two methods that I would expect based on the difference between the two methods when simply using a standard scheme algorithm for solving the PDE. When using the Standard Explicit Scheme, the development of the temperature gradient over time was quite different when using Euler's Forward Method and the Implicit Euler's Method, where the former produced an output with several rough edges and unpredictable spikes, whereas the latter produced a smoother result with a round curve that was easier to follow. The RMSE achieved by the optimal parameters of the Recurrent Neural Network was very good, having an accuracy close to 10^-6. Nevertheless it took many attempts to get at this combination. The Standard Explicit Scheme on the other hand, solved the equation straightforwardly and with little computational effort. When using the Recurrent Neural Network, adding a very small l2 regularization term yielded a large RMSE. By increasing the term by a factor of 10^1, the RMSE increased by a factor of $1.3 * 10^1$. By increasing the factor of 10^2, the RMSE decreased by 200 percent. Increasing the l2 regularization one more time decreased the RMSE, and increasing it again increased

the RMSE by a factor of 50 percent. The best RMSE came from $\lambda = 1$. This illustrates that there is no linear relationship between the size of the l2 regularization term and the size of the RMSE, nor between the size and the convergence of the RMSE for the Recurrent Neural Network. Another very noticable thing is the fact that the RMSE does not improve after 50 epochs in some instances, and 200 epochs in other instances, indicating that the model either does not learn after a certain number of epochs, or that the RMSE is optimized after a low no. of epochs. Given that the RMSE I ended up with is very good, the latter conclusion seems more appropriate. Regardless of reason, in this case running over 1000 epochs is computationally expensive and unnecessary. From the results we also see that for the Activation Functions where numerical instability is an issue, the model does very well in handling this numerical instability at early iterations and fixing it very quickly. The Convolutional Neural Network, while expected to be even more configurable and potentially accurate given the right parameters, was without question capable of handling the PDE. The Loss I ended up with was, while acceptable, not comparable to the Recurrent Neural Network, where the latter gave an accuracy of 10^-5. For reference the former attained an accuracy of within 1. Using the Convolutional Neural Network, by increasing the no. of max pooling layers, comparison of Loss w. 3 rather than 1 max pooling layer, the loss now converged, and indicated it could become better by running more epochs. When using only 1 Max pooling layer, the loss was not converging, a key characteristic I am looking for, to be capable of further optimization. The difference in Loss when changing the pooling layers and sizes were quite small, making it clear that the model was fitted well for the pooling operation. Adding l2-regularization had much larger effects on the Loss, and a less than perfect l2-term destroyed the predictive capability of the model completely. Comparing with the Recurrent Neural Network, a correct l2-strength was more detrimental in the CNN, where the worst l2-term yielded a loss 10^8 times worse than the optimal one. In the case of the RNN, the worst l2-term yielded an RMSE 10^5 times worse than the optimal one, making clear that even though the Recurrent Neural Network thoroughly outperformed the Convolutional Neural Network in this case, the importance of having a correct l2-regularization, should one choose to add one, is unquestionable.

It must also be mentioned that the modification and addition of parameters to the Recurrent Neural Network code was an easier operation. In this case I was able to compare 120 different combinations of Activation Functions, Hidden Layer sizes, Momentum terms and Learning rate, something I was not able to do for the Convolutional Neural Network because adding each of these parameters and implementing them meant editing parts of the Convolutional Neural Network setup with regards to normalized data and dimensions, which was more mathematically and computationally expensive. It is therefore a possibility that the RNN would not outperform the CNN given the latter had recieved the optimal parameters, but finding such parameters was beyond the capacity for this project.

4.1 Future Work

The Convolutional Neural Network gave a sub-optimal RMSE, and it took a long time to compute given the current model architecture. Mitigating this issue involves optimizing the parameters and other ways of normalizing the data. A rigid grid search as well as implementing a TensorFlow comparison of the same networks is also desired. An easily recognizable area of improvement in this case is using the dot product as the main mathematical operation during the forward pass. This is computationally expensive and is easily improved by using batches and depthwise/pointwise convolution. [3]

5 Appendix

The full mathematical, analytical solution for the Forward Euler's Method is given as:

$$\begin{aligned}
\frac{\partial^2 u(x,t)}{\partial x^2} &= \frac{\partial u(x,t)}{\partial t}, \quad t > 0, \quad 0 < x < 1 \\
u(x,0) &= \sin(\pi x), \quad 0 < x < 1 \\
u(0,t) &= 0, \quad t \geq 0 \\
u(1,t) &= 0, \quad t \geq 0
\end{aligned}$$

