University of Helsinki
Faculty of science
Department of mathematics and statistics

Master's thesis

Operator recurrent neural network approach to an inverse problem for the wave equation

Aili Joutsela

Supervisor: Lauri Oksanen April 4, 2023

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Science	Mathematics and statistics

Tekijä — Författare — Author

Aili Joutsela

Työn nimi — Arbetets titel — Title

Operator recurrent neural network approach to an inverse problem for the wave equation

Oppiaine — Läroämne — Subject

Mathematics

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	February 2023	43

Tiivistelmä — Referat — Abstract

In my mathematics master's thesis we dive into the wave equation and its inverse problem and try to solve it with neural networks we create in Python. There are different types of artificial neural networks. The basic structure is that there are several layers and each layer contains neurons. The input goes to all the neurons in the first layer, the neurons do calculations and send the output to all the neurons in the next layer. In this way, the input data goes through all the neurons and changes and the last layer outputs this changed data. In our code we use operator recurrent neural network. The biggest difference between the standard neural network and the operator recurrent neural network is, that instead of matrix-vector multiplications we use matrix-matrix multiplications in the neurons.

We teach the neural networks for a certain number of times with training data and then we check how well they learned with test data. It is up to us how long and how far we teach the networks. Easy criterion would be when a neural network has learned the inversion completely, but it takes a lot of time and might never happen. So we settle for a situation when the error, the difference between the actual inverse and the inverse calculated by the neural network, is as small as we wanted.

We start the coding by studying the matrix inversion. The idea is to teach the neural networks to do the inversion of a given 2×2 real valued matrix. First we deal with networks that don't have the activation function ReLU in their layers. We seek a learning rate, a small constant, that speeds up the learning of a neural network the most. After this we start comparing networks that don't have ReLU layers to networks that do have ReLU layers. The hypothesis is that ReLU assists neural networks to learn quicker.

After this we study the one-dimensional wave equation and we calculate its general form of solution. The inverse problem of the wave equation is to recover wave speed $c(x)$ when we have boundary terms. Inverse problems in general do not often have a unique solution, but in real life if we have measured data and some additional a priori information, it is possible to find a unique solution. In our case we do know that the inverse problem of the wave equation has a unique solution.

When coding the inverse problem of the wave equation we use the same approach as with the matrix inversion. First we seek the best learning rate and then start to compare neural networks with and without ReLU layers. The hypothesis once again is that ReLU supports the learning of the neural networks. This turns out to be true and happens more clearly with wave equation than with matrix inversion. All the teaching was run on one computer. There is a chance to get even better results if a more powerful computer is used.

Avainsanat — Nyckelord — Keywords

Wave equation, Inverse problem, Operator recurrent neural network

Säilytyspaikka — Förvaringsställe — Where deposited

Library of Kumpula campus

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

In this mathematics master's thesis we dive into the wave equation and its inverse problem and try to solve it with neural networks we create in Python [8]. There are different types of artificial neural networks [5], [6], [14]. The basic structure is that there are several layers and each layer contains neurons. The input goes to all the neurons in the first layer, the neurons do calculations and send the output to all the neurons in the next layer. In this way, the input data goes through all the neurons and changes and the last layer outputs this changed data. In our code we use operator recurrent neural network introduced in article [6]. The biggest difference between the standard neural network and the operator recurrent neural network is, that instead of matrix-vector multiplications we use matrix-matrix multiplications in the neurons.

We teach the neural networks for a certain number of times with training data and then we check how well they learned with test data. It is up to us how long and how far we teach the networks. Easy criterion would be when a neural network has learned the inversion completely, but it takes a lot of time and might never happen. So we settle for a situation when the error, the difference between the actual inverse and the inverse calculated by the neural network, is as small as we wanted.

We start the coding by studying the matrix inversion [7]. The idea is to teach the neural networks to do the inversion of a given 2×2 real valued matrix. First we deal with networks that don't have the activation function ReLU in their layers. We seek a learning rate, a small constant, that speeds up the learning of a neural network the most. After this we start comparing networks that don't have ReLU layers to networks that do have ReLU layers. The hypothesis is that ReLU assists neural networks to learn quicker [4], [11].

After this we study the one-dimensional wave equation [1], [2], [12], and we calculate its general form of solution. The inverse problem of the wave equation is to recover wave speed $c(x)$ when we have boundary terms. Inverse problems in general do not often have a unique solution, but in real life if we have measured data and some additional a priori information, it is possible to find a unique solution [9]. In our case we do know that the inverse problem of the wave equation has a unique solution [10].

When coding the inverse problem of the wave equation we use the same approach as with the matrix inversion. First we seek the best learning rate and then start to compare neural networks with and without ReLU layers. The hypothesis once again is that ReLU supports the learning of the neural networks. This turns out to be true and happens more clearly with wave equation than with matrix inversion.

