Using neural networks to approximate the solutions of differential equations.

Christiaan van Vliet

June 2021

Abstract

In this work neural networks are used to approximate the solutions of multiple differential equations. Three main differential equations are investigated: the mass-spring equation, the heat equation and the Navier-Stokes equations coupled with the continuity equation. Each of these equations are approached differently with the current methods, therefore these equations test the applicability of the newly proposed method on a general differential equation.

The results of various experiments performed in this work show that the method is able to approximate to great precision the mass-spring equation as well as the heat equation. However, the method does not perform well on the Navier-Stokes equations coupled with the continuity equation.

The main advantage of this method is that meshes used by this method can often be very small compared to other approximation algorithms. When a slight decrease in performance is allowed, the method can be made completely mesh free.

A huge drawback of the method is that there is no guarantee that the method will generate a reasonable approximation, for a given neural network configuration. This makes it necessary to look for a good configuration of the neural network. At this moment, this can only be done using trial and error and is therefore very time-consuming. Sometimes no good configuration is found at all.

Another drawback is that the generated approximation does not satisfy the boundary and initial conditions. Multiple ways are proposed in this work to tackle this problem, but they introduce much more complexity and a performance decrease.

The neural networks used in this work are generated using software called PyTorch, which comes with an automatic analytic differentiation tool. This part of PyTorch makes it possible to approximate differential equations using neural networks.

Contents

1 Introduction 1

2 Artificial neural networks 2

2.1 Neurons 2

2.2 Fully connected layers 2

2.3 Mathematical model of fully connected layers 3

2.4 Nonlinear activation function 4

2.5 Differentiable activation function 4

2.6 Convolutional layers 4

2.7 Loss function 5

2.8 Optimization algorithm 5

3 Implementing a neural network to solve differential equations 6

3.1 Algorithm description 6

3.1.1 Discussion on the algorithm 7

3.2 Activation function 7

3.3 Probability function 8

3.4 Sampling 8

3.5 PDE loss 9

3.6 Boundary loss 9

3.7 Learning rate adaption 9

4 Experimental results 10

4.1 Solving the mass-spring ODE 10

4.2 Solving the damped mass-spring ODE 11

4.3 Solving the heat equation 13

4.3.1 Unweighted loss and uniform sampling 14

4.3.2 Weighted loss with fixed weights 15

4.3.3 Weighted loss with loss-dependent weights 18

4.3.4 Weighted loss and loss-dependent sample distribution 20

4.3.5 Discussion on the results of the heat equation 22

4.4 Solving the Navier-Stokes and Continuity equation 22

4.4.1 Implementation in PyTorch 24

4.4.2 Convergence 25

4.5 Mass-spring system 2 25

4.5.1 Solving the mass spring equation and results 25

4.5.2 Detailed problem determination 27

4.5.3 Proposed solutions and their results 28

5 Conclusions from the experiments and further research recommendations 36

Appendices 38

A In depth description of PyTorch implementation 38

A.1 Mass-spring system 1 38

A.2 Sampling points from a custom distribution 41

A.2.1 Mesh free solving 41

A.2.2 Updating the distribution 42

1 Introduction

Neural networks have made their entrance in various fields of computer science. They have proven themself very useful in detecting objects in images [8], recognizing speech by a computer [3] and reading and processing human typed text [4]. In this work, it is proposed to use a neural network to approximate the solutions of a differential equation. The software used to create the neural network, which is called PyTorch, comes with an automatic analytic differentiate tool [1], which makes it possible to check if the output of a neural network satisfies a given differential equation.

Differential equations have proven themself to be notoriously hard to solve, while playing a prominent role in physics. There are multiple fields of physics which are solely about solving one differential equation, such as fluid flow analysis, structural analysis and heat flow analysis. It is important to have algorithms which obtain approximations to the solutions of these equations, since an exact solution to most of these problems have not been found so far.

This work connects the recent developments in computer science with the need for approximation algorithms in physics. Approximations to differential equations are searched using neural networks. In section 2 the general working of a neural network is explained. In section 3 it is explained how differential equations can be solved using a neural network and how the boundary conditions of a differential equation are being satisfied using the same neural network. In section 4 the results of the performed experiments are shown and discussed. The conclusions of the experiments are shown in section 5, as well as a discussion on these results. In appendix A an in-depth explanation of the implementation in PyTorch is given as well as an in-depth explanation of the used sampling algorithm, which was used heavily in the experiments.

2 Artificial neural networks

In this report, artificial neural networks are used to approximate solutions to differential equations. In this section, a short introduction is given into neural networks. Most information is available in [6], but also less formal in the documentation of various deep learning libraries. PyTorch is used in this work, which provides information in [1].

Standard neural networks consist of layers of neurons, called fully-connected layers. Neurons are non-linear functions with an internal state. When a neural network is trained to give a certain output, the internal states of the neurons are altered to match the desired output.

These architectures are a base for more sophisticated architectures. In this work the slightly more complex convolutional layer is also used, which gave much better results during some of the experiments.

2.1 Neurons

A neuron is a function with a vector $\boldsymbol{x}$ as input and a scalar y as output. It is usually graphically illustrated as in figure 1. It has an internal state, which consists of a weights vector $\boldsymbol{w}$ and a bias scalar b.

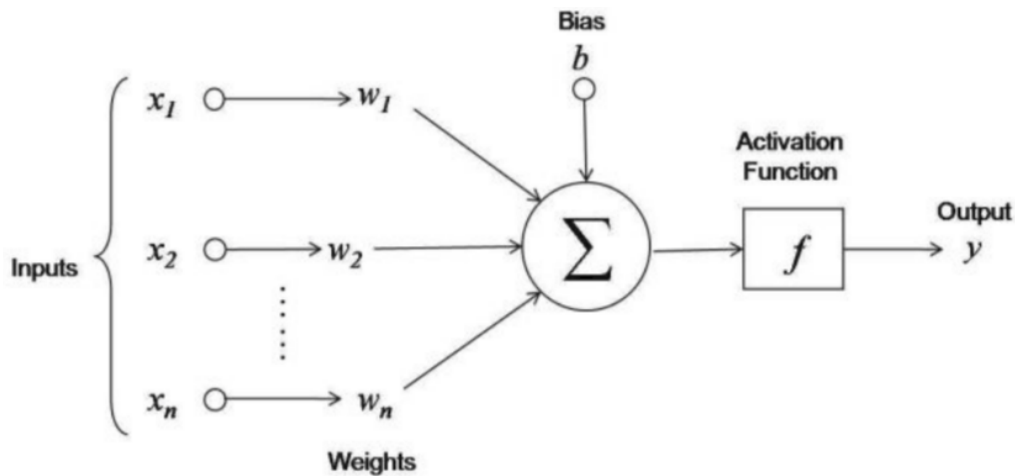

Figure 1: Graphical illustration of a neuron with input vector $\boldsymbol{x}$, weights vector $\boldsymbol{w}$, bias scalar b and output scalar y.

The output of the neuron is calculated as follows: The standard inner product is taken between the input vector $\boldsymbol{x}$ and the weights vector $\boldsymbol{w}$. Then, the bias b is added and an activation function $f(x)$ is applied, which yields the output of the neuron: $y = f(\boldsymbol{x}^T\boldsymbol{w} + b)$. The activation function will be discussed in more detail in section 2.4

2.2 Fully connected layers

A fully connected layer is a layer where multiple neurons are placed next to each other. Each neuron has the same input and activation function, so the neurons only differ in their weights and biases. When the outputs of each neuron in the layer are placed in a vector the output vector $\boldsymbol{y}$ of the layer is obtained. Note that the length of the output vector is the same as the number of

neurons in the layer. Also note that the length of the output vector is generally not equal to the length of the input vector.

A complete neural network is obtained when layers are stacked behind each other. That means, the output of the first layer serves as input of the second layer, the second layer serves as input of the third layer, and so on until the final layer. In figure 2 a graphical representation is shown of a neural network.

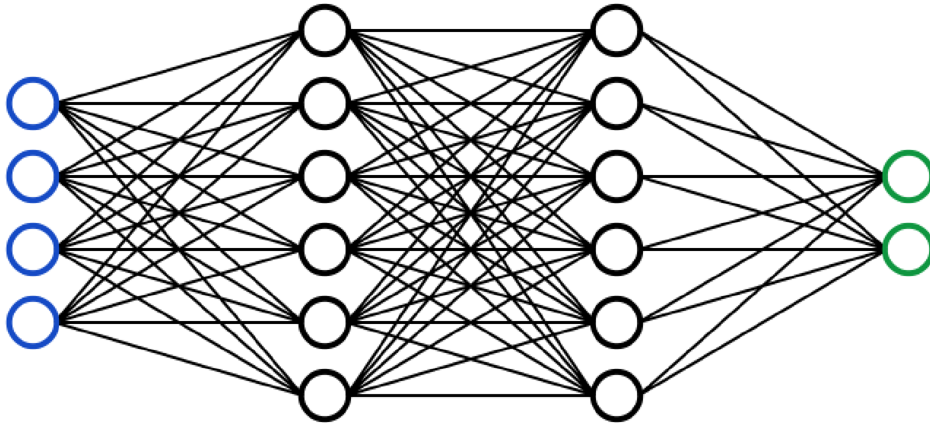

Figure 2: Graphical illustration of a neural network. The black circles in the middle represent neurons, ordered in two layers. The blue circles are the input vector components. The green circles represent the output layer and are therefore also the output vector of the neural network.

2.3 Mathematical model of fully connected layers

The simplest type of neural network consists of fully connected layers. A fully connected layer can be modelled by a matrix multiplication. Define matrix W_i such that all weight vectors of layer i produce the rows of W_i: $W_i = \begin{bmatrix} \boldsymbol{w}_{1,i} & \boldsymbol{w}_{2,i} & \cdots & \boldsymbol{w}_{n,i} \end{bmatrix}^T$, where n is the number of neurons in the layer. Similarly, all the bias scalars can be put in a vector: $\boldsymbol{b}_i = \begin{bmatrix} b_{1,i} & b_{2,i} & \cdots & b_{n,i} \end{bmatrix}^T$. Then, the output vector $\boldsymbol{y}_i$ of layer i is given in equation 1:

$$\boldsymbol{y}_i = \boldsymbol{f}(W_i \boldsymbol{x}_i + \boldsymbol{b}_i) \tag{1}$$

In equation 1, $\boldsymbol{f}(\boldsymbol{x})$ performs the activation function f to all elements of the vector $\boldsymbol{x}$ and $\boldsymbol{x}_i$ is the input of layer i.

The whole neural network can now be modelled as in equation 3, where n is the number of layers, $\boldsymbol{x}$ is the input and $\boldsymbol{y}$ is the output:

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}(W_1 \boldsymbol{f}(W_2 \boldsymbol{f}(W_3 \cdots \boldsymbol{f}(W_n \boldsymbol{x} + \boldsymbol{b}_n) \cdots + \boldsymbol{b}_3) + \boldsymbol{b}_2) + \boldsymbol{b}_1) \tag{2}$$

During the training phase of a neural network, the weights are adjusted by an optimization algorithm, which is briefly discussed in paragraph 2.8. Therefore equation 2 is only valid when the neural network was already trained and is being evaluated. During training, equation 2 becomes equation 3:

$$\boldsymbol{y}(\boldsymbol{x}, W_1, W_2, ..., W_n, \boldsymbol{b}_1, \boldsymbol{b}_2, ..., \boldsymbol{b}_n) = \boldsymbol{f}(W_1 \boldsymbol{f}(W_2 \boldsymbol{f}(W_3 \cdots \boldsymbol{f}(W_n \boldsymbol{x} + \boldsymbol{b}_n) \cdots + \boldsymbol{b}_3) + \boldsymbol{b}_2) + \boldsymbol{b}_1) \tag{3}$$

2.4 Nonlinear activation function

The importance of a nonlinear activation function is shown in this paragraph. Suppose a linear activation function is used: $f(x) = cx$, where c is some constant. The output of the neural network would be:

$$\begin{aligned} \boldsymbol{y} &= c(W_1 c(W_2 c(W_3 \cdots c(W_n \boldsymbol{x} + \boldsymbol{b}_n) \cdots + \boldsymbol{b}_3) + \boldsymbol{b}_2) + \boldsymbol{b}_1) \\ &= c^n W_1 W_2 W_3 \cdots W_n \boldsymbol{x} + c^{n-1} W_1 W_2 \cdots W_{n-1} \boldsymbol{b}_n + \cdots + c W_1 \boldsymbol{b}_2 + \boldsymbol{b_1} \\ &= P\boldsymbol{x} + \boldsymbol{k} \end{aligned} \tag{4}$$

Where P is some matrix and $\boldsymbol{k}$ is some vector independent of $\boldsymbol{x}$. Note that the output is a linear function of the input, so the neural network is only able to create a linear model.

Generally, a neural network should not create a linear model since there are more efficient methods of creating and solving linear models. Therefore, the described behaviour is not desirable for any problem which is solved using a neural network. For this usage specifically the undesirability is even more obvious; solutions to differential equations are in general not linear, so a linear model would not suffice. When a nonlinear activation function is used, equation 3 cannot be reduced into the $\boldsymbol{y} = P\boldsymbol{x} + \boldsymbol{k}$ form, so the output will be a nonlinear function of the input.

2.5 Differentiable activation function

From equation 3, it is clear that $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ can only be calculated if $\boldsymbol{y}(\boldsymbol{x}, W_1, W_2, ..., W_n, \boldsymbol{b}_1, \boldsymbol{b}_2, ..., \boldsymbol{b}_n)$ is differentiable. Moreover, it is required that $\boldsymbol{y}$ is differentiable everywhere, since $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ will be discontinuous if it was not differentiable. Most physical differential equations however require differentiable solutions. In this work, $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ and $\frac{\partial^2 \boldsymbol{y}}{\partial \boldsymbol{x}^2}$ are calculated in order to check if $\boldsymbol{y}(\boldsymbol{x}, W_1, W_2, ..., W_n, \boldsymbol{b}_1, \boldsymbol{b}_2, ..., \boldsymbol{b}_n)$ is a solution of the considered differential equation, so the derivative plays a crucial role in this work. A differentiable activation function is therefore required.

2.6 Convolutional layers

A more sophisticated form of a neural network makes use of the so-called convolutional layer. Experimentally, it is found that these layers often perform better than fully connected layers in image recognition problems [8].

A convolutional layer does not consist of neurons, like a fully connected layer. The whole layer has one weights vector $\boldsymbol{w}$ and one bias vector $\boldsymbol{b}$, which form the parameters of the layer. Equation 5 shows how the output of the layer is calculated, where $\boldsymbol{y}$ is the output of the layer, $\boldsymbol{x}$ is the input, $\boldsymbol{f}$ is the activation function and $*$ is the convolution function.

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{b} + \boldsymbol{w} * \boldsymbol{x}) \tag{5}$$

Note that $\boldsymbol{w}$ and $\boldsymbol{x}$ are both of finite length. The length of $\boldsymbol{w}$ is a configuration parameter of the neural network. Like most configuration parameters, the best length of $\boldsymbol{w}$ is found by trial and error. Also note that by definition of the convolution: $z[n] = (f * g)[n] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$. Since both $\boldsymbol{x}$ and $\boldsymbol{w}$ have finite elements, this sum reduces to $z[n] = (x * w)[n] = \sum_{m=1}^{L} x[n-m]w[m]$, where L is the length of $\boldsymbol{w}$. Note that this is undefined if $n < L + 1$ or $n > p + 1$, where p is the length of $\boldsymbol{x}$. Therefore, $z[n]$ is defined for $p - L + 1$ points, so the dimension of the output vector of the convolutional layer is $L - 1$ less than the input vector.

2.7 Loss function

The loss function measures the error between the output of the neural network and the desired result. Multiple options are available for this function, but in this work the L2 norm is used: $L = \sqrt{\sum (\boldsymbol{p} - \boldsymbol{y})^2}$, where $\boldsymbol{p}$ is the value outputted by the neural network and $\boldsymbol{y}$ is the desired value. Note that a good choice of loss function is crucial to the performance of the optimization algorithm. It is unknown if there is a systematic way to determine the best suitable loss function. In this work, well performing loss functions are always found by trial and error.

2.8 Optimization algorithm

The optimization algorithm adjusts the parameters of the neural network such that the loss function is minimized. The optimization algorithm is therefore directly responsible for obtaining an approximation of the solution of the differential equation.

Choosing or creating a good optimization algorithm is a complex task, since the function to optimize is highly non-linear. In this work, an optimization algorithm is used, which is commonly used when training neural networks for other tasks, such as computer vision or natural language processing. This algorithm is called Adam [7] and it is based on stochastic gradient descent. Implementing this algorithm is relatively simple, since it comes with PyTorch.

3 Implementing a neural network to solve differential equations

In this section, the implementation of a neural network to solve differential equations is discussed. Implementing a neural network by itself is relatively simple when using the proper tools. The algorithm is programmed in Python and uses a deep learning library called PyTorch.

PyTorch is very much comparable to Numpy, since it is optimized for matrix and vector algebra. PyTorch is capable of doing its calculations on a GPU, which makes it very fast. It comes with a lot of build-in functions and tools, making it an easy library to use.

For the first experiments, Google Colab is used as a remote server to train the neural network on. By using the GPU available in Colab training, the neural network can be done in reasonable time, which is not possible on a CPU only machine. Later on, a local machine with a GPU became available which has been used ever since.

3.1 Algorithm description

The goal of this work is to approximate $\boldsymbol{y}(\boldsymbol{x})$ which is a solution to the differential equation of interest. Let this differential equation be represented by the differential operator L, and a solution of L satisfies $L[\boldsymbol{y}(\boldsymbol{x})] = 0$. Now, given the neural network output $\boldsymbol{g}(\boldsymbol{x})$, the error can be calculated: $\delta(\boldsymbol{x}) = L[\boldsymbol{g}(\boldsymbol{x})]$.

If $\delta(\boldsymbol{x}) = 0$ everywhere, then an exact solution to the differential equation is obtained. By defining the loss function to be $L = \sqrt{\sum \delta(\boldsymbol{x})^2}$, we can use an optimization algorithm to minimize L, as described in section 2.8.