Using separation of variables: Assuming $u(x,t)$ can be expressed as a product of two functions, $X(x)$ and $T(t)$:

$$u(x,t) = X(x) \cdot T(t) \tag{25}$$

, then substituting this into the heat equation and the initial conditions, and separating the variables:

$$\frac{1}{X}\frac{d^2X}{dx^2} = -\lambda^2 \quad (1) \tag{26}$$

$$\frac{1}{T}\frac{dT}{dt} = -\lambda^2 \quad (2) \tag{27}$$

Solving the equations separately, and then combining the solutions to find $u(x,t)$. Solving the first equation: The general solution to $\frac{1}{X}\frac{d^2X}{dx^2} = -\lambda^2$ is given by:

$$X(x) = A\sin(\lambda x) + B\cos(\lambda x) \tag{28}$$

Applying the boundary conditions $X(0) = 0$ and $X(1) = 0$, $B = 0$ and $\lambda = n\pi$, where n is a positive integer. So

$$X_n(x) = A_n \sin(n\pi x) \tag{29}$$

Solving the second equation: The general solution to $\frac{1}{T}\frac{dT}{dt} = -\lambda^2$ is given by

$$T(t) = C_n \cos(n\pi t) + D_n \sin(n\pi t) \tag{30}$$

Combining the solutions, yields that the general solution for $u(x,t)$ is obtained by combining the solutions for $X_n(x)$ and $T_n(t)$:

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x)\left[C_n \cos(n\pi t) + D_n \sin(n\pi t)\right] \tag{31}$$

Applying the initial condition $u(x,0) = \sin(\pi x)$, I can determine the coefficients A_n, C_n, and D_n by using Fourier series, using the initial condition $u(x,0) = sin(\pi * x)$, the Fourier series representation for the initial condition is given by

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x)\cos(n\pi t) \tag{32}$$

, where

$$A_n = \begin{cases} 0 & \text{if } n \text{ is even} \\ \frac{1}{n\pi}\int_0^1 \sin(\pi x)\sin(\pi n x)\,dx & \text{if } n \text{ is odd} \end{cases} \tag{33}$$

Using the orthogonality property of sine functions, then:

$$\int_0^1 \sin(\pi x)\sin(\pi n x)\,dx = \begin{cases} 0 & \text{if } m \neq n \\ \frac{1}{2} & \text{if } m = n \end{cases} \tag{34}$$

,so $A_n = \begin{cases} 0 & \text{if } n \text{ is even} \\ \frac{1}{n\pi}\cdot\frac{1}{2} & \text{if } n \text{ is odd} \end{cases}$ and $C_n = \frac{2}{n\pi}$. Since $D_n = 0$, the Fourier series solution for the given problem is:

$$u(x,t) = \sum_{n=1}^{\infty} \frac{1}{n\pi}\cdot\frac{1}{2}\sin(n\pi x)\cos(n\pi t) \tag{35}$$

. The full analytical solution using the Implicit Euler's Method, is:

$$u(x,t_{j+1}) = u(x,t_j) + \frac{\Delta x^2}{\Delta t}\left(u(x+\Delta x,t_{j+1}) - 2u(x,t_{j+1}) + u(x-\Delta x,t_{j+1})\right) \tag{36}$$

Applying Spatial Discretization:

$$x_i = i \cdot \Delta x, \quad i = 0, 1, 2, \ldots, N, \quad \Delta x = \frac{1}{N} \tag{37}$$

, time discretization:

$$t_{j+1} = t_j + \Delta t, \quad j = 0, 1, 2, \ldots \tag{38}$$

, applying the Implicit Euler's Method:

$$u(x_i, t_{j+1}) = u(x_i, t_j) + \frac{\Delta x^2}{\Delta t} \left(u(x_{i+1}, t_{j+1}) - 2u(x_i, t_{j+1}) + u(x_{i-1}, t_{j+1}) \right) \tag{39}$$

Expressing the system based on the given discretization:

$$u(x_i, t_{j+1}) + \frac{\Delta x^2}{\Delta t}(2u(x_i, t_{j+1})) = u(x_i, t_j) + \frac{\Delta x^2}{\Delta t}(u(x_{i+1}, t_{j+1}) + u(x_{i-1}, t_{j+1})) \tag{40}$$

The Fourier series solution again takes on the form:

$$u(x, t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x) \cos(n\pi t) \tag{41}$$

, substituting this solution into the PDE and applying the initial condition:

$$\textstyle\sum_{n=1}^{\infty} \left(-A_n (n\pi)^2 \sin(n\pi x) \cos(n\pi t) \right) = \sum_{n=1}^{\infty} A_n n\pi \sin(n\pi x) \left(-n\pi \sin(n\pi t) \right) \tag{42}$$

. For finding the coefficients An:

$$A_n = \frac{2}{n\pi} \int_0^1 \sin(\pi x) \sin(n\pi x)\, dx \tag{43}$$

And the result:

$$A_n = \frac{n\pi}{1}$$

with the Fourier series solution

$$u(x, t) = \sum_{n=1}^{\infty} \frac{n\pi}{1} \sin(n\pi x) \cos(n\pi t) \tag{44}$$

, slightly different than the one found with the Forward Euler's Method.