All the teaching was run on one computer. There is a chance to get even better results if a more powerful computer is used. Thanks to Līva Freimane for helping with the code. Līva's code can be found in [3].

2 About artificial neural networks

2.1 Standard neural network

Initially we have a data set and a problem regarding the data set that we want to solve. We form a mapping to help us. When approaching the problem from an artificial neural network perspective we use repeatedly some simple nonlinear function σ, for example sigmoid function, to the data we have.

At a general form of a neural network there are an input layer, general layers and an output layer. Each layer has multiple neurons. The number of layers and neurons varies between different networks. At the input layer the neutrons receive the input vector. At the general layers, each neuron receives the values formed by the neurons in the previous layer and produces a value which is passed to every neuron at the next layer. Each neuron forms its own weighted (scaled) combination of these values, adds its own bias (shifting) and applies the function σ. Finally at an output layer the neurons provide the overall output of the network. So neural network is like mapping. We give input data to the network and it gives us an output.

We vectorize the nonlinear function σ so we will manage the notation of every layer of neurons. For $z \in \mathbb{R}^m, \sigma\colon \mathbb{R}^m \to \mathbb{R}^m$ is defined componentwise such that $(\sigma(z))_i = \sigma(z_i)$. If we collect the values, real numbers, produced by the neurons in one layer into a vector a, then the output vector from the next layer has the form

$$\sigma(Wa + b), \tag{1}$$

where W is a matrix that contains the weights and b is a vector that contains the biases. The number of columns in W is the number of neurons in the previous layer. The number of rows in W and the number of components in b is the number of neurons at the current layer. So the value produced by the ith neuron in the network would be

$$\sigma\left(\sum_j w_{ij}a_j + b_i\right), \tag{2}$$

where the sum runs over all entries in the vector a.

2.1.1 General form of notation

We mark the layers of a network with l. Layer 1 is the input layer and layer L is the output layer. Layer l, for $l = 1, 2, 3, \ldots, L$, contains n_l neurons and this indicates the dimension of the layer. The input data has a dimension of n_1 and the output data has a dimension of n_L, so the network maps from $\mathbb{R}^{n_1}$ to $\mathbb{R}^{n_L}$. The matrix W of weights at the layer l is denoted by $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$. The element $w^{[l]}_{jk}$ of the matrix $W^{[l]}$ is the weight that the neuron j at the layer l applies to the output from the neuron k at the previous layer $l-1$. And the vector b of biases for layer l is marked by $b^{[l]} \in \mathbb{R}^{n_l}$, so the neuron j at the layer l uses the bias $b^{[l]}_j$. For the input $x \in \mathbb{R}^{n_1}$ and the output, or activation, $a^{[l]}_j$, from the neuron j at the layer l, the action of the network is

$$a^{[1]} = x \in \mathbb{R}^{n_1}, \tag{3}$$

$$a^{[l]} = \sigma\left(W^{[l]}a^{[l-1]} + b^{[l]}\right) \in \mathbb{R}^{n_l} \quad \text{for } l = 2, 3, \ldots, L. \tag{4}$$

To make this model look comparable with the operator recurrent network, introduced in section 2.2, we need to modify it a little. We write the standard neural network as a function $f_\theta : \mathbb{R}^{n_1} \to \mathbb{R}^{n_L}$. We also write the weight matrices and the bias vectors a little differently by $W^{[l]} = W_\theta^{[l,1]}$ and $b^{[l]} = b_\theta^{[l]}$, respectively. The function f_θ handles the given data in two ways and adds them together. The first part is to multiply the data with the matrix $W_\theta^{[l,0]} \in \mathbb{R}^{n_l \times n_{l-1}}$ and add the vector $b_\theta^{[l]} \in \mathbb{R}^{n_l}$. The matrix $W_\theta^{[l,0]}$ skips an activation function (introduced in section 2.3.3) and connects the output from previous layer $l-1$ to the next layer l. In the second part the data is put to the function σ. We call the function σ an *activation function*. The activation function applies a scalar function to each component $a^{[l]}$, that is, for $a = (a_j)_{j=1}^{n_l} \in \mathbb{R}^{n_l}$, $\sigma_l(a) = (\sigma_l(a_j))_{j=1}^{n_l} \in \mathbb{R}^{n_l}$. The *standard neural network* defined this way is

$$a^{[1]} = x \in \mathbb{R}^{n_1}, \tag{5}$$

$$f_\theta(a^{[1]}) = a^{[L]} \in \mathbb{R}^{n_L}, \tag{6}$$

$$a^{[l]} = W_\theta^{[l,0]} a^{[l-1]} + \sigma\left(W_\theta^{[l,1]} a^{[l-1]} + b_\theta^{[l]}\right) \in \mathbb{R}^{n_l} \quad for\ l = 2, 3, \ldots, L. \tag{7}$$

Each of $W_\theta^{[l,0]}$, $W_\theta^{[l,1]}$, $b_\theta^{[l]}$ are dependent on parameters θ, which is to be learned later.