Using these tools, the algorithm consists of the following steps:

- Define the space in which the differential equation is solved and sample n random points $\boldsymbol{x}_i$ from this space.
- Get the neural network approximation $\boldsymbol{g}(\boldsymbol{x}_i)$.
- Calculate the error of the approximation $\delta(\boldsymbol{g}(\boldsymbol{x}_i))$ and the accompanying loss $L_{PDE} = \sqrt{\sum_{i=1}^{n} \delta^2(\boldsymbol{g}(\boldsymbol{x}_i))}$.
- Define the boundary conditions of the differential equation and sample n random points x_i from each boundary.
- Get the neural network approximation $\boldsymbol{g}(\boldsymbol{x}_i)$ for each of the boundary points.
- Calculate the error of the approximation at the boundaries and the accompanying losses: $L_{Boundary} = \sqrt{\sum_{i=1}^{n} (\boldsymbol{g}(\boldsymbol{x}_i) - \boldsymbol{y}(\boldsymbol{x}_i))^2}$, where $\boldsymbol{y}(\boldsymbol{x}))$ is the prescribed boundary condition.
- Sum all the losses: $L = L_{PDE} + \sum_{Boundaries} L_{Boundary}$.
- Use the optimization algorithm to change the parameters of the neural network such that L is minimized.
- Repeat the process.

In order to have a completely mesh free algorithm, custom probability distributions can be used to sample the input points $\boldsymbol{x}_i$ from. Usually, a uniform distribution is used. However, if the probability function depends on earlier found errors, as described in section 3.3, points with a high

error can be sampled more often. This process does need a small mesh to model the probability distribution.

Note that the loss depends on the number of sampled points n. Sometimes the loss is compensated for this by putting a factor $\frac{1}{n}$ in front of the sum. In this work that is not done since n is constant during a training session, therefore the optimization algorithm would simply see a scaled version of the uncompensated loss. This only adds unnecessary complexity and is therefore omitted.

3.1.1 Discussion on the algorithm

In this paragraph a few upsides and downsides of the algorithm are described.

First of all, a mesh is not required since points are randomly sampled. This is a huge advantage over other numerical methods, which require large meshes. However, as we will see in section 4.3.4, a small mesh used to create a custom probability density function from which the random points $\boldsymbol{x}_i$ are sampled will increase the performance of the neural network. This mesh is still orders of magnitude smaller than the meshes constructed in conventional methods.

Secondly, it is not possible to determine the maximum error of the approximation, $\max_{\boldsymbol{x}} (\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{g}(\boldsymbol{x}))$. Heuristically, it is possible to say that a lower L should give a better approximation, but as is seen during the experiments, this is not always the case.

Thirdly, the boundary conditions cannot be fixed. From equation 3, it is clear that only the parameters of the neural network can be varied. It would be possible to constrain the parameters such that the boundaries are satisfied, but this would prohibit the optimization algorithm to vary the parameters. The differential equation can therefore not be approximated if the parameters are constrained to the boundaries. The alternative is to include the error at the boundaries in the loss function, as is done in this work and is described in section 3.1. It is however possible to generate a weighted loss $L = w_{PDE} \cdot L_{PDE} + \sum_{Boundaries} w_b \cdot L_{Boundary}$, where w_{PDE} and w_b are the weights of the individual loss terms, such that $w_{PDE} + \sum_{Boundaries} w_b = 1$. Using these weights, it possible to let the approximation only satisfy the boundaries in the beginning phase of the training and forget about the differential equation completely.

Fourthly, it is not possible to determine in advance of the training phase whether the approximation converges to the analytical solution. It is therefore possible that after a time-consuming training session, the approximation does not even come close to the analytical solution. This is a huge disadvantage as was seen in the experiments, since bad approximations occur at seemingly random moments. Since there are a lot of configuration options on the neural network, it is a very time-consuming process to try whether the configuration yields good results for all configurations.

3.2 Activation function

From section 2.4, it is clear that the activation function plays a crucial role. The activation function is required to be nonlinear and differentiable. If $W_i = I$ and $\boldsymbol{b}_i = \boldsymbol{0}$ for all $0 < i \leq n$, it is clear from equation 3 that if the activation function changes the order of the input, the output might be too big or too small to handle it properly.

For example, if the activation function reduces the order of magnitude of the input by 1, a 10 layer neural network will output numbers 10 order of magnitudes smaller than the input. For any reasonable input x, this is generally undesirable.

Therefore, an activation function that does not change the order of the input is required, or an activation function that only changes very large or very small input values.

Examples of suitable activation functions are $f(x) = \tanh x$, $f(x) = \frac{1}{1+e^{-x}}$, $f(x) = \arctan x$ and $f(x) = \ln(1+e^x)$. Note that in image recognition, the activation $f(x) = \max(0, x)$ has shown good results. Since this function is not differentiable at $x = 0$, a differentiable approximation can be used: $f(x) = \ln(1+e^x)$. This function is however much less successful, as showed in the experiments.

It is not required that the activation function after each layer is the same activation function. An activation function is generally chosen by trial and error. In this work, the function $f(x) = \tanh(x)$ performed relatively good, but the output range is only [-1, 1]. For a lot of differential equations, the solution is expected to be non-negative and not too large, albeit larger than 1. To overcome this problem, the activation of the last layer is often chosen as $f(x) = e^x - 0.5$. This allows not only for larger values than 1, but also for the range [0,1] to be reached using reasonable values.

3.3 Probability function

A bounded region must be specified for which the differential equation is solved. For each point on that region, a probability that this point is sampled is desired. That way, points with a large error are more often sampled than points with a low error. The optimization algorithm is then able to focus on the errors it made rather than improving regions that are already good. Therefore, a mesh is constructed on that region and the loss is calculated on those points. Let $c(\boldsymbol{x})$ be the loss at point $\boldsymbol{x}$. This loss is used as a probability density function, after normalization. Since $c(\boldsymbol{x})$ is only known at the mesh points, a linear interpolation is used to interpolate the probability on the whole region.

Note that the mesh is only used for probability estimation. The neural network estimation of the solution is also optimized at points other than the mesh points. It is also possible to assume a uniform probability distribution, which needs no mesh creation at all. The method then becomes a truly mesh-free method, but it converges more slowly to a solution.

3.4 Sampling

When the numerical probability density function is known, points must be sampled from it. Since the distribution might be more dimensional, up to 4 dimensions in this work, sampling is nontrivial.

A pivot dimension is chosen. The distribution is summed in all dimensions except the pivot dimension. Remember that it is a numerical distribution, consisting of discrete points, which allows the use of a sum instead of an integral. After summing, a one dimensional distribution is obtained, which is transformed into a cumulative distribution. Subsequently, there is an injective and surjective function from the pivot dimension to $[a, 1]$, when an interpolation is used for all points in between the known mesh points. Here, a is the probability of the first point in the pivot dimension. The $[a, 1]$ range is undesirable, so the first element in the pivot dimension is shifted by a small number ϵ, which allows for an artificial mesh point at $x = 0$ with value 0, so that the interpolation range becomes $[0, 1]$.

In order to pick a random point from the cumulative function, a random number from the uniform distribution between 0 and 1 is drawn. This corresponds with some point on the pivot dimension, given by the inverse of the interpolated cumulative probability function. Now, on the pivot dimension, a point is chosen.

All dimensions serve as pivot dimension once. All drawn points together form a random point drawn from the numerical probability distribution.

PyTorch does not have a built-in function for this procedure, which means that it must be implemented in the program itself. Since PyTorch functions are generally much faster than built-

in Python functions, the procedure is implemented with PyTorch functions only. A fully detailed explanation of the algorithm is given in appendix A.2.

3.5 PDE loss

Once a mini-batch of points has been drawn from the distribution, the loss can be calculated. For each point $\boldsymbol{x}_i$, the error $\delta(\boldsymbol{x}_i)$ is calculated. Normally, the L2 norm, also called mean squared error, is used as loss function. Since the desired value of the error is zero, calculating the PDE loss is given by $\sqrt{\sum_{i=1}^{n} \delta^2(\boldsymbol{x}_i)}$.

Implementing the calculation of the function $\delta(\boldsymbol{x})$ is easily done using PyTorch. PyTorch provides automatic analytic differentiation functions, which allows for quick implementation of differential equations. This is one of the huge benefits of using PyTorch, as implementing a differentiation function manually would require a lot of effort.

3.6 Boundary loss

Not only the differential equation should be satisfied, this also applies to the boundary conditions. This is done by comparing the neural network value at the boundary with the known boundary value. The L2 norm is also used here. Let $\boldsymbol{x}_i$ be a sampled boundary point, then the unweighted boundary loss is given by $L_b = \sqrt{\sum_{i=1}^{n} \left[g(\boldsymbol{x}_i) - y(\boldsymbol{x}_i)\right]^2}$, where $\boldsymbol{y}$ is the boundary value and $\boldsymbol{g}$ is the neural network value.

As we will see in the experiments, weighted versions of the loss often provide a performance increase: $L_b = \sqrt{\sum_{i=1}^{n} w_i \left[g(\boldsymbol{x}_i) - y(\boldsymbol{x}_i)\right]^2}$ where w_i represents the weight of the ith term. Note that multiple boundaries may be defined and that the total loss is simply all the losses summed together. If the output of the neural network has multiple dimensions the sum is over each dimension of the input $\boldsymbol{x}_i$ as well as the output dimensions.

3.7 Learning rate adaption

The loss is passed to the optimization algorithm. In order for the optimization algorithm to work, it must be configured properly. One of the most important configuration options is the learning rate. In the Adam algorithm, it is one of the factors by which the gradients of the loss with respect to the parameters are scaled. When a optimization algorithm is used without a well-chosen learning rate, the neural network will not converge to the desired value. It is therefore extremely important to perform a learning rate optimization, which is usually done by trial and error.

During the beginning of training, the neural network might need a higher learning rate than further down the training process. A learning rate adaption scheme is set up to adjust the learning rate during training. This adaption scheme comes with its own configuration options, which in turn requires its own trial and error optimization.

4 Experimental results

Multiple experiments have been done using the neural network method. Four main experiments have been performed: solving a mass-spring system (ODE), solving the heat equation (One dimensional PDE), solving the Navier-Stokes equations coupled with the continuity equation (Multi dimensional PDE) and solving a mass-spring system with the Navier-Stokes solver. Inside each main experiment, lots of sub-experiments were performed, each with a slightly different configuration.

4.1 Solving the mass-spring ODE

A mass spring system is solved by the neural network. The differential equation is given by $\frac{d^2u}{dt^2}(t) + cu(t) = 0$, where u is the distance of the mass measured from the equilibrium position and $ck > 0$. In these experiments c is chosen as $\frac{1}{10}$.

The initial conditions where chosen as $u(0) = 1$ and $\frac{du}{dt}(0) = 0$. The analytical solution to this equation is known and given by $u(t) = \cos \sqrt{c}t$. The neural network approximation is made on the range $0 \leq t \leq 30$.

For the neural network, the following configuration is chosen: learning rate of $5 \cdot 10^{-4}$, 30 mini-batches per epoch, mini-batch size of 10 and the $\tanh(x)$ activation function. A plot of the neural network approximation and the analytical solution is given in figure 3.

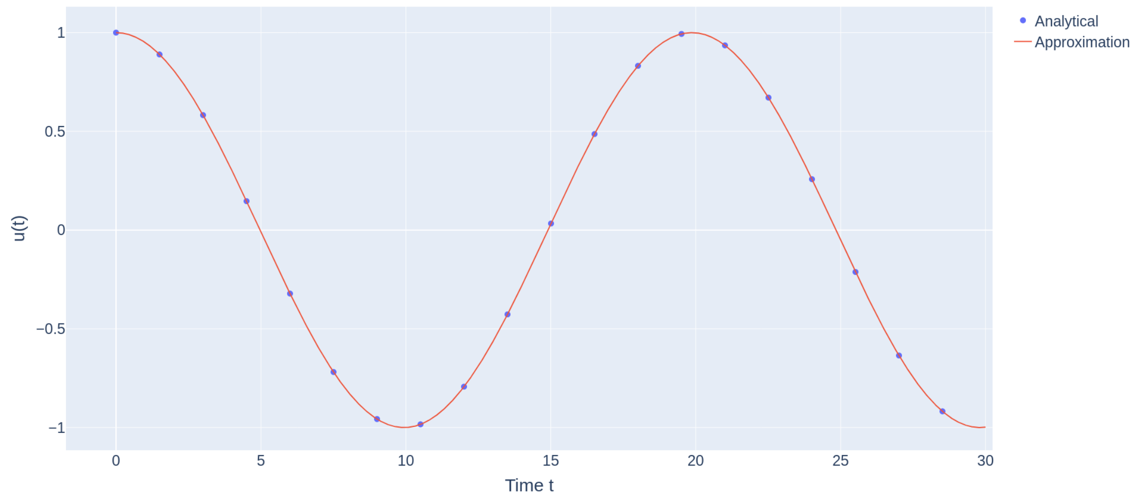

Figure 3: Neural network approximation of a mass-spring system after 1800 epochs together with the analytical solution. The loss is $1.1 \cdot 10^{-5}$ and the maximum difference between the approximation and the analytical solution is $1.5 \cdot 10^{-3}$.

The method diverged for larger values of the initial conditions or larger values of c, such as $u(0) = 30$ or $c = 300$, the method diverged. This resulted in a computer memory overflow. The experiment was repeated multiple times and with different configurations but the method always diverged.

Later in this work the same system is solved using a more sophisticated method, which will fail for a large part. These two methods are extensively described in appendices A.1 and A.2, so that a comparison can be made. Note that this experiment is repeated using the x^3, $\ln(e^x + 1)$ and e^x activation functions, but that these activation functions overflow the computer memory after

some epochs. Also, $\sin(x)$ is used as activation function and the results are similar to the $\tanh(x)$ activation function.

4.2 Solving the damped mass-spring ODE

The mass-spring system is altered by a damping term. The differential equation becomes $\frac{d^2u}{dt^2}(t) + k \cdot u(t) + c \cdot \frac{du}{dt} = 0$. If $c^2 > 4k$, the system is called over damped and the analytical solution is given by expression 6, where $u(0) = 1$ and $\frac{du}{dt}(0) = 0$ are used as boundary conditions.

$$u(t) = e^{-\frac{1}{2}ct}\left[(1 - \frac{4k}{c^2})^{-\frac{1}{2}} \sinh(t \cdot \sqrt{\frac{1}{4}c^2 - k}) + \cosh(t \cdot \sqrt{\frac{1}{4}c^2 - k})\right] \tag{6}$$

If $c^2 < 4k$, the system is called under damped and the solution is given in equation 7, where the same boundaries are used.

$$u(t) = e^{-\frac{1}{2}ct}\left[(\frac{4k}{c^2} - 1)^{-\frac{1}{2}} \sin(t \cdot \sqrt{k - \frac{1}{4}c^2}) + \cos(t \cdot \sqrt{k - \frac{1}{4}c^2})\right] \tag{7}$$

If $c^2 = 4k$, the system is called critically damped and the solution is given by equation 8, where the same boundaries are used again.

$$u(t) = e^{-\frac{1}{2}ct}\left[1 + \frac{1}{2}ct\right] \tag{8}$$

Using the neural network, the damped mass-spring system is solved. Firstly, a system that is slightly over damped is solved, so $c^2 > 4k$. No oscillations therefore occur. The result is seen in figure 4.

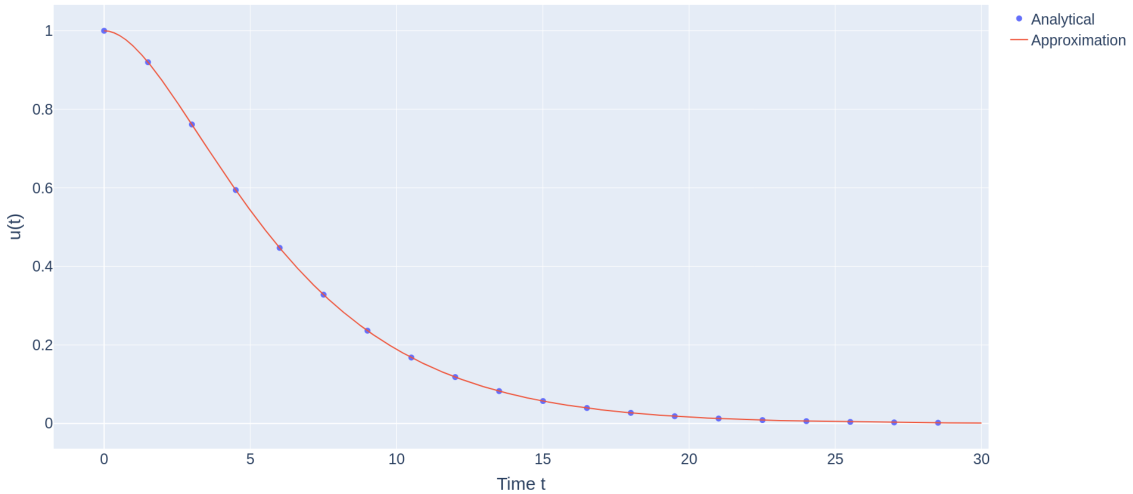

Figure 4: Neural network approximation together with the analytical solution of an over damped mass-spring system. The analytical solution is given by equation 6. This result is achieved after 500 epochs with a loss of $1.3 \cdot 10^{-4}$. The maximum difference between the analytical result and the approximation is $6.4 \cdot 10^{-4}$.

Experiments are also performed for heavily over damped systems which yielded similar results, some converging slightly faster. Secondly, a slightly under damped mass-spring system is solved, so $c^2 < 4k$ and oscillations occur. The result is seen in figure 5.