Converting the equation to a form interpretable to the Convolutional Neural Network using the Forward Euler's Method means: using the following equations:

$$\frac{\partial^2 u(x, t)}{\partial x^2} = \frac{\partial u(x, t)}{\partial t}, \quad t > 0, \quad 0 < x < 1 \tag{45}$$

$$\frac{\partial u(x, t)}{\partial t} \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \tag{46}$$

$$\frac{\partial^2 u(x, t)}{\partial x^2} \approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2} \tag{47}$$

$$\frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2} \tag{48}$$

$$\Delta t \left(u(x, t + \Delta t) - u(x, t) \right) \approx \frac{\Delta x^2}{2} \left(u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t) \right) \tag{49}$$

$$\Delta t\alpha \left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right) \tag{50}$$

Converting the equation to a form interpretable to the Recurrent Neural Network using the Forward Euler's Method:

$$u(x,t+\Delta t)-u(x,t)\approx \alpha \left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right) \tag{51}$$

Convolutional Form:

$$(u(x,t+\Delta t)-u(x,t))\approx \alpha \left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right)*K \tag{52}$$

Simplified:

$$\frac{u(x_i,t_{j+1})-u(x_i,t_j)}{\Delta t}\approx \frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2} \tag{53}$$

Using the recurrence relation:

$$u(x_i,t_{j+1})=u(x_i,t_j)+\alpha \tag{54}$$

$$(u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)) \tag{55}$$

The form of the Differential Equation, interpretable for the Recurrent Neural Network, using the Implicit Euler's Method: The implicit Euler's method for the one-dimensional heat equation is given by:

$$\frac{u(x_i,t_{j+1})-u(x_i,t_j)}{\Delta t}\approx \alpha\frac{\partial^2 u}{\partial x^2} \tag{56}$$

Applying a second-order central difference scheme for spatial discretization:

$$\frac{\partial^2 u}{\partial x^2}\approx \frac{u(x_i+\Delta x,t_{j+1})-2u(x_i,t_{j+1})+u(x_i-\Delta x,t_{j+1})}{\Delta x^2} \tag{57}$$

Substitute (2) into (1) and rearrange:

$$u(x_i,t_{j+1})-u(x_i,t_j)\approx \alpha\frac{\Delta t}{\Delta x^2}\left(u(x_i+\Delta x,t_{j+1})-2u(x_i,t_{j+1})+u(x_i-\Delta x,t_{j+1})\right) \tag{58}$$

Let $h=\frac{\alpha\Delta t}{\Delta x^2}$, then the equation becomes:

$$u(x_i,t_{j+1})-u(x_i,t_j)\approx h\left(u(x_i+\Delta x,t_{j+1})-2u(x_i,t_{j+1})+u(x_i-\Delta x,t_{j+1})\right) \tag{59}$$

This equation is interpretable for a Recurrent Neural Network (RNN), where $u(x_i,t_{j+1})$ depends on $u(x_i,t_j)$, $u(x_i+\Delta x,t_{j+1})$, $u(x_i,t_{j+1})$, and $u(x_i-\Delta x,t_{j+1})$. It represents a recurrence relation suitable for training an RNN, with intermediate steps: Using this ansatz:

$$\frac{\partial^2 u(x,t)}{\partial x^2}=\frac{\partial u(x,t)}{\partial t} \tag{60}$$

Substituting the ansatz:

$$f(t)\cdot g''(x)=f'(t)\cdot g(x) \tag{61}$$

. Solving the Recurrent Neural Network using the Implicit Euler's Method: Using the Implicit Euler's Method,

$$u(x_i,t_{j+1})-u(x_i,t_j)\approx h\left(u(x_i+\Delta x,t_{j+1})-2u(x_i,t_{j+1})+u(x_i-\Delta x,t_{j+1})\right) \tag{62}$$

is the interpretable form used for the Recurrent Neural Network. Using the Implicit Euler's Method on the Convolutional Neural Network:

$$(u(x,t+\Delta t)-u(x,t))\approx \alpha \left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right) \tag{63}$$

$$\approx \frac{\Delta x^2}{2}\left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right) \tag{64}$$

, the form interpretable to the Convolutional Neural Network. The convolutional Neural Network is a convolution operation with a specific kernel K:

$$(u(x,t+\Delta t)-u(x,t))\approx \alpha\cdot\left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right)*K \tag{65}$$

,

5.1 GitHub

The Python code used in this project, is found at https://github.com/ingebrigtkj/Project-3-FYS-STK-4155

References

[1] Notes on Recurrent Neural Networks, visited from 17.11.23 to 25.11.23: https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week45.ipynb

[2] Notes on Convolutional Neural Networks, visited from 27.11.23 to 08.12.23: https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week44.ipynb

[3] Article on Depthwise Convolution (MobileNet) visited on 01.12.23: https://towardsdatascience.com/understanding-depthwise-separable-convolutions-and-the-efficiency-of-mobilenets-6de3d6b62503

[4] Comparison, Performance and Appropriateness of Activation Functions, visited from 20.11.23 to 30.11.23: https://arxiv.org/abs/2109.14545

[5] More on Deep Learning Methods, specific types of Neural Networks, visited from 25.11.23 to 02.12.23: https://arxiv.org/abs/2310.20360

[6] The Modern Mathematics of Deep Learning, visited from 27.11.23 to 01.12.12: https://arxiv.org/abs/2105.04026