2.1.2 Example of a neural network

In figure 1 we see an example of how a neural network can look. First we have an input layer containing one circle denoting one input vector. The second and the third layer, general or hidden layers, have three circles indicating three neurons. The fourth layer, the output layer, has two circles indicating two neurons. Arrows from the neuron in the input layer to the neurons in the first hidden layer indicate that the data from the input vector is available for all three neurons in the first hidden layer. In the same way the data from the neurons in the first hidden layer is available to all the neurons in the next hidden layer and from this layer to the output layer. Arrows from the output layer indicates the overall output that the network gives.

If the input data has the form of $x \in \mathbb{R}$, by using the notation in equations (3) and (4) the weights and biases for the first hidden layer, overall layer 2, may be represented by a matrix $W^{[2]} \in \mathbb{R}^{3\times 1}$ and a vector $b^{[2]} \in \mathbb{R}^3$, respectively. So the output from layer 2 has the form

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^3. \tag{8}$$

The second hidden layer, overall layer 3, has also three neurons, each receiving input in $\mathbb{R}^3$ from layer 2. The weights and biases for the second hidden layer may be represented by a matrix $W^{[3]} \in \mathbb{R}^{3\times 3}$ and a vector $b^{[3]} \in \mathbb{R}^3$, respectively and the output from layer 3 has the form

$$\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) \in \mathbb{R}^3. \tag{9}$$

The fourth and the last layer, output layer, has two neurons. These neurons are receiving input in $\mathbb{R}^3$ and then the weights and biases for layer 4 may be

represented by a matrix $W^{[4]} \in \mathbb{R}^{2\times 3}$ and a vector $b^{[4]} \in \mathbb{R}^2$, respectively and the output from layer 4, and the overall network, has the form of

$$F(x) = \sigma\left(W^{[4]}\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) + b^{[4]}\right) \in \mathbb{R}^2. \tag{10}$$

The function $F : \mathbb{R} \to \mathbb{R}^2$ of the network is defined in the expression (10). The function has 26 parameters; all the entries of the weight matrices and bias vectors.

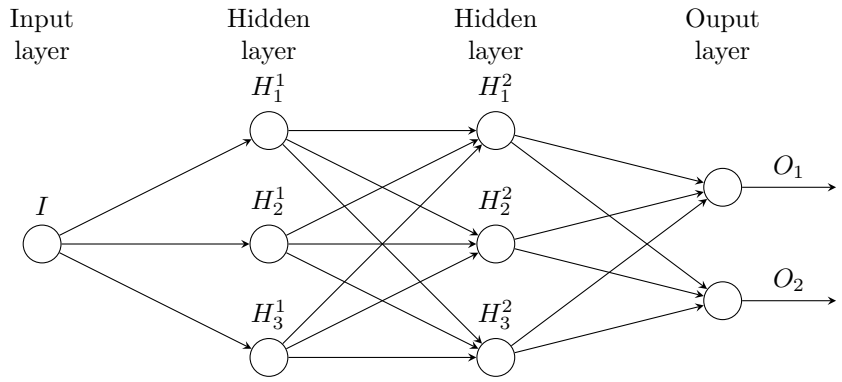

Figure 1: *An example of a neural network layout.*

2.2 Basic and general operator recurrent network

Standard neural networks are useful in many applications, but for approximating functions with multiplicative and highly nonlinear structure they are not efficient [14]. This motivates us to get to know the operator recurrent neural network. When the standard neural network vectorizes the input and performs matrix-vector multiplications, the operator recurrent network performs matrix-matrix multiplications directly. This is the biggest difference between these two networks. Next two definitions are based on [6].

Definition 1. *A basic operator recurrent network* with depth L, width n and set of parameters (or weights) θ is defined as a function $f_\theta : \mathbb{R}^{n\times n} \to \mathbb{R}^n$ given by

$$f_\theta(X) = a^{[L]}, \tag{11}$$

$$\begin{aligned} a^{[l]} &= b_\theta^{[l,0]} + W_\theta^{[l,0]} a^{[l-1]} + B_\theta^{[l,0]} X a^{[l-1]} \\ &\quad + \sigma_l(b_\theta^{[l,1]} + W_\theta^{[l,1]} a^{[l-1]} + B_\theta^{[l,1]} X a^{[l-1]}), \end{aligned} \tag{12}$$

where $X \in \mathbb{R}^{n\times n}$ is the input dataset, $a^{[1]} \in \mathbb{R}^n$ is an initial vector not explicitly given by the data, the quantities $b_\theta^{[l,0]}, b_\theta^{[l,1]} \in \mathbb{R}^n$ and $W_\theta^{[l,0]}$, $W_\theta^{[l,1]}$, $B_\theta^{[l,0]}$, $B_\theta^{[l,1]} \in \mathbb{R}^{n\times n}$ are dependent on the parameters θ, and the σ_l are the activation functions.