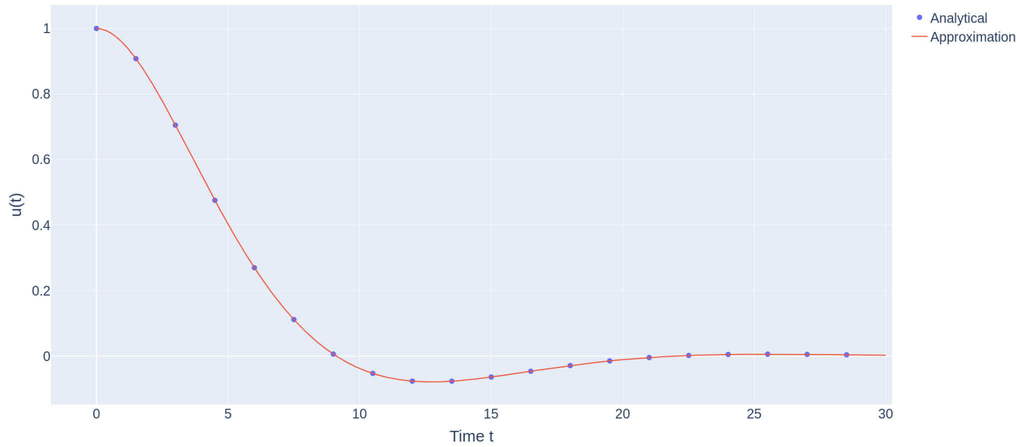

Figure 5: Neural network approximation together with the analytical solution of an under damped mass-spring system. The analytical solution is given by equation 7. This result is achieved after 2000 epochs with a loss of $1.3 \cdot 10^{-5}$. The maximum difference between the analytical result and the approximation is $3.4 \cdot 10^{-4}$.

Experiments are also performed for more heavily under damped systems, for which one result is shown in figure 6.

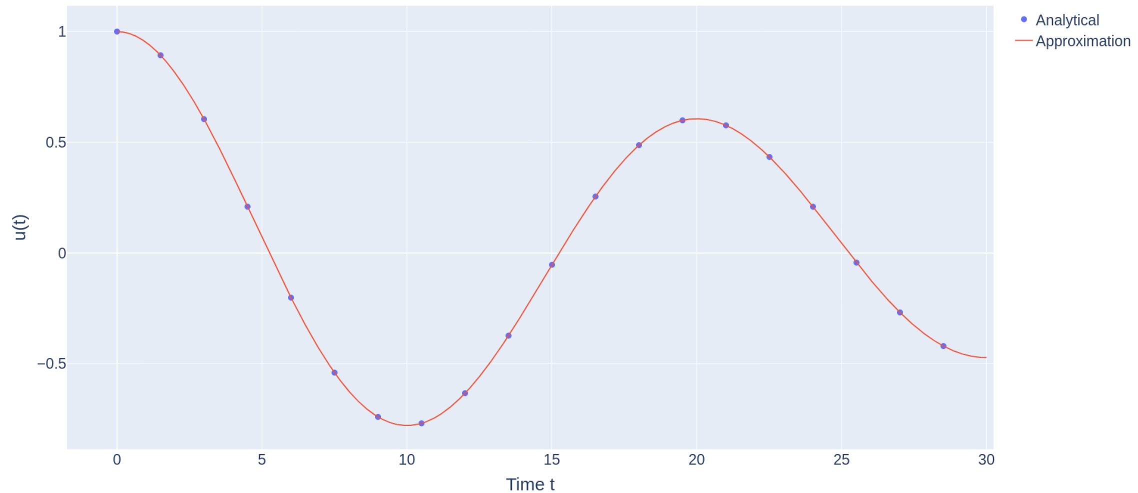

Figure 6: Neural network approximation together with the analytical solution of a heavily under damped mass-spring system. The analytical solution is given by equation 7. This result is achieved after 1200 epochs with a loss of $1.0 \cdot 10^{-4}$. The maximum difference between the analytical result and the approximation is $2.3 \cdot 10^{-3}$.

Experiments were also performed for the critically damped system, so $c^2 = 4k$. The result is seen in figure 7.

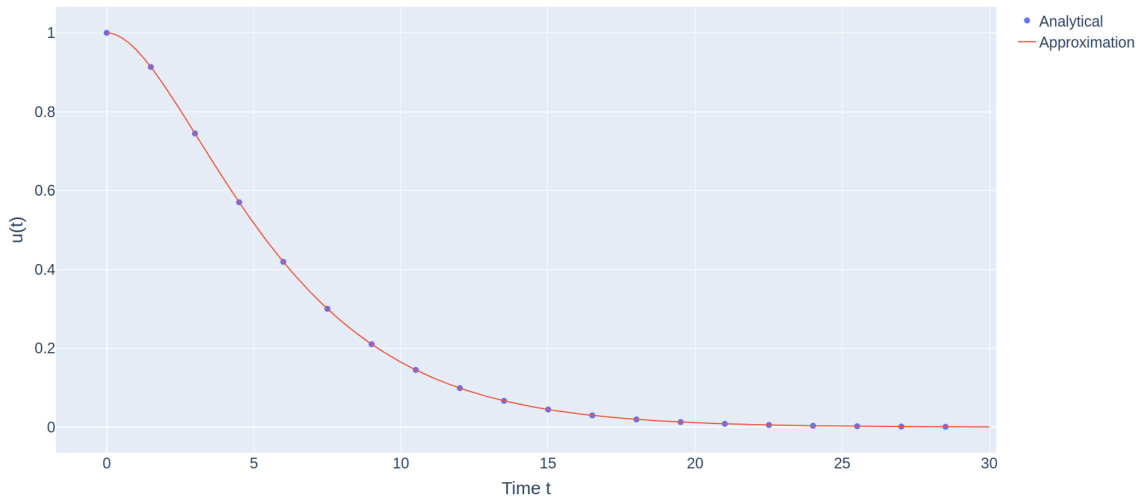

Figure 7: Neural network approximation together with the analytical solution of a critically damped mass-spring system. The analytical solution is given by equation 8. This result is achieved after 850 epochs with a loss of $6.4 \cdot 10^{-5}$. The maximum difference between the analytical result and the approximation is $3.1 \cdot 10^{-4}$.

Similar to the undamped mass-spring system, this experiment is repeated using the x^3, $\ln(e^x+1)$ and e^x activation functions, but the results do not converge. $\sin(x)$ is also used as activation function, wherein the results are similar to the $\tanh(x)$ activation function.

4.3 Solving the heat equation

The method was also used to solve the one dimensional heat equation. The heat equation is given by $\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}$, where $k \in \mathbb{R}$ and $k > 0$. This equation can physically be interpreted as the temperature in a one dimensional rod, where heat is only flowing in the direction parallel to the rod. Here, $u(x,t)$ represents the temperature at position x at time t. k represents the properties of the material of which the rod is made, as well as the chosen units of time, distance and temperature. To simplify calculations, this section assumes those quantities are unitless. After all, choosing some set of units only changes k.

An analytical solution to this equation is known in the form of a Fourier-series, which is given in equation 9. The used boundary conditions are $u(0,t) = u(L,t) = 0$, with L the length of the rod, usually taken to be 50 in this work. The used initial condition is $u(x,0) = f(x)$, where f is required to be piecewise continuously differentiable.

$$u(t) = \sum_{n=1}^{\infty} B_n \sin\left(\frac{n\pi x}{L}\right) e^{-\frac{n^2\pi^2 kt}{L^2}} \tag{9}$$

$$B_n = \frac{2}{L}\int_0^L \sin\left(\frac{n\pi x}{L}\right) f(x)\, dx \tag{10}$$

The exact solution cannot be plotted since the integral in equation 10 and the infinite sum in equation 9 cannot be evaluated exactly. In order to make a plot, the sum in equation 9 is truncated and the integral in equation 10 is approximated using a finite sum. The result is a semi-exact plot and it is shown in figure 8.

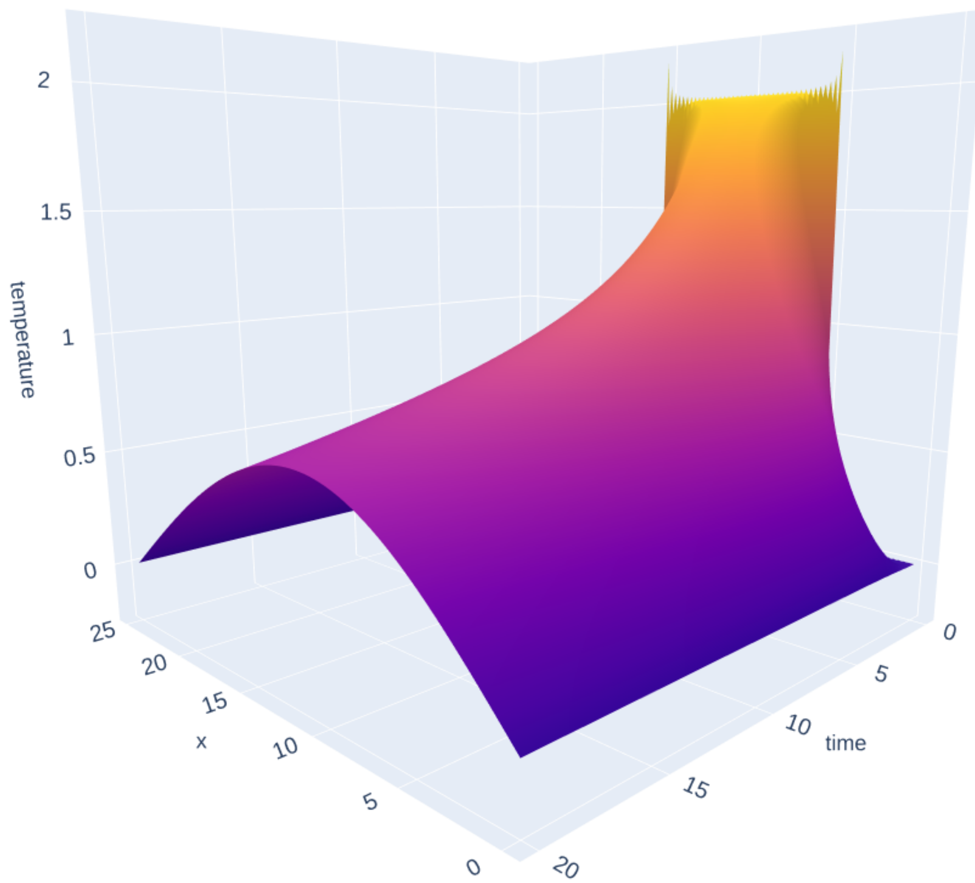

Figure 8: Semi exact solution of the heat equation. The exact solution given in equations 9 and 10 is approximated by a truncated Fourier-series. Note that the initial condition is not continuous, which results in a Gibbs phenomenon. The used initial and boundary conditions are:
$u(x,0) = \begin{cases} 2 & \text{if } 10 \leq x \leq 30 \\ 0 & \text{else} \end{cases}$, $u(0,t) = u(50,t) = 0$

4.3.1 Unweighted loss and uniform sampling

The method described in section 3.1 is used. 30 points were sampled from a uniform distribution on each boundary and in the interior region. The individual loss terms are not weighted; they are all equally important. A learning rate of 1, 50 mini-batches per epoch and a 100 samples per mini batch were used as the configuration for the neural network. The $\tanh x$ activation function was used. This configuration was found to perform relatively good after experimentation with other configurations. The results are plotted in figure 9.

The result is not good. The method seems to satisfy the differential equation, but it has trouble satisfying the boundaries. To address this problem, a weighted loss function is constructed in section 4.3.2.

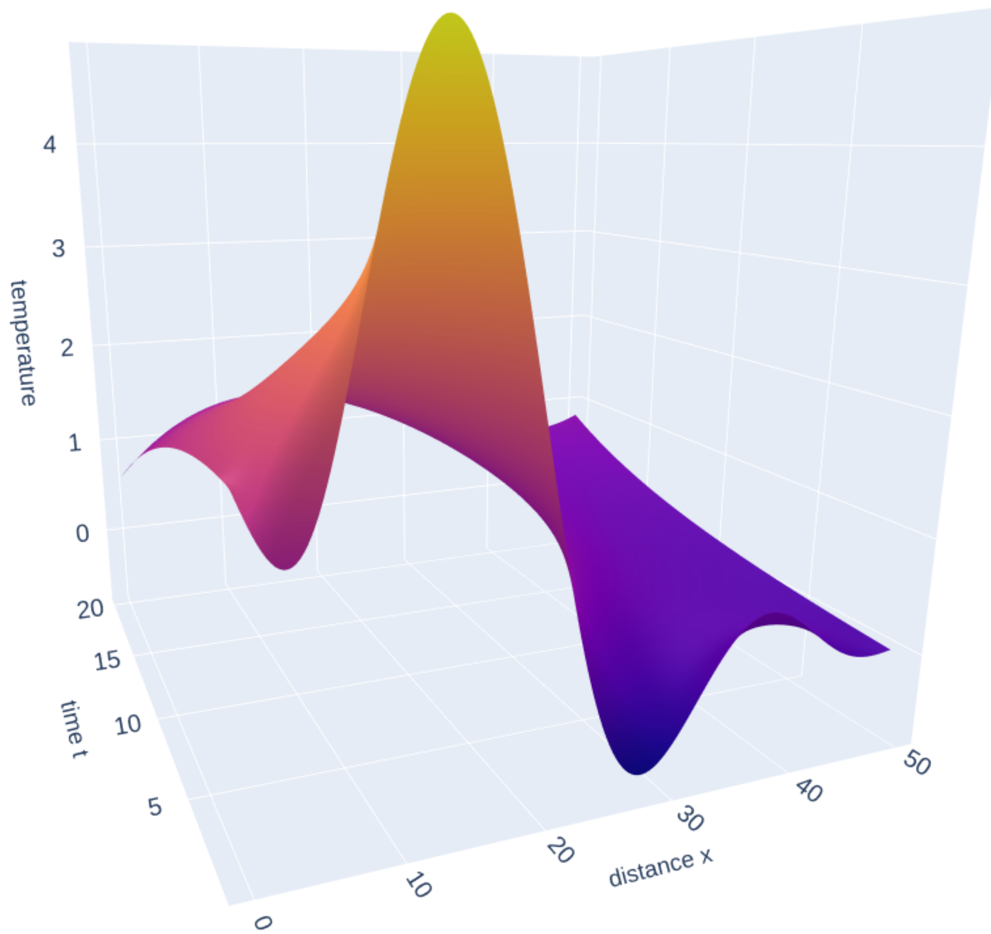

Figure 9: Approximation of the heat equation using the neural network method described in section 3.1. The used initial and boundary conditions are:
$u(x,0) = \begin{cases} 2 & \text{if } 10 \leq x \leq 30 \\ 0 & \text{else} \end{cases}$, $u(0,t) = u(L,t) = 0$

4.3.2 Weighted loss with fixed weights

The same problem was solved as in section 4.3.1, also with the same neural network configuration. This time, the loss function was weighted combination of the individual loss terms: $L = (L_{PDE} + 20 \cdot L_I + 10 \cdot L_{B1} + 10 \cdot L_{B_2})/41$, where L_{PDE} is the loss from the differential equation, L_{Bi} is the loss from boundary i and L_I is the loss from the initial condition. The results are shown in figure 10.

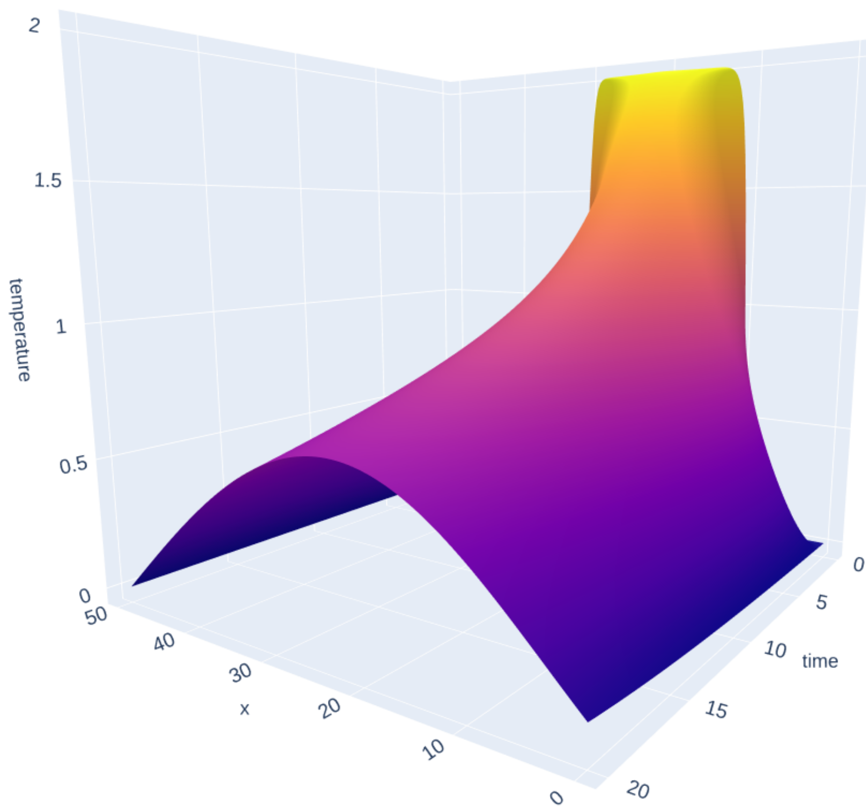

Figure 10: Neural network approximation of the heat equation, with a weighted loss function using fixed weights. The approximation agrees well with the exact solution, except at the discontinuities of the initial condition. The loss of the approximation is 1.004. The approximation at $t = 0$ is getting close to the initial condition. Note that the boundary at $x = 50$ is properly satisfied, but the boundary at $x = 0$ is not exactly satisfied. The used initial and boundary conditions are:
$u(x,0) = \begin{cases} 2 & \text{if } 10 \leq x \leq 30 \\ 0 & \text{else} \end{cases}$, $u(0,t) = u(50,t) = 0$

The final solution gets relatively close to the exact solution. The approximation does not agree well to the initial condition around the discontinuities. However, since the neural network is a continuously differentiable function, it was expected that the approximation did not satisfy the initial condition perfectly. In fact, the approximation goes almost exactly through the average of the left and right limit of the discontinuity. Therefore this approximation of the initial condition is seen as a fairly good result.

The absolute difference between the final approximation and the exact solution is shown in figure 11.

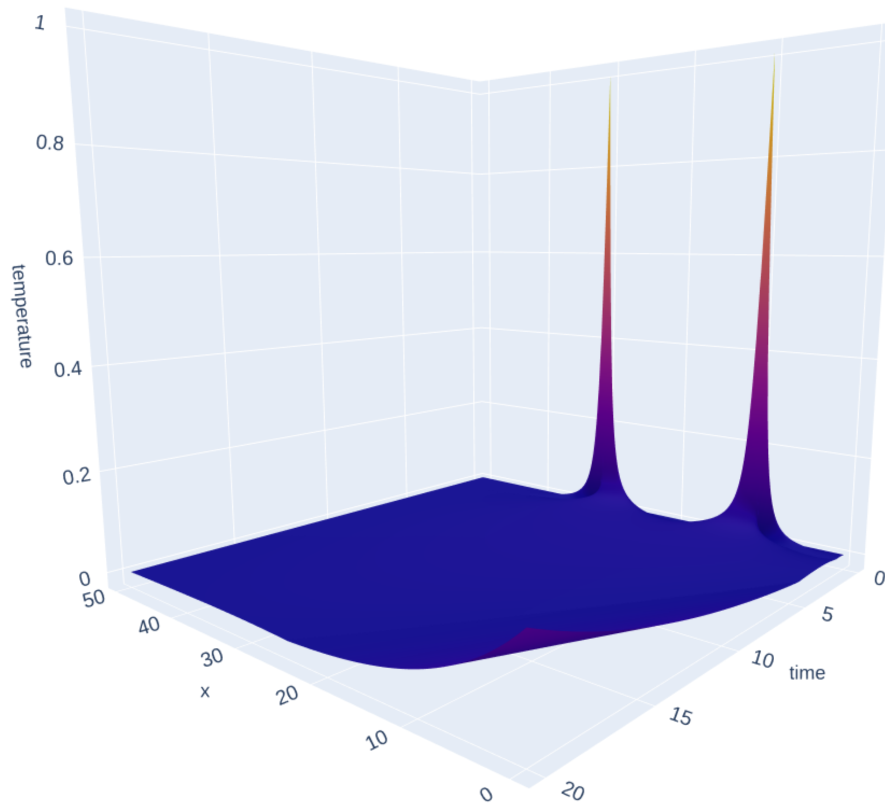

Figure 11: Absolute difference between the semi-exact solution shown in figure 8 and the final approximation made by the neural network shown in figure 10.

As stated earlier, it is expected that there is a high error in the approximation at the discontinuities of the initial condition. In figure 12, the difference between the exact solution and the approximation is shown for $t > 0.6$, which does not include these discontinuities and is therefore more interesting.

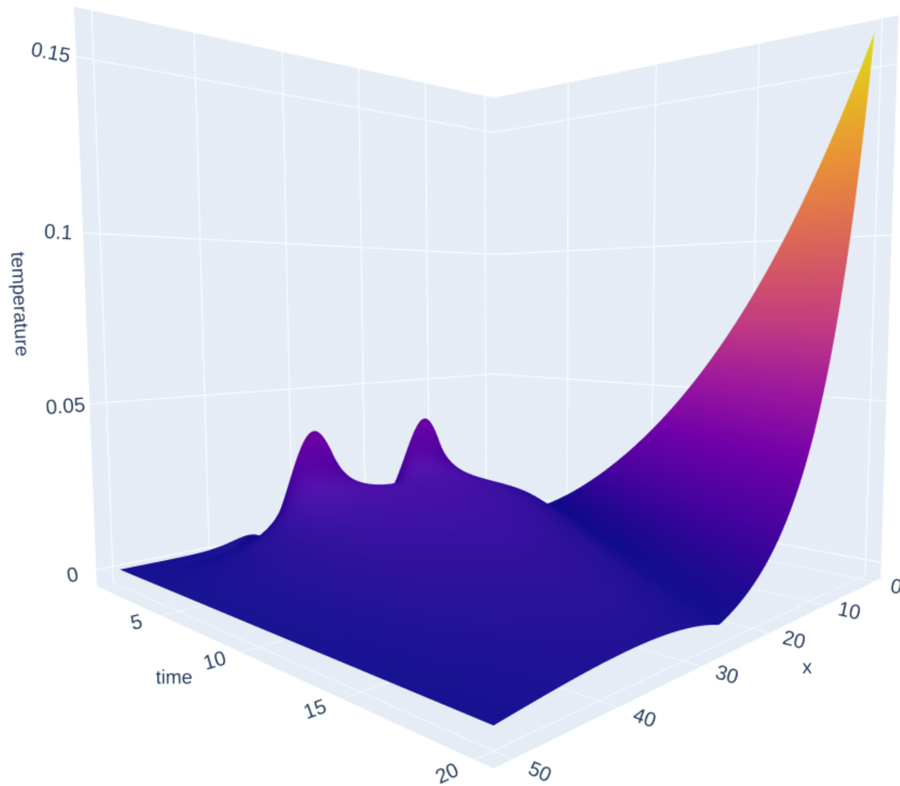

Figure 12: Absolute difference between the semi-exact solution shown in figure 8 and the final approximation made by the neural network shown in figure 10 for $t > 0.6$.

It seems like the neural network has sacrificed one of the boundaries for the initial condition. That makes sense, since the initial condition is twice as important as the boundary. The chosen weights do not seem to be right for fine-tuning the approximation. To address this problem, in section 4.3.3, the weights are chosen as a function of the loss.

4.3.3 Weighted loss with loss-dependent weights

The same problem was solved as in section 4.3.2, with the same neural network configuration as well. This time, weights of the terms in the loss function are chosen as a function of the loss itself: $w_i = \frac{L_i}{\sum_i L_i}$, where i can be a boundary loss, the initial loss or the PDE loss. By using these weights, the loss term with the highest loss becomes more important than terms with lower loss. The results are shown in figure 13.

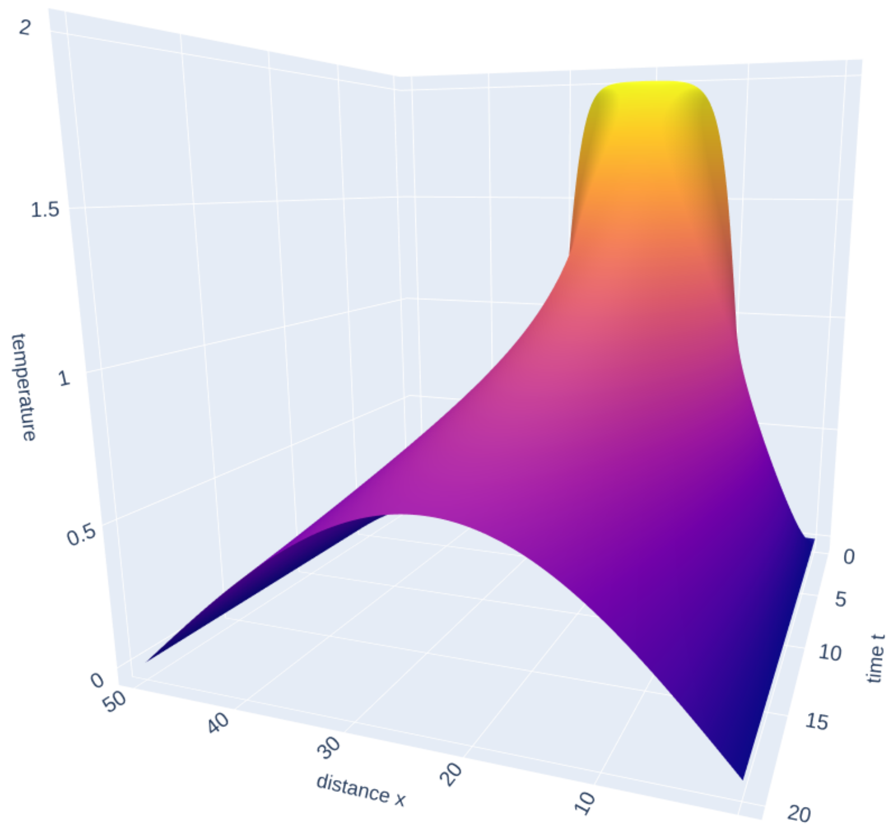

Figure 13: Neural network approximation of the heat equation, with a weighted loss function using loss-dependent weights. The approximation agrees well with the exact solution, except at the discontinuities of the initial condition. The loss of the approximation is 1.203. The approximation at $t = 0$ is getting close to the initial condition. The used initial and boundary conditions are: $u(x,0) = \begin{cases} 2 & \text{if } 10 \leq x \leq 30 \\ 0 & \text{else} \end{cases}$, $u(0,t) = u(50,t) = 0$

This time, both boundaries are satisfied. The initial condition is however less well-fulfilled than in section 4.3.2. Also note that up until now, all the methods did not use a mesh.

For this method, another experiment was performed using different initial conditions to see if the method would perform better using a continuous initial condition. The used initial condition is $u(x,0) = \sin\left(\frac{\pi x}{L}\right)$. The result can be seen in figure 14.

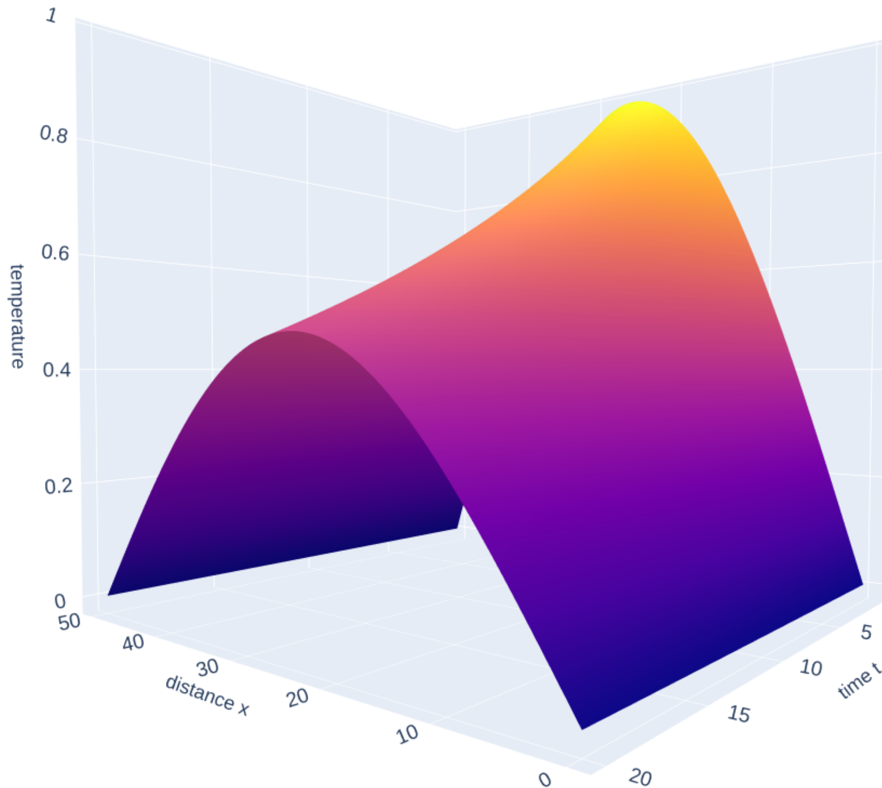

Figure 14: Neural network approximation of the heat equation, with a weighted loss function using loss-dependent weights. The approximation agrees well with the exact solution. The loss of the approximation is 0.385. The used initial and boundary conditions are: $u(x,0) = \sin(\frac{\pi x}{50})$, $u(0,t) = u(50,t) = 0$

The neural network almost perfectly approximates the analytical solution. Note that the loss is not zero, so it is not an analytic solution (as expected).

4.3.4 Weighted loss and loss-dependent sample distribution

In all previous attempts to solve the heat equation, a uniform distribution was used to sample the random input points from, as described in section 3.1. This time, a distribution is used that is dependent on the loss. A detailed description is provided in section A.2 and a summary is provided in section 3.3.

A small equidistant mesh is created of 250 points (10 points in the time dimension, 25 points in the space dimension). At these points, the PDE loss is calculated. If the point happens be on a boundary, then the boundary loss is also calculated. These losses are then transformed to probability distributions such that points with a high loss get a high probability of being sampled. by sampling the random points from these newly created distributions, the neural network focuses more on points that have a high approximation error. The results are shown in figure 15.

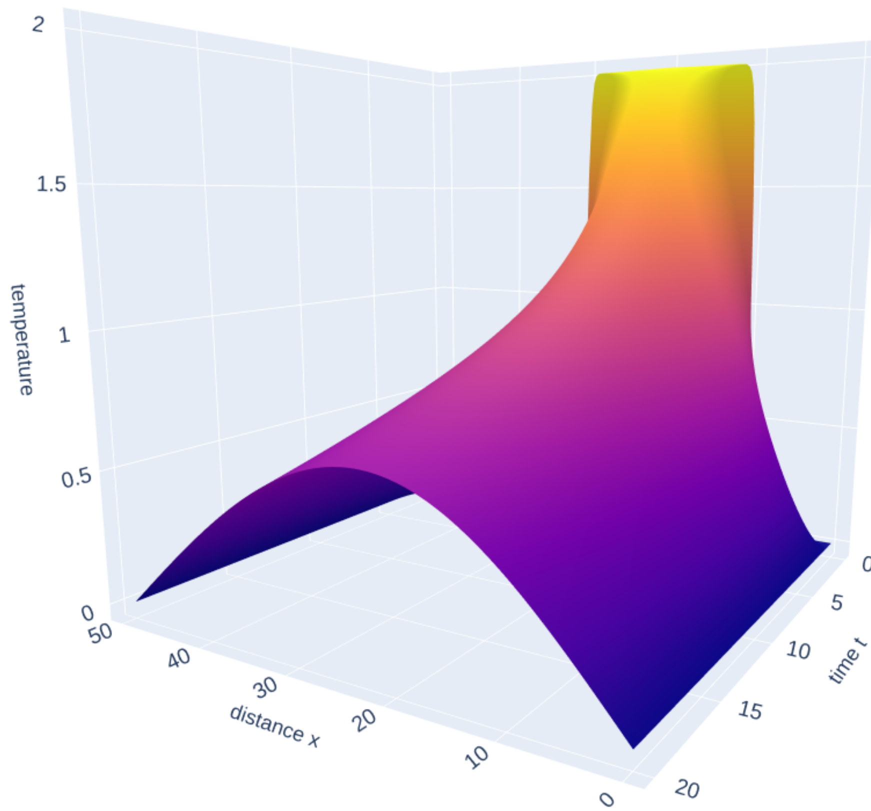

Figure 15: Neural network approximation of the heat equation, with a weighted loss function using loss-dependent weights and random points sampled from a custom probability distributions. The approximation agrees well with the exact solution, except at the discontinuities of the initial condition. The loss of the approximation is 0.264. The used initial and boundary conditions are: $u(x,0) = \begin{cases} 2 & \text{if } 10 \leq x \leq 30 \\ 0 & \text{else} \end{cases}$, $u(0,t) = u(50,t) = 0$

This method performs very well. The initial condition is almost perfectly satisfied, while the boundary conditions and the differential equation are satisfied as well.

In figure 16 is the absolute difference between the analytical solution and the approximation shown. Again the region $0 \leq t \leq 0.6$ is omitted since the discontinuities in the initial condition are not expected to be satisfied.

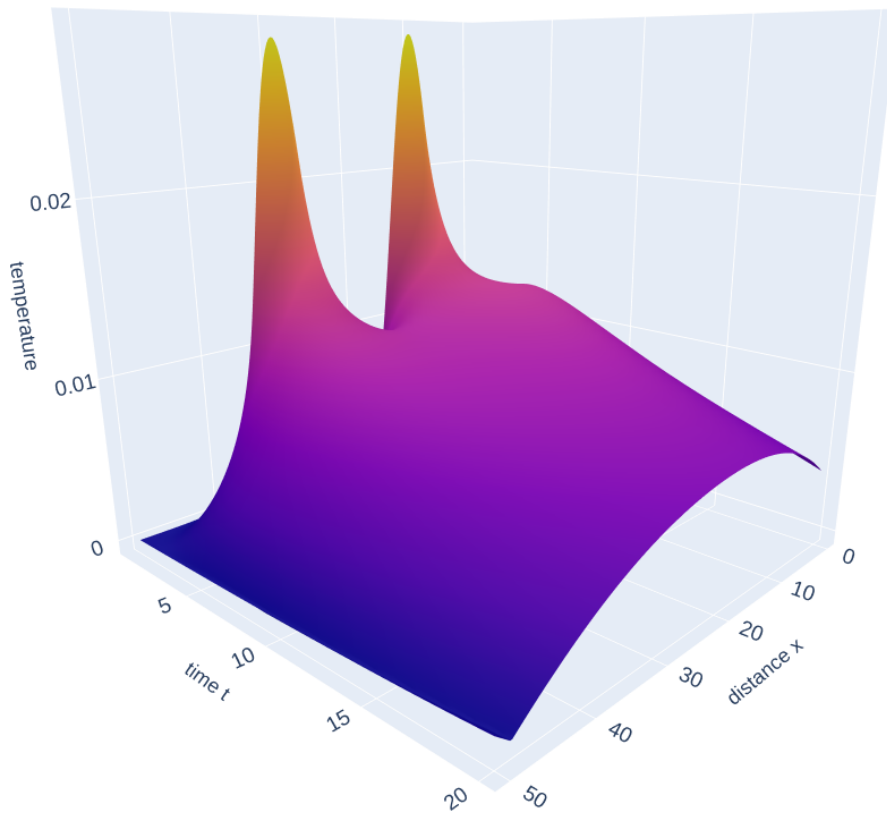

Figure 16: Difference between the semi-exact solution shown in figure 8 and the final approximation made by the neural network shown in figure 15 for $t > 0.6$.

4.3.5 Discussion on the results of the heat equation

The output of a neural network used here is always continuous, since all the layers discussed in section 2 are continuous. By supplying discontinuities in the initial condition, the neural network was therefore given an impossible task. However, these kinds of initial conditions are often used when solving the heat equation, which makes it interesting to see whether the neural network is able to deal with these kinds of initial conditions. In figure 15, a promising result is shown, wherein all required boundary and initial conditions were satisfied.

4.4 Solving the Navier-Stokes and Continuity equation

Solving the Navier-Stokes equations by the described method turned out to be a challenging task. After a lot of experimentation, a very simple model is made of an ideal gas through a pipe. The approximation failed to converge in almost all the experiments. Only in one case reasonable values were obtained. This case is discussed in this section. The approximation made by the neural network for this case seems to agree with common sense, although no verification is made yet.

The Navier-Stokes equations are given by equation 11.

$$\rho \frac{\partial \boldsymbol{v}}{\partial t} + \rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} = \mu \nabla^2 \boldsymbol{v} - \frac{RT}{M} \nabla \rho \tag{11}$$

In this equation, $\boldsymbol{v}$ is the velocity of the flow vector, t is time, ρ is the density of the flow, μ is the dynamic viscosity, T is the temperature of the flow (assumed at $T = 293K$), R is the ideal gas constant ($R = 8.3145 \frac{J}{K \cdot mole}$) and M is the molar mass of the gas. Air is assumed to have

an average molar mass of $M = 0.0289647 \frac{kg}{mole}$. The dynamic viscosity of air is assumed to be $\mu = 18.12 \cdot 10^{-6} \frac{Ns}{m^2}$. Note that gravity has been neglected.

Since mass is a conserved quantity, the continuity equation 12 must be solved at the same time:

$$\frac{\partial \rho}{\partial t} = -\frac{\partial}{\partial x}(\rho v_x) - \frac{\partial}{\partial y}(\rho v_y) - \frac{\partial}{\partial z}(\rho v_z) \tag{12}$$

The gas is assumed to have constant temperature and constant concentration, so no heat equation and concentration equation is solved (although this is believed to be possible using the neural network method).

Usually, a pressure gradient is present in the Navier-Stokes equations. This has however been replaced by a density gradient using the ideal gas law, modified to offer a pressure gradient, given in equation 13.

$$\nabla p = \frac{RT}{M} \nabla \rho \tag{13}$$

Where p is the pressure in $\frac{N}{m^2}$. Note that R, T and M are assumed to be constant in time and space. By using this equation, we assume that the described flow obeys the ideal gas law.

The problem is solved for a flow through a square pipe. The entrance and exit of the pipe are the planes $x = 0$ and $x = L_x$ respectively. The boundary conditions are given in equations 14 until 20.

$$\boldsymbol{v} = \mathbf{0} \text{ on the walls of the pipe} \tag{14}$$

$$\nabla \rho = 0 \text{ on the walls of the pipe} \tag{15}$$

$$v_x = 2 \sin(y\pi/L_y) \sin(z\pi/L_z)(1 - e^{t \cdot \ln 20}) \text{ at the entrance of the pipe} \tag{16}$$

$$v_y = v_z = 0 \text{ at the entrance of the pipe} \tag{17}$$

$$\rho = 1.2047 \text{ at the entrance of the pipe} \tag{18}$$

$$\nabla \rho = 0 \text{ at the entrance of the pipe} \tag{19}$$

$$\frac{\partial \boldsymbol{v}}{\partial x} = \mathbf{0} \text{ At the entrance of the pipe} \tag{20}$$

Please note that a more realistic scenario would entail a gas flow forced by a pressure gradient, meaning that the pressure on one side of the pipe is higher than the pressure on the other side of the pipe. Multiple experiments were done with this scenario, all of which resulted in the approximations failing to converge. These scenarios are therefore not further elaborated on.

As seen in equation 20, $\boldsymbol{v} = 0$ at the walls of the pipe, which should also be true near the entrance of the pipe. Therefore v_x is chosen in such a way that $v_x = 0$ at $y = 0$, $y = L_y$, $z = 0$ and $z = L_z$. Experiments were performed where this condition was not met, but the neural network was not able to give a good approximation in all attempts.

An attempt is made to solve the Navier-Stokes equations with the described neural network method and the result is shown in figure 17, where one time step is shown. When judging qualitatively, it seems as if this is a success. One should be however careful of course, since the boundary condition at the entrance of the pipe highly suggests this solution. The job of the neural network is therefore relatively simple and the success of this experiment cannot be generalized to more realistic boundary conditions. Furthermore, no quantitative judge is made on this experiment. The reason for this decision becomes clear in section 4.5.

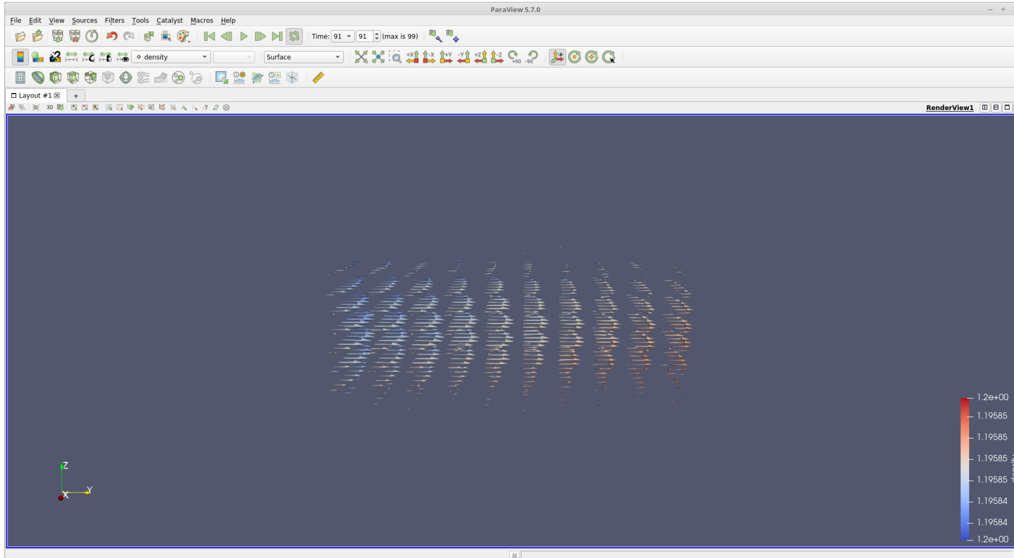

Figure 17: A result at time step t = 99. Note that the walls of the pipe are not drawn. This is the first successful attempt to solve the Navier-Stokes equations with the neural network method

4.4.1 Implementation in PyTorch

Implementation of these equations is in concept the same as implementing the heat equation. PyTorch is however not build for such usage, which causes a lot of unexpected problems and bugs in PyTorch and makes implementation relatively difficult.

The loss is constructed by summing up all the individual losses. For each of the differential equations, the loss is calculated and summed. For each boundary, the loss is calculated and added to the PDE loss. The resultant total loss is given to the optimization algorithm.

The used neural network is a 5 layer convolutional neural network with the hyperbolic tangent as activation function. The optimization algorithm used is the Adam algorithm, which comes with the PyTorch library.

Usage of optimization algorithms that are based on gradient descent comes with the drawback that they do not converge to the global minimum of a function, but rather to a local minimum. In this case, one of the most obvious local minima is the solution where the flow is zero everywhere, and density is equal to the in-flowing density everywhere. The differential equations are satisfied everywhere, and most of the boundaries as well. The only contribution to the loss is therefore the boundary loss of the in-flowing boundary, which is usually smaller than the situation where the PDE is not satisfied correctly and multiple boundaries are not properly satisfied. However, this local minimum is not the desired output of the neural network and some tricks are used to circumvent this problem.

One of the advantages of PyTorch is the ability to modify the loss function during training. In the beginning of the training process, the optimization algorithm only optimizes the boundary loss and thereby ignores the satisfaction of the differential equation. When the model converges, it should satisfy the boundary conditions relatively good.

Another trick is to leave out some boundaries when constructing the loss function. Since most of the boundaries have a constant value, they are easily approximated. Other boundaries, on the other hand, are difficult to model by the neural network. The optimization algorithm should therefore optimize more often on difficult boundaries.

The difficulty of a boundary is already measured by the boundary loss, which is calculated during the creation of the probability distribution of the boundary. In order to solve more difficult boundaries more often, boundaries are sampled with a probability proportional to their loss.

After the boundaries are satisfied, the differential equation loss is turned back on again. The model hopefully does not lose the satisfaction of the boundary conditions when it solves the PDE loss.

4.4.2 Convergence

For the sampling probability estimation, a mesh is used of $10 \times 10 \times 10 \times 10$ points, where $0 \leq x \leq 1$, $0 \leq y \leq 2$, $0 \leq z \leq 1$ and $0 \leq t \leq 1$. The PDE loss is constructed over 10 sampled points and each boundary loss is also constructed over 10 sampled points before a parameter optimization step is performed. After 50 of such optimizations, the sampling probability estimation is recalculated.

Without applying the tricks described in section 4.4.1 the model converged to the local minimum where $\boldsymbol{v} = \mathbf{0}$ and constant density. Therefore the tricks described above seems to be necessary to get a converging solution. As stated earlier, every other experiment done on these equations failed to converge. Event changing the boundary conditions made the method diverge. It is therefore worth doing more research on the convergence criteria for neural networks.

4.5 Mass-spring system 2

In order to test the implementation of the Navier-Stokes solver, the script is adjusted to solve a mass-spring system. Such a system is much easier to visualize, which makes is easier to tell if the results of the script make sense.

The implementation is very similar to the Navier-Stokes implementation; the Navier-Stokes equations are replaced by the mass-spring differential equation and two initial conditions are made. For the model, initial conditions and boundary conditions are handled in the same way and are collectively called boundaries, since they represent the state of the system at the boundary of the region where the differential equation is solved. Multiple experiments are done with different parameters, but all experiments resulted in the same disappointing behaviour. Only one experiment is described here as an example and a qualitative judgement is made on it. Since all other experiments resulted in the same behaviour, this judgement is generalized and a recommendation for further research is made.

4.5.1 Solving the mass spring equation and results

The used differential equation is $(\frac{d^2u}{dt^2}(t) + ku(t) = 0,\ k = \frac{1}{9})$ and two boundaries are being made $(u(0) = 5$ and $\frac{du}{dt}(0) = 0)$. The boundary sampler is turned off since there are only two boundaries. A learning rate of 10^{-3} is used. The analytical solution is given by $u(t) = 5\cos(\frac{1}{3}t)$, which is plotted together with the neural network approximation in figures 18, 19 and 20.

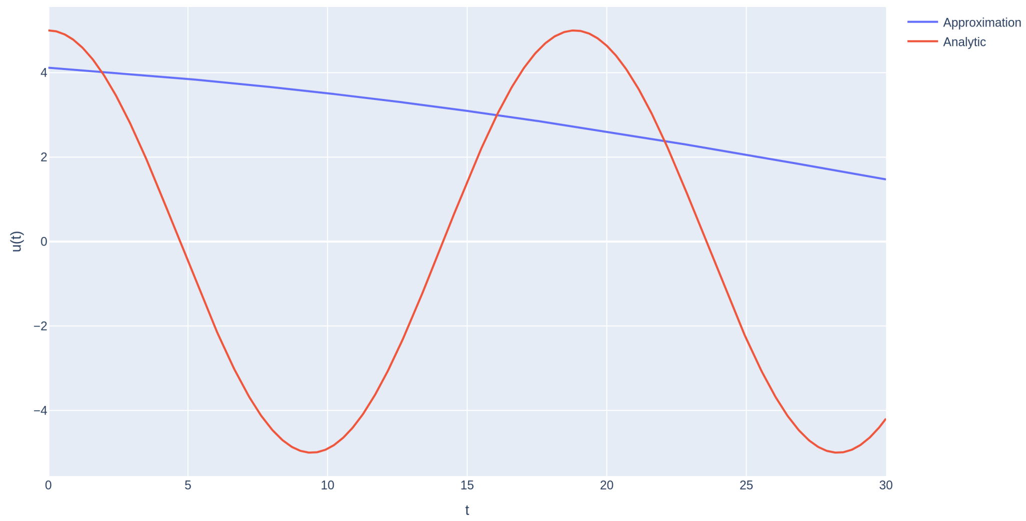

Figure 18: Neural network approximation of the mass-spring system after 5 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. The loss of the approximation is 21.21.

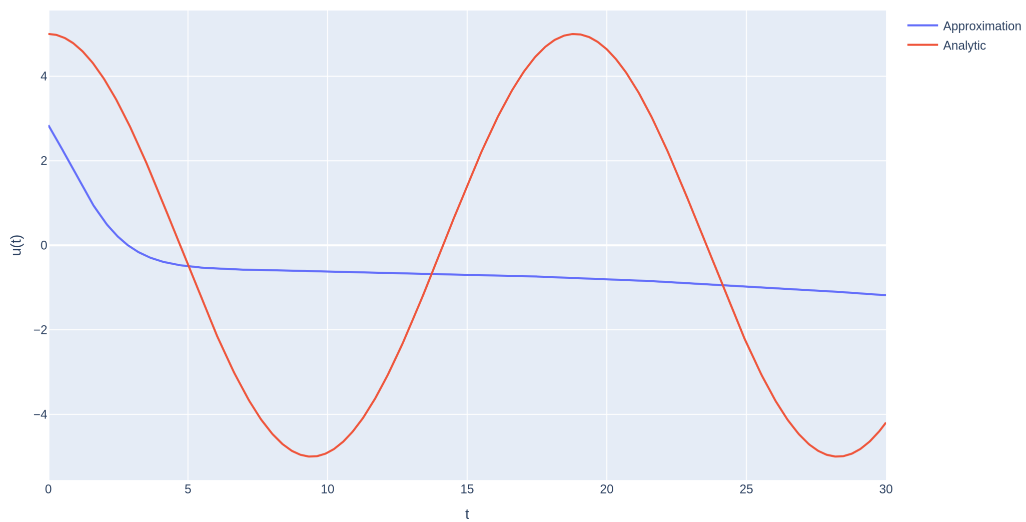

Figure 19: Neural network approximation of the mass-spring system after 20 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. The loss of the approximation is 13.82.

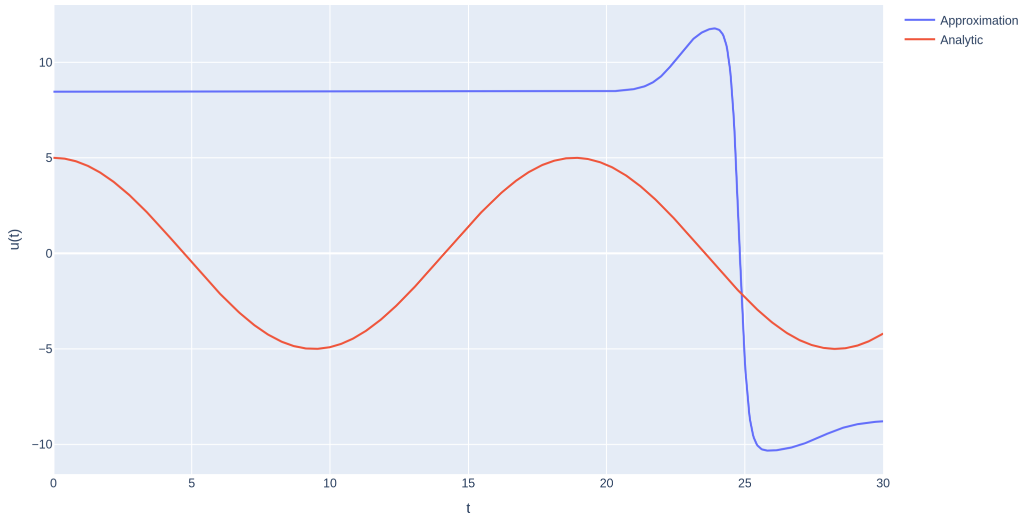

Figure 20: Neural network approximation of the mass-spring system after 40 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. The loss of the approximation is 56804.9.

Of course, the high loss and the odd approximation at epoch 40 immediately stand out. From epoch 28 to 37, the order of the loss increased from 10^1 to 10^4. After epoch 40, the approximation seems to become random. it sometimes stabilizes at a weird "spiky" solution (see, for instance figure 20), whereas at other times, it cycles to multiple bad approximations and sometimes even explodes to extremely large numbers until they overflow the computer memory. Overall, the approximation seems to diverge. This weird behaviour is also happening when other optimization algorithms are used, such as stochastic gradient, and when the convolutional layers are replaced by fully connected layers.

Since the same script is used to solve the Navier-Stokes equations, the positive result in that experiment loses its value as that method does not work here. No quantitative analysis is therefore made on results of the Navier-Stokes solver.

4.5.2 Detailed problem determination

An attempt has been made to determine the source of these problems. Some interesting properties were found, which are outlined in this section. Firstly the sampler and the calculation of the loss have been checked, and they are both correct. The calculation of the derivatives is correct as well.

Remember that the neural network consists of five layers and uses the hyperbolic tangent as activation function. For $x > 3$, $\tanh(x) \approx 1$ and for $x < -3$, $\tanh(x) \approx -1$. Since the input of the neural network is a range, it is expected that the output is a range as well. For most neurons, the input of the tanh function should therefore be in the interval $[-3, 3]$, which means that the weights and biases of the network should be of order 10^{-1} or lower.

It is observed that at the beginning of the solving phase, this requirement is satisfied. The output is a continuous range of numbers. However, after approximately 40 epochs, the weights and biases start to grow and the output is almost always one of two numbers, regardless of the input. This phenomenon can be seen already in figure 20, where approximation is almost always one of 2 numbers except for the rough region around $x = 25$. It is observed that this spikey region can become much smaller after more epochs.

4.5.3 Proposed solutions and their results

An obvious solution would be to use a different activation function. Therefore, $f(x) = e^x$, $f(x) = \sin(x)$, $f(x) = x^3$ and $f(x) = \ln(e^x + 1)$ are also tried as activation functions. The exponential function always overflows the computer memory after approximately 100 epochs. Even after normalizing the input, scaling between layers, properly adjusting the learning rate, weights and biases initialization, this behaviour does not disappear. The exponential function is thus considered as an unsuitable activation function.

More success is achieved with the sine function, since the approximation does not diverge. The approximation does not however converge to the analytical solution even though it came very close. The results are shown in figures 21, 22, 23 and 24. Note that the learning rate and parameter initialization is properly adjusted.

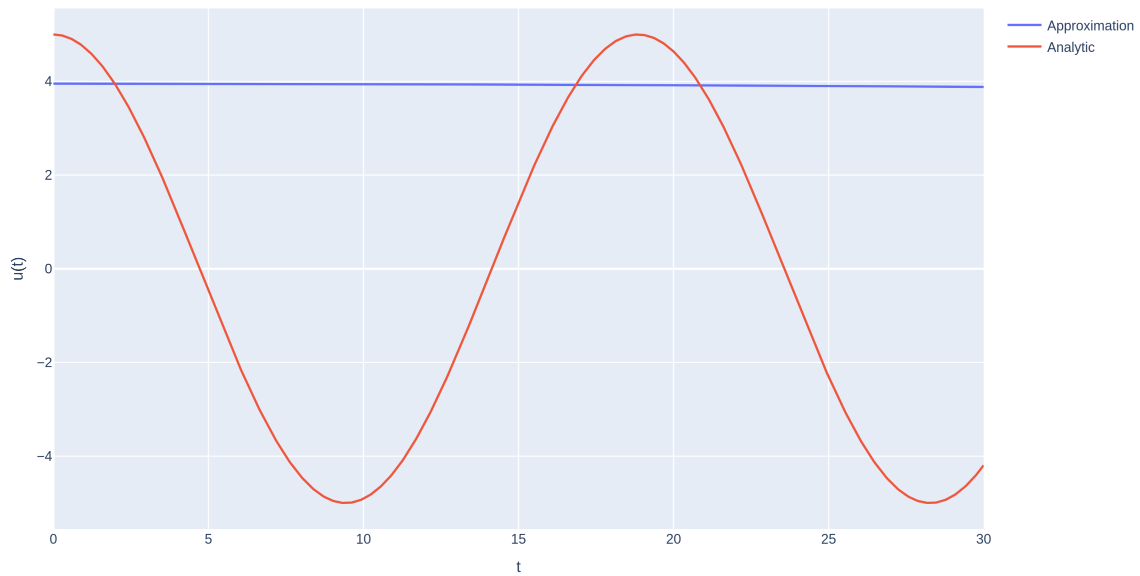

Figure 21: Neural network approximation of the mass-spring system after 15 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \sin(x)$ is used as activation function. The loss of the approximation is 24.60.

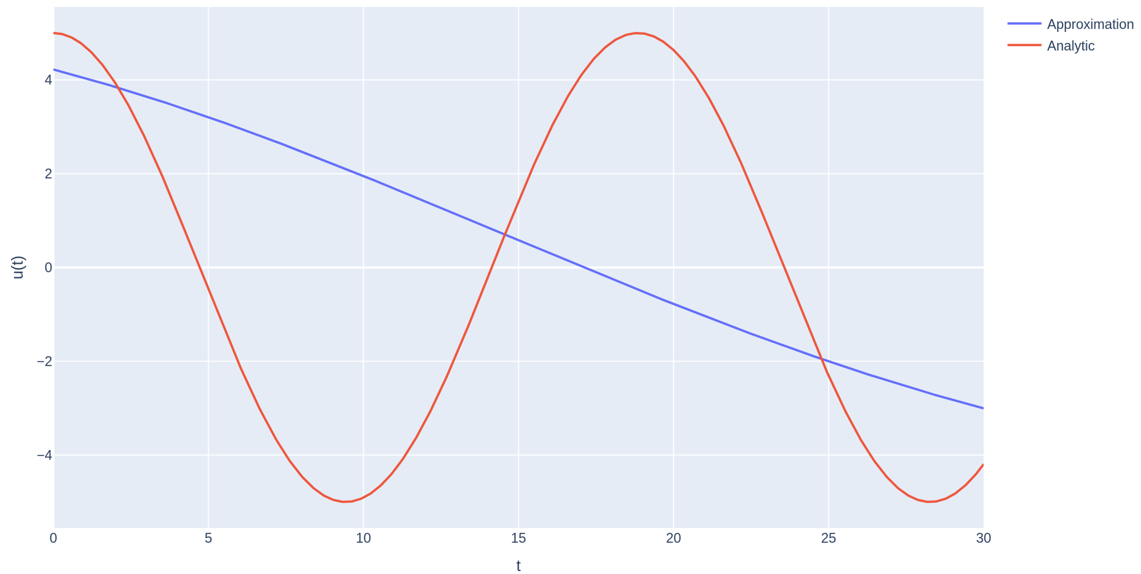

Figure 22: Neural network approximation of the mass-spring system after 50 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \sin(x)$ is used as activation function. The loss of the approximation is 13.86.

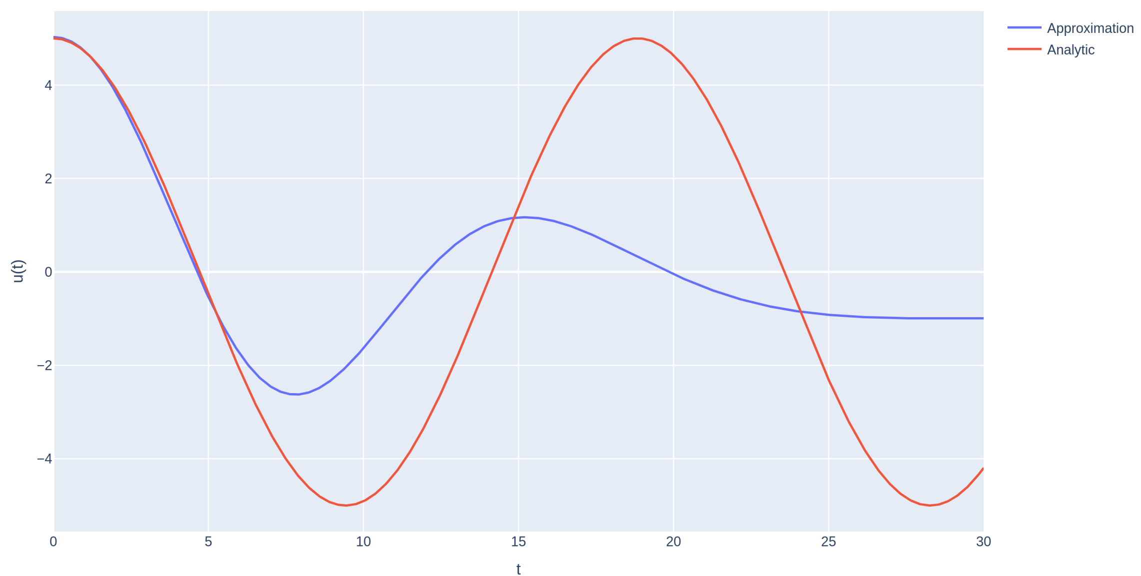

Figure 23: Neural network approximation of the mass-spring system after 135 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \sin(x)$ is used as activation function. The loss of the approximation is 3.719.

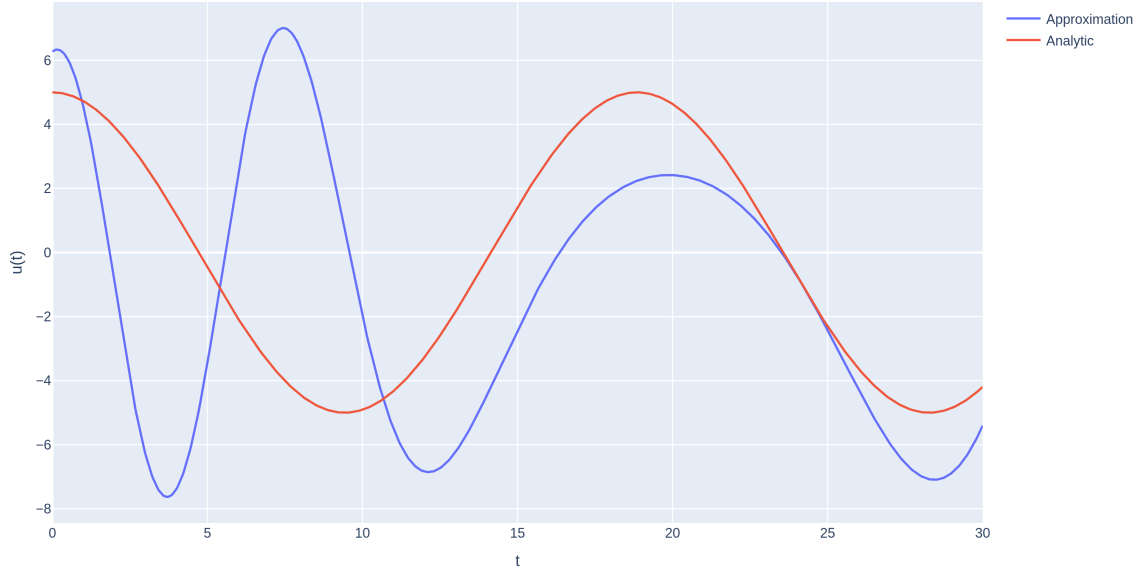

Figure 24: Neural network approximation of the mass-spring system after 280 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \sin(x)$ is used as activation function. The loss of the approximation is 783.3.

At the beginning of the training phase, the approximation seems to converge to the analytical solution and it achieves low losses. However, after approximately 150 epochs, the approximation gets worse and the loss rises. In contrast to the hyperbolic tangent activation function, the approximation remains a range and it does not diverge.

Thirdly, $f(x) = x^3$ is tried as an activation function. Note that this is the only bijective activation function which is tried with $\mathbb{R}$ as domain and range. This time, the result is again successful, since the approximation is not simply 2 numbers nor does it diverge. The results are shown in figures 25, 26, 27 and 28. Note that the learning rate and parameter initialization are properly adjusted.

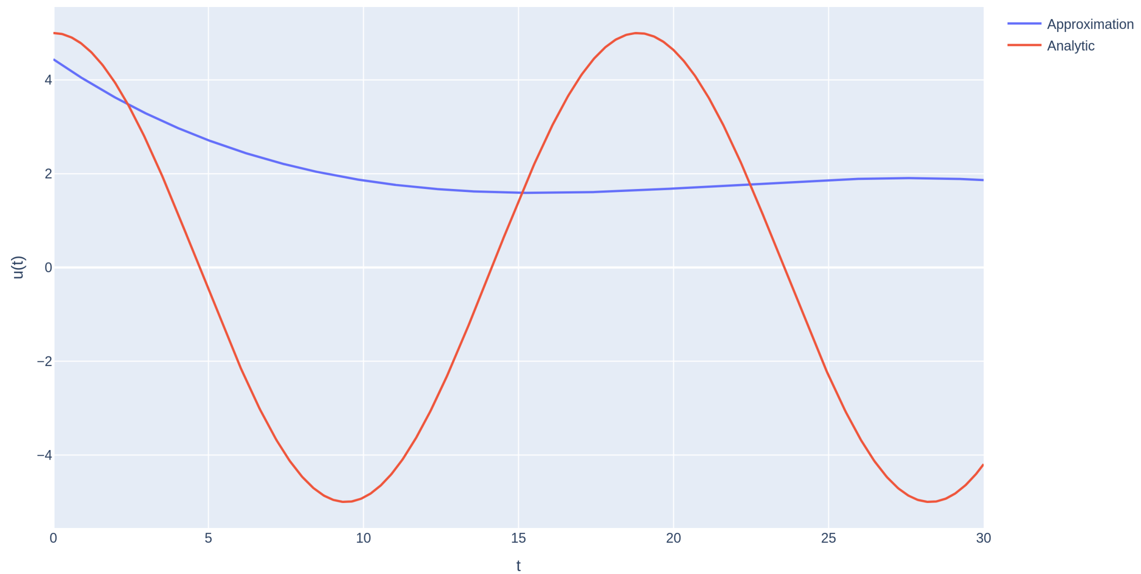

Figure 25: Neural network approximation of the mass-spring system after 5 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = x^3$ is used as activation function. The loss of the approximation is 14.52.

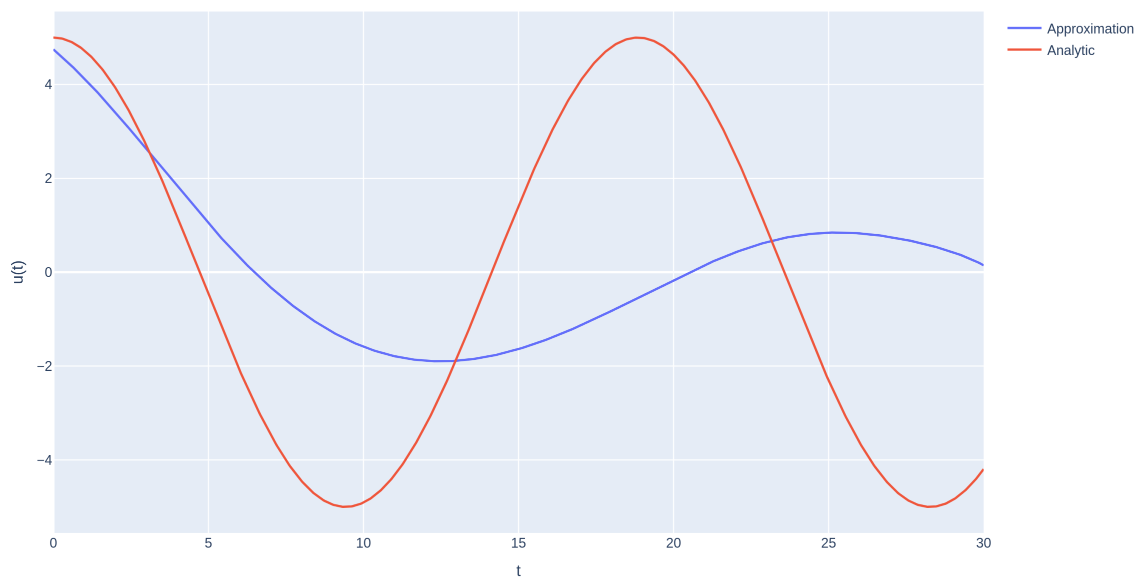

Figure 26: Neural network approximation of the mass-spring system after 180 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = x^3$ is used as activation function. The loss of the approximation is 8.695.

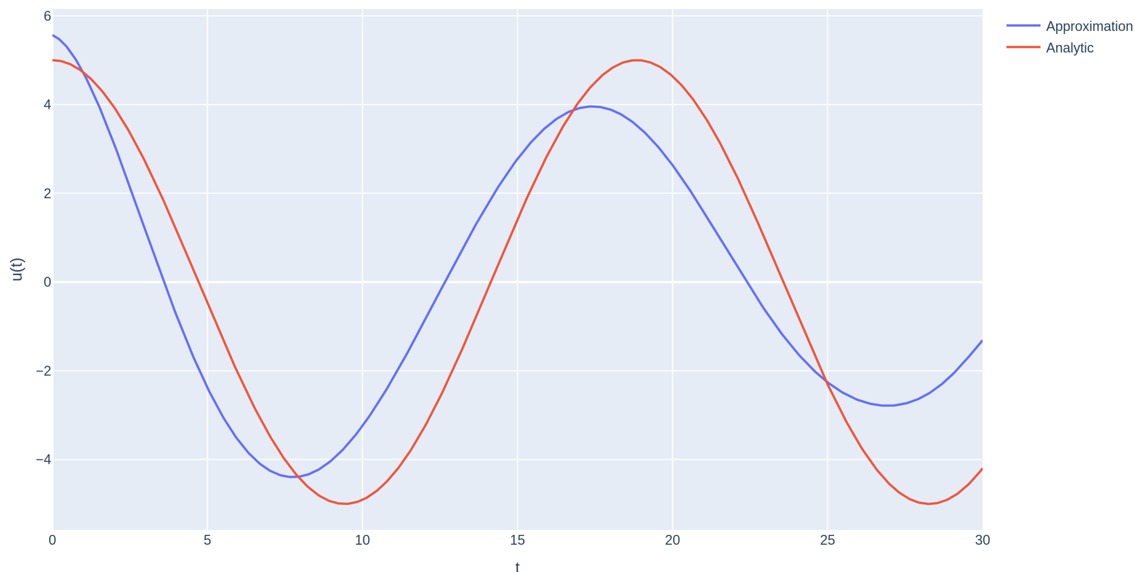

Figure 27: Neural network approximation of the mass-spring system after 455 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = x^3$ is used as activation function. The loss of the approximation is 14.87.

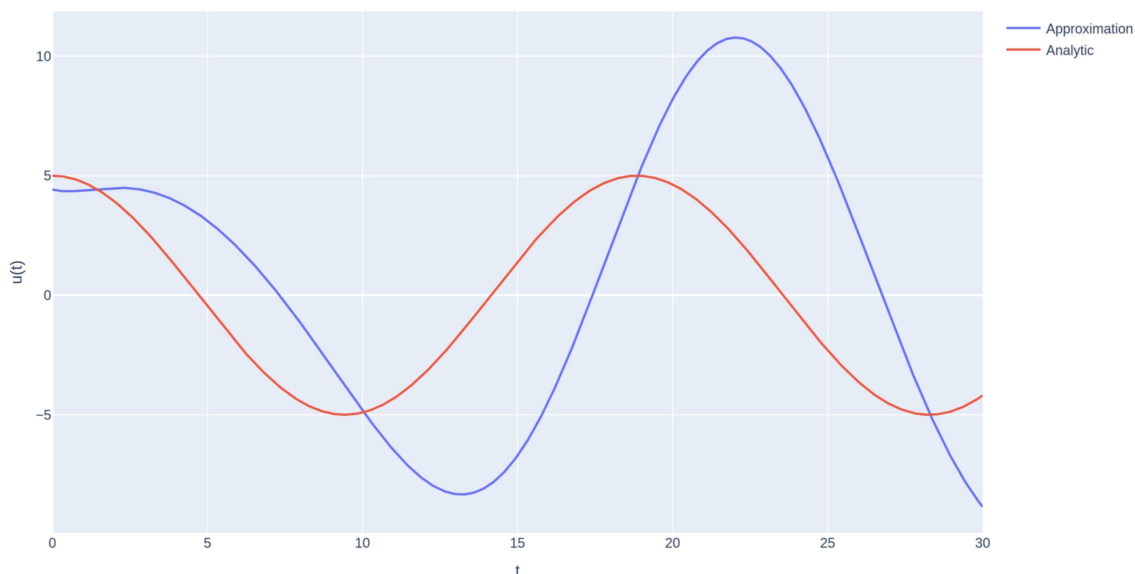

Figure 28: Neural network approximation of the mass-spring system after 805 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = x^3$ is used as activation function. The loss of the approximation is 34.40.

Here, we can see that this activation function does give a relatively good approximation in comparison to the other activation functions. However, the approximation is still not good at all. It is also interesting to see that more epochs do not necessarily make the approximation better.

Lastly, $f(x) = \ln(e^x + 1)$ is tried as activation function. Note that this function is very close to $f(x) = \max(0, x)$, which can be seen in figure 29. $f(x) = \max(0, x)$ achieved good results in image recognition, which makes it worth trying this activation function here [8]. It is however unsuitable here, since it is not continuously differentiable. Instead, $f(x) = \ln(e^x + 1)$ is used as replacement.

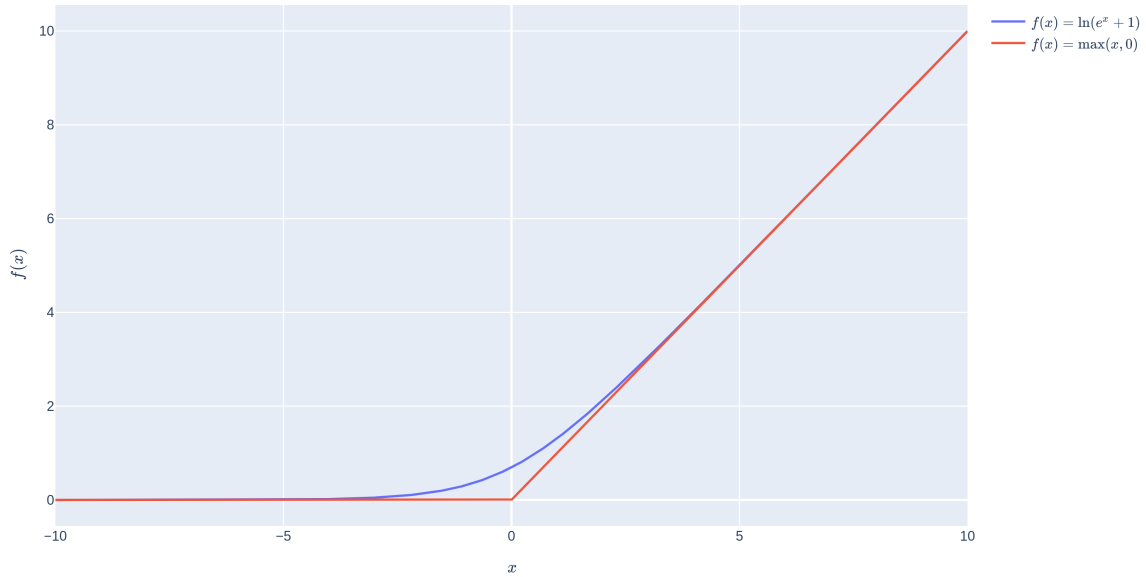

Figure 29: Comparison between the $f(x) = \ln(e^x + 1)$ function and the $f(x) = \max(0, x)$ function. Note that the former is continuously differentiable whereas the latter is not.

The results of this activation function can be seen in figures 30, 31, 32 and 33. Note that the learning rate is properly adjusted.

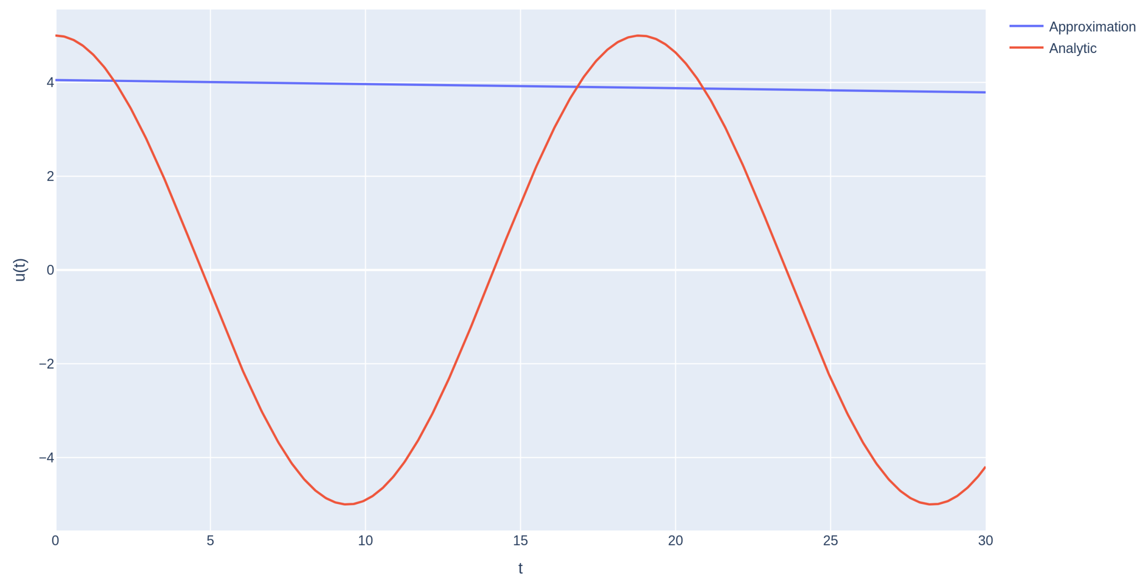

Figure 30: Neural network approximation of the mass-spring system after 30 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \ln(e^x + 1)$ is used as activation function. The loss of the approximation is 23.57.

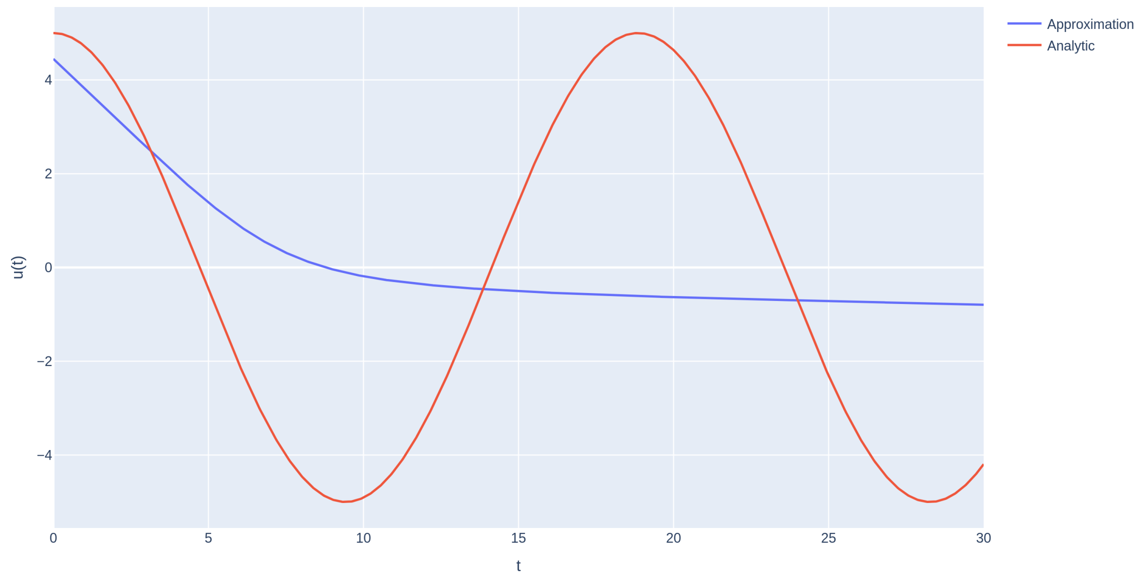

Figure 31: Neural network approximation of the mass-spring system after 280 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \ln(e^x + 1)$ is used as activation function. The loss of the approximation is 12.89.

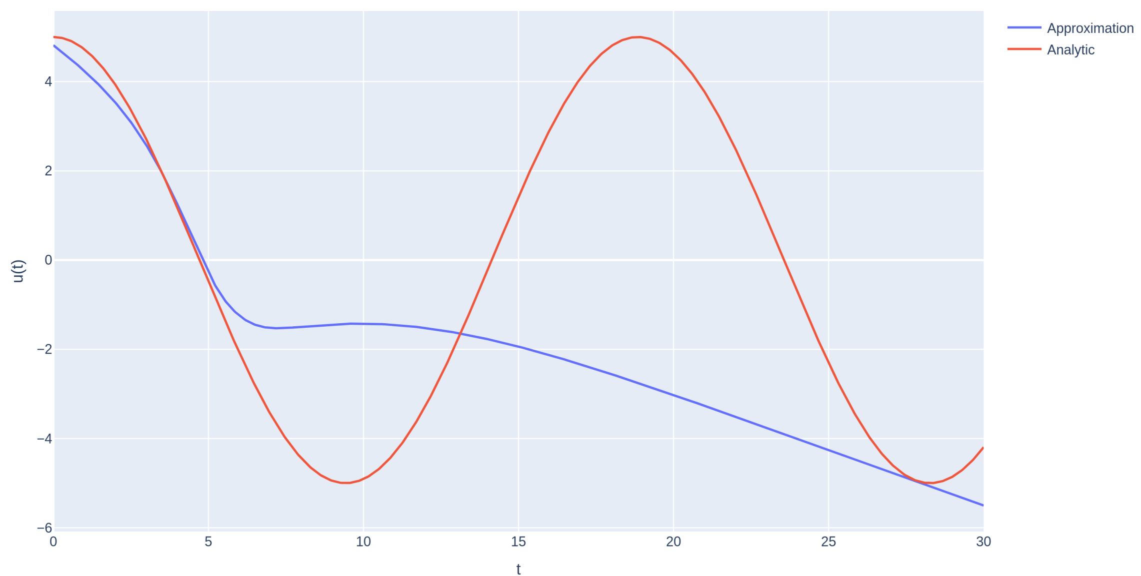

Figure 32: Neural network approximation of the mass-spring system after 565 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \ln(e^x + 1)$ is used as activation function. The loss of the approximation is 16.48.

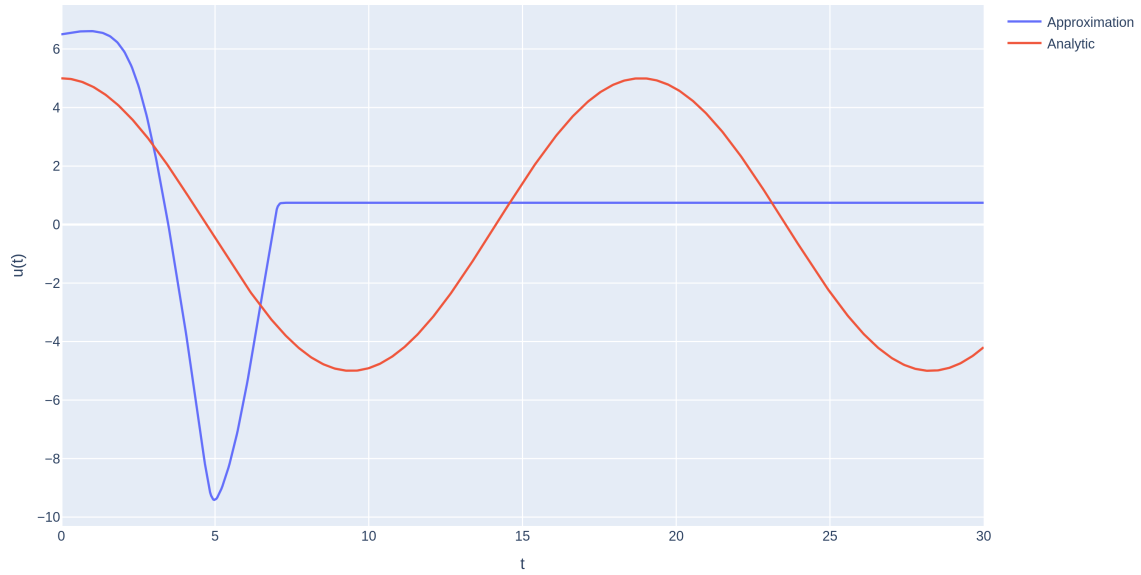

Figure 33: Neural network approximation of the mass-spring system after 640 epochs using the Navier-Stokes neural network solver together with the analytic solution, which is given by $u(t) = 5\cos(\frac{1}{3}t)$. Here, $f(x) = \ln(e^x + 1)$ is used as activation function. The loss of the approximation is $1.217 \cdot 10^4$.

The results of this activation are not better than the other tried activation functions. Note that this activation function always overflows the computer memory after approximately 700 epochs. Before the overflow, it does not seem to get a notion of the desired solution. This function is therefore considered an unsuitable activation function.

In general, the chosen activation function changes the behaviour of the neural network significantly, as expected. There is however not a single case wherein the approximation converged to the analytic solution, even if some approximations came very close. It thus looks as if the posed problem cannot be solved by choosing another activation function.

5 Conclusions from the experiments and further research recommendations

It becomes clear from the experiments that using a neural network as a differential equation solver has some potential and is worth researching. Some success was achieved in solving the mass-spring equation and the heat equation but every experiment lead to more questions than answers. Most of these questions need to be answered before the method can be of any practical usage.

When solving the mass-spring equation it was seen that different implementations give different results, while they should give the same answers. This hints to some inconsistency in the used software package PyTorch and needs more investigation. It was also found that the method did not converge for large initial conditions ($u(0) = 30$ was tried) or for large values of the spring constant ($c = 300$ was tried). From the analytical solution, the frequency is given by $\omega = \sqrt{c}$. Therefore, the neural network is having problems with larger frequencies. At this moment it is not known if there is a limit on the frequency and initial condition or that these problems were merely a result of a bad neural network configuration. It is recommended to search for this limit and why this limit is there, as it will place a condition on the usability of this method.

The neural network is able to solve the heat equation, even if it is provided with a discontinuous initial condition. The method needs to be configured properly and some adjustments need to be made to the loss calculation and sampling method. The questions which arose for the mass-spring equation also arise here. Is there a limit on the maximum value of the initial condition?

It is shown that the performance of a neural network is highly dependent on the chosen configuration. For a given neural network, a suitable optimization algorithm, learning rate, parameter initialization, batch size, epoch size and other options must be found. Since this configuration is dependent on the problem, a given configuration might work for some differential equations, but not on other differential equations. Moreover, changing the boundary conditions of a differential equation might already change the performance of the algorithm.

Also note that the algorithm depends on randomness, so when the experiments are repeated, the results might be totally different. This makes research difficult, since it makes it impossible to judge the algorithm based on a few runs.

In image recognition, it was found that the performance of the neural network was highly dependent on the configuration of the neural network [8]. The usage of neural networks have really gained attention at the moment when more sophisticated architectures, such as ResNet [5], were used. Since there are countless possible configurations, computer science has mainly focused on finding this configurations in the last years.

As a starting point to address the found problems, It is suggested to try different neural network configurations, especially the configurations which perform well for other fields of computer science.

Also note that the role of the optimization algorithm is very important. Since Adam and related methods always performed well in other fields of computer science, this method was used here as well. However, it is expected that other methods will perform better since the loss function to optimize behaves totally different in these computer science fields. Developing a new optimization algorithm designed to solve differential equations might increase the performance of this method. Also note that this would solve another problem, since the boundary conditions of the differential equation could not be fixed when using the Adam optimization algorithm. Developing a new algorithm which supports the fixation of the boundary conditions would solve this.

In summary, the newly suggested method does have potential to approximate the solutions of differential equations, as it was able to approximate the solution of the mass-spring equation and the heat equation. It was not able to approximate the solution of the Navier-Stokes equations coupled with the continuity equation, but it is expected that this is due a wrong configuration of

the neural network and the usage of an optimization algorithm which was not designed for this purpose. When a new optimization algorithm is used together with a proper configuration, the neural network might be able to approximate the solution of the Navier-Stokes equation as well.

Appendices

A In depth description of PyTorch implementation

Since some unexpected results were encountered, an in-depth explanation of those experiments is required. There are a lot of subtle differences between the first and second solver of the mass-spring system. Personally, I did not expect that these differences would lead to the failure of the second method (as described in section 4.5), especially since I considered the second method to be the better one. The two mass-spring solvers are described in-depth here and references to the code are made. Core pieces of the code are included in this report as figures. The interested reader can read this section along with the code.

Firstly, there is an obvious difference between source of the two methods. The first method is developed as a first test case of the neural network differential equation solver method, while the second method is developed as a check for the Navier-Stokes solver. The second solver might therefore seem overkill for the mass-spring system, but it should still work.

A.1 Mass-spring system 1

In this section the method used in solving the mass-spring system is described extensively. This method is relatively simple in comparison to the second system. The system is meant to solve the equation $\frac{d^2u}{dt^2}(t) + k \cdot u(t) + c \cdot \frac{du}{dt} = 0$, which is the damped mass-spring differential equation. For the undamped system, the differential equation is not changed but c is simply set to zero. The method is programmed in a single file. The file also includes functionality for finding the right learning rate, which is not described here.

The method is implemented using Python 3.8, running on a GPU-enabled machine. The versions of the used libraries are: numpy - 1.18.4, torch - 1.5.0+cu101 and plotly - 4.8.1. The used IDE is the community version of PyCharm, version 2020.1.1

The starts with class definitions and function definitions. Here, a neural network is created with 7 layers, 300 neurons per layer and the $\tanh(x)$ function is set as activation function, as shown in figure 34.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.fc1 = nn.Linear(1, 300)
        self.fcf = nn.Linear(300, 1)
        self.fc3 = nn.Linear(300, 300)
        self.fc4 = nn.Linear(300, 300)
        self.fc5 = nn.Linear(300, 300)
        self.fc6 = nn.Linear(300, 300)
        self.fc7 = nn.Linear(300, 300)

        self.dropout = nn.Dropout(p=0.0)

    def forward(self, x):
        return self.tanhActivation(x)

    def tanhActivation(self, x):
        y = torch.tanh(self.fc1(x))
        y = torch.tanh(self.fc3(y))
        y = torch.tanh(self.fc4(y))
        y = torch.tanh(self.fc5(y))
        y = torch.tanh(self.fc6(y))
        y = torch.tanh(self.fc7(y))
        y = self.fcf(y)
        return y
```

Figure 34: Code used for the initialization of the neural network. The network consists of 7 layers, each with 300 neurons. The $\tanh(x)$ activation function is used.

After the training class definition, the execution of the file starts at the bottom, where the neural network is moved to the used device, which refers to the GPU on the machine used in this report. If no GPU is available, the CPU will be used. The initial conditions are set, k and c from the differential equation are set. In the experiments, $k = 0.1$ is used, where c differs from experiment to experiment. The loss is set to mean squared error loss, using the built-in function in PyTorch, which is in the module torch.nn.MSELoss(). A learning rate scheduler is set, but is only used for finding the correct learning rate. The learning rate scheduler used during solving is set later on. Lastly, the optimization algorithm is set and the training algorithm is started. The code is shown in figure 35.

```python
# Set device to GPU if available, otherwise CPU
device = torch.device('cuda:0')
print(device)

# Convert neural net to the new device
net.to(device)

# Set initial conditions and K value
IC = torch.tensor([1.0, 0.0], device=device)
k = torch.tensor([.1], device=device)
c = torch.sqrt(4*k)

# Set loss type
criterion = nn.MSELoss()

# Set LR scheduler
lrLambda = lrSchedulers.tanhActivationScheduler(10 ** -3, 10 ** -1, 500)

# Set optimizer
optimizer = optim.Adam(net.parameters(), lr=0.0005, betas=(0.9, 0.999))

# Initialize training routine
routines = trainingRoutines(epochs=5000, startEpoch=0, numberOfMiniBatches=30, miniBatchSize=10, optimizer=optimizer,
                            schedulerLambda=lrLambda, criterion=criterion, IC=IC, K=k, c=c, device=device,
                            resultShowingInterval=50)

# Start training
routines.normalTrainingRoutine(net, f)
```

Figure 35: Code used for the description of the definitions used in the first mass-spring implementation.

Now, the training algorithm is described. A loop over all the epochs is started. A range in which the differential equation is solved is set and discretized. The loss of each point in the discretization is determined and normalized. This is then used as initial probability distribution to sample batches from. The code is shown in figure 36.

```python
def normalTrainingRoutine(self, net, f):
    running_loss = 0.0

    for epoch in range(self.startEpoch, self.epochs):
        totalLoss = torch.tensor(0.0, device=device)
        self.optimizer.zero_grad()
        net.eval()

        allPoints = torch.linspace(0, 30, 1000, requires_grad=True, device=self.device)
        allPoints = allPoints[:, None]

        delta = f(allPoints, net, self.K, self.c, self.device)
        distribution = delta ** 2

        self.cumulative, self.scaleFactor = self._cumulativeFromDistribution(distribution)

        allInputs = torch.tensor([], dtype=torch.float, device=device)
        allProps = torch.tensor([], dtype=torch.float, device=device)
        net.train()
```

Figure 36: Code used for at the start of an epoch.

Now, the loop over all mini batches is started. Firstly, a batch is sampled randomly from the probability distribution that has been created earlier. The mini batch is then entered into the neural network, from which the output is differentiated twice and the loss is created from the differential equation. The loss from the initial conditions is also calculated and the two losses are added together to get the total mini batch loss. Now, an optimization step and a learning rate scheduler step have been set. The code can be seen in figure 37.

```
for i in range(self.numberOfMiniBatches):
    self.optimizer.zero_grad()

    input, props = self._randomFromNumericalDistribution(self.cumulative, self.miniBatchSize, allPoints)
    allInputs = torch.cat((allInputs, input))
    allProps = torch.cat((allProps, props))

    output = f(input, net, self.K, self.c, self.device)

    IV = self._iv(net, device)

    target = torch.zeros(output.shape, device=self.device)

    lossIc = self.criterion(IV, self.IC)
    lossL = 10 * self.criterion(output, target)
    loss = lossIc + lossL

    loss.backward()
    self.optimizer.step()

    running_loss += loss.item()
    totalLoss += loss

self.scheduler.step(totalLoss)
```

Figure 37: Code used for performing an optimization step over a mini batch.

In order to plot the results of the training algorithm, a plot of the output of the neural net of every 50 epochs is made. Since the output of the neural network is the approximation of the solution, this is directly plotted together with the analytical solution of the system.

A.2 Sampling points from a custom distribution

In this section, the second method of solving the mass-spring system is described. This method is originally made to solve the Navier-Stokes equations. For the Navier-Stokes equations however, it is difficult to quantitatively judge the results, since an analytical solution is unknown and a 3-dimensional solution that changes over time is more difficult to plot than a 1-dimensional solution. As a test case for the method, the same method is therefore used to solve the less complicated mass-spring system.

The method has some core differences compared to the first method. Firstly, the method tries to be as mesh free as possible. This means that the differential equation is not solved on a discrete set of points, but points are rather sampled from a continuous distribution. However, this continuum is limited by the computer precision.

Secondly, the distribution from which batches are sampled is updated after each mini batch from the same loss calculation, which is used for the optimization step.

A.2.1 Mesh free solving

The ability to solve a differential equation without having to generate a mesh would give this method an advantage over existing methods. During the loss creation of a batch, the loss for each point is calculated independently of the loss at other points. Therefore, only the batch generator should be modified to sample points from a continuous probability density function. The continuous probability density function is created by linearly interpolating a discrete probability mass function. This means that a discrete mesh should be created for generating the probability density function.

Note that the mesh for this probability density function is allowed to be much smaller than the typical meshes used in finite element methods. Moreover, if a truly mesh free method is desired, the probability density function can be set to the uniform distribution. The solver will be less efficient, since it is not able to focus on points which have a high loss. A linearly interpolated uniform distribution remains uniform, so a mesh of only 2 points per dimension is needed.

Now, the code used to generate a continuous probability density function is discussed. Firstly, a mesh is generated for the discrete probability mass function. For these points, the loss is calculated, which gives the probability distribution when normalized. Note that the loss might be zero everywhere, so the distribution cannot be normalized. Therefore, it is checked if the distribution is zero everywhere and if it is, some small random noise is added.

Now, the discrete cumulative distribution is generated using the build in function torch.cumsum(), which is equivalent to numpy.cumsum() or the cumsum function in Matlab. The cumulative distribution is immediately normalized by dividing the whole distribution by the last value. Note that the implemented function is able to handle multiple dimensions, which makes this process possible when solving the Navier-Stokes equations as well.

To sample from this distribution, the inverse of the cumulative function is created. Since the function is still discrete, this is simply done by swapping the coordinates of each point. The discrete cumulative function is now interpolated linearly, in such a way that a continuous cumulative probability function is retrieved, which is called $F(x)$.

Now, Y is defined as a random variable with the uniform distribution on $[0, 1]$. PyTorch provides the torch.rand() function, which does exactly that. Note that $F(Y)$ is distributed according to the earlier constructed distribution. Since we are now able to sample points from this continuous distribution, the whole method only uses the mesh created earlier for the discrete distribution.

After this initialization, the distribution is updated after each epoch. The update procedure is described in the next section.

A.2.2 Updating the distribution

In order to construct the loss of a batch, it is necessary to calculate the loss at all the points in that batch. The total loss is then primarily used to set an optimization step, but it can be used secondarily to update the sampling distribution. In this section, the update process is described in detail. Note that this updating procedure is not present in the implementation of the Navier-Stokes equation, because this procedure is not easily extended to more dimensions.

First of all, batch points are added to the discrete distribution. Since the distribution is not normalized before, this can be done without any scaling of the obtained loss.

There are two problems with this approach that need to be solved. Firstly, if the loss at the batch points is much higher then the distribution values, the distribution might become very "spiky" at the batch points. When points are sampled from the new distributions, these will almost all be in the new spikes. This might result in the spikes growing even larger, which subsequently results in the collapse of the distribution into a few spikes. This is undesired behaviour, since the optimization algorithm should optimize the solution over the whole range.

Secondly, if a point is analyzed batches ago, the loss in that point might have changed considerably, meaning that the distribution is not a representation of the error of the solution anymore. Older loss evaluations should therefore be less meaningful than newer evaluations.

Both of these problems are solved with the same method: the spikes in the distribution should "spread out" over training time. That way, the distribution cannot collapse to a spiky solution and older loss evaluations will converge to the average loss, in such a way that the old evaluation loses

its meaning. The method should also treat the total loss as a conserved quantity.

Note that these properties are all satisfied if the distribution is allowed to diffuse. A diffusion problem is governed by the differential equation $\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}$ where $k > 0$. The boundary conditions are $\frac{\partial u}{\partial x}(t, x = 0) = \frac{\partial u}{\partial x}(t, x = L) = 0$. The initial condition is given by some continuous function $f(x)$, $u(t = 0, x) = f(x)$. In order to work with complex exponentials, the even extension of $f(x)$ is used: $f(x) = f(-x) \quad x \in [-L, 0]$. The solution is given by equation 21:

$$u(t,x) = \sum_{n=-\infty}^{n=\infty} A_n e^{\frac{n\pi i x}{L}} e^{-\frac{n^2\pi^2}{L^2}kt} \tag{21}$$

$$A_n = \frac{1}{2L}\int_{-L}^{L} f(x)e^{-\frac{n\pi i x}{L}}\,dx$$

Note that for large t, $u(t, x) = A_0$, so the peaks in the initial condition are spread out. Also note that $\int_0^L u(t, x)dx$ is a conserved quantity, which is seen in equation 22.

$$\frac{d}{dt}\int_0^L u(t,x)dx = \int_0^L \frac{\partial u}{\partial t}dx = \int_0^L k\frac{\partial^2 u}{\partial x^2}dx = k\left.\frac{\partial u}{\partial x}(t,x)\right|_{x=0}^{x=L} = 0 \tag{22}$$

In our application, the time variable t becomes the training time and k becomes some parameter that controls the rate at which the peaks spread out. A good value of this parameter is determined experimentally.

An example of this distribution diffusion process is given in figures 38 and 39.

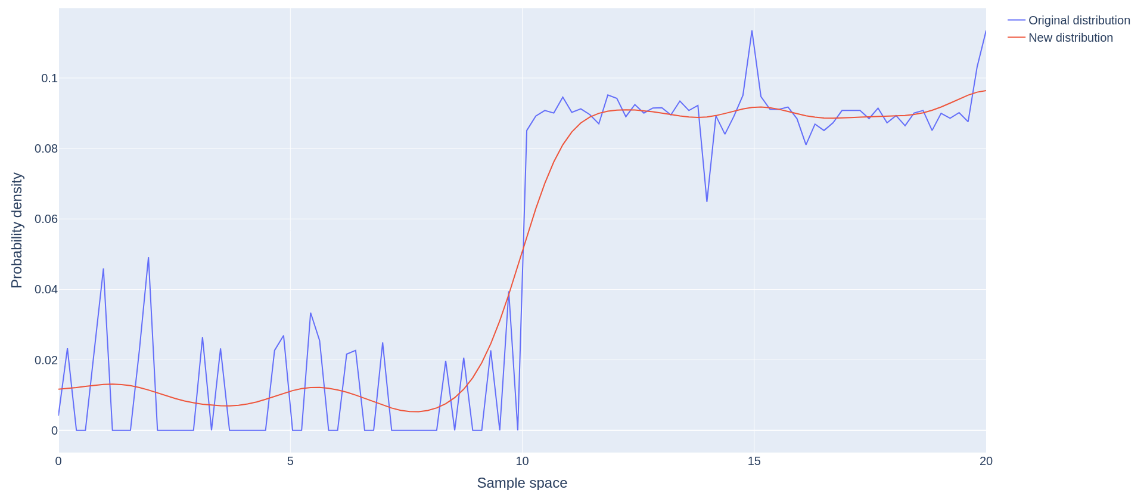

Figure 38: Example of a distribution based on the loss per point and a smoothed version of this distribution. The smoothed version is obtained by allowing the original distribution to diffuse.

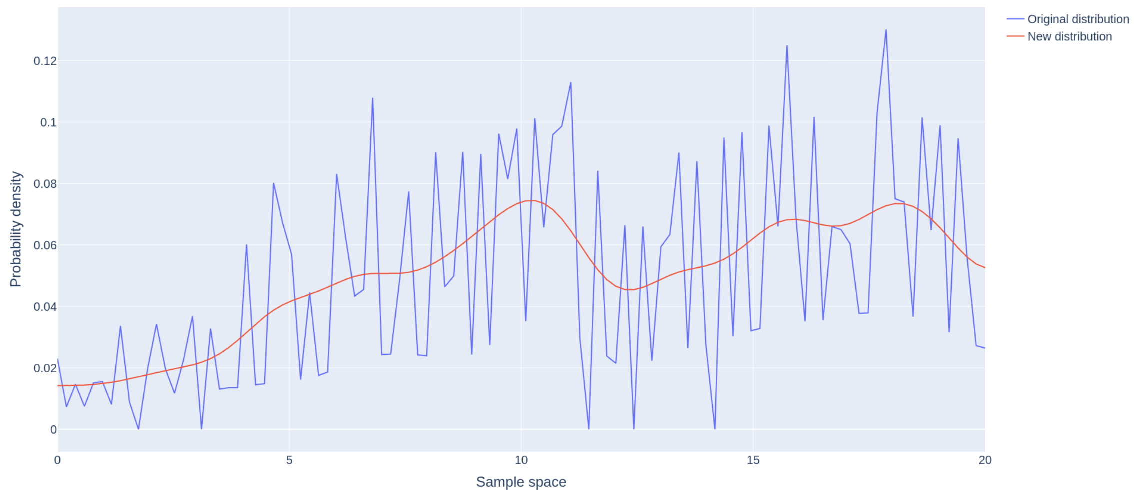

Figure 39: Example of a distribution based on the loss per point and a smoothed version of this distribution. The smoothed version is obtained by allowing the original distribution to diffuse.

PyTorch provides the useful Real Fast Fourier Transform (torch.rfft) and Inverse Real Fast Fourier Transform (torch.irfft) functions. These functions calculate the discrete Fourier transform of a discrete signal x_m of length n.

The PyTorch rfft implementation is given in equation 23 and the irfft implementation is given in equation 24 [2]. Note that the PyTorch documentation merely states that the functions are equal to the Numpy implementation. A reference is therefore made to the Numpy implementation.

$$A_k = \sum_{m=0}^{n-1} x_m \exp\left(-2\pi i \frac{mk}{n}\right) \qquad k = 0, \dots, n-1 \tag{23}$$

$$x_m = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp\left(2\pi i \frac{mk}{n}\right) \qquad m = 0, \dots, n-1 \tag{24}$$

Even though this implementation cannot be used for the heat equation, since a discrete and finite series is required for these equations, the implementation is still useful. The coefficients A_k are simply multiplied with the exponential decaying factor in equation 21, so the desired behaviour is still there.

Remember that the distribution is now known as a discrete sequence which is linearly interpolated. When a batch is created and the loss is calculated for this batch, the batch is added to the distribution sequence. Note that the distance between the points in the sequence is not constant anymore, which is a requirement for the use of the torch.rfft function. The sequence is therefore interpolated linearly for a given number of equidistant points. Usually, the number of interpolation points is equal to the mesh size, since a constant number of mesh points is then guaranteed.

It was found during experimentation that the neural network does not optimize easily on the linearly interpolated points between the mesh points. The loss at these points never gets updated into the new distribution, since the equidistant sampling is always at the mesh points. It is found that some randomness in the number of equidistant points is beneficial, so a random number between 0 and 10 is added to the number of equidistant points.

Now, the conservation of loss property given in equation 22 is only satisfied if $f(x)$ is an even function. The even continuation of the distribution is therefore constructed. If the original

distribution is represented by $\tilde{x}[n]$, then the even continuation is given by equation 25.

$$x[n] = \begin{cases} \tilde{x}[n], & \text{if } 0 \leq n \leq N \\ \tilde{x}[2N-n], & \text{if } N < n \leq 2N \end{cases} \tag{25}$$

which results in a sequence $x[n], \quad 0 \leq n \leq 2N$.

Now, the sequence A_k is calculated according to equation 23 using torch.rfft(). From equation 21, the exponential factor is added: $\tilde{A}_n = A_n \exp\left(-\frac{n^2\tau}{L^2}\right)$, where $\tau = \pi^2 kt$ is the decay speed parameter that has been determined experimentally. The new distribution is subsequently obtained by using torch.irfft().

References

[1] Pytorch. URL: https://pytorch.org/tutorials/.

[2] Numpy. URL: https://numpy.org/doc/stable/reference/routines.fft.html.

[3] William Chan et al. "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition". In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 4960–4964. DOI: 10.1109/ICASSP.2016.7472621.

[4] Ronan Collobert et al. "Natural Language Processing (almost) from Scratch". In: *CoRR* abs/1103.0398 (2011). arXiv: 1103.0398. URL: http://arxiv.org/abs/1103.0398.

[5] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *arXiv e-prints*, arXiv:1512.03385 (Dec. 2015), arXiv:1512.03385. arXiv: 1512.03385 [cs.CV].

[6] Catherine F. Higham and Desmond J. Higham. *Deep Learning: An Introduction for Applied Mathematicians*. 2018. arXiv: 1801.05894 [math.HO].

[7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[8] Zheng Yi. "Evaluation and Implementation of Convolutional Neural Networks in Image Recognition". In: *Journal of Physics: Conference Series* 1087 (Sept. 2018), p. 062018. DOI: 10.1088/1742-6596/1087/6/062018. URL: https://doi.org/10.1088/1742-6596/1087/6/062018.