From the basic operator recurrent network we get to the general network by adding memory.

Definition 2. *A general operator recurrent network* of level K is an extension of the basic operator recurrent network, including terms that contain $a^{[l-k]}$ in the expression for $a^{[l]}$, that is

$$f_\theta(X) = a^{[L]}, \tag{13}$$

$$a^{[l]} = b_\theta^{[l,0]} + \sum_{k=1,\dots,K} \left(W_\theta^{[l,k,0]} a^{[l-k]} + B_\theta^{[l,k,0]} X a^{[l-k]} \right) + \tag{14}$$

$$\sigma_l \left(b_\theta^{[l,1]} + \sum_{k=1,\dots,K} \left(W_\theta^{[l,k,1]} a^{[l-k]} + B_\theta^{[l,k,1]} X a^{[l-k]} \right) \right),$$

for $l \geq 1$, where $a^{[0]} \in \mathbb{R}^n$ is some initial vector not explicitly given by the data and the quantities $b_\theta^{[l,0]}, b_\theta^{[l,1]} \in \mathbb{R}^n$ and $W_\theta^{[l,k,0]}$, $W_\theta^{[l,k,1]}$, $B_\theta^{[l,k,0]}$, $B_\theta^{[l,k,1]} \in \mathbb{R}^{n \times n}$ are dependent on the parameters θ, and the σ_l are the activation functions.

So how the biases and weights depend on the parameters θ? For a basic operator recurrent network we have $4n$ column vectors $\theta_1^{l,i}, \dots, \theta_{4n}^{l,i} \in \mathbb{R}^n$ within the parameter set θ for each layer l and $i = 0, 1$ such that

$$W_\theta^{[l,i]} = W^{[l,i,(0)]} + W_\theta^{[l,i,(1)]}, \quad W_\theta^{[l,i,(1)]} = \sum_{p=1}^{n} \theta_{2p-1}^{l,i} \left(\theta_{2p}^{l,i} \right)^T, \tag{15}$$

and

$$B_\theta^{[l,i]} = B^{[l,i,(0)]} + B_\theta^{[l,i,(1)]}, \quad B_\theta^{[l,i,(1)]} = \sum_{p=n+1}^{2n} \theta_{2p-1}^{l,i} \left(\theta_{2p}^{l,i} \right)^T. \tag{16}$$

Both $W^{[l,i,(0)]}$ and $B^{[l,i,(0)]}$ are fixed operators that do not depend on parameter θ. They are chosen depending on the specific application, but normally they are either zero operator or the identity operator.

For the bias vectors we parametrize them by $b_\theta^{[l,i]} = \theta_0^{l,1,i} \in \mathbb{R}^n$, where $i = 0, 1$. Now parameters θ can be written as an ordered sequence

$$\begin{aligned} \theta = &[\theta_p^{l,i} \in \mathbb{R}^n : l = 1, 2, \dots, L,\ p = 1, 2, \dots, 4n,\ i = 0, 1] \\ &\cup [\theta_0^{l,i} \in \mathbb{R}^n : l = 1, 2, \dots, L,\ i = 0, 1]. \end{aligned} \tag{17}$$

To get to the general recurrent operator networks we add the index $k = 1, \dots, K$ to the ordered sequences

$$\begin{aligned} \tilde{\theta} = &[\tilde{\theta}_p^{l,i,k} \in \mathbb{R}^n : l = 1, 2, \dots, L,\ p = 1, 2, \dots, 4n,\ k = 1, 2, \dots, K,\ i = 0, 1] \\ &\cup [\tilde{\theta}_0^{l,i,0} \in \mathbb{R}^n : l = 1, 2, \dots, L,\ i = 0, 1]. \end{aligned} \tag{18}$$

With the parameters shown above, each layer is allowed to have different and independent parameters.

2.3 How to teach a neural network

2.3.1 Stochastic gradient descent

When training a neural network, the main goal is to choose the weights and biases so that they minimize a cost function. In this section we consider that the weights and biases are stored in a single vector $p \in \mathbb{R}^s$, for some $s \in \mathbb{N}$. A cost function is a function of the weights and biases. It tells of the discrepancy between an output of a network and a wanted output. To get closer to this wanted output, we want to minimize the discrepancy and thereby minimize the cost function. The cost function is an objective function and one common form of it is a quadratic cost function, where the square Euclidean norm is averaged over the k data points, or training points, in $\mathbb{R}^{n_1}$, $\{x^{\{i\}}\}_{i=1}^k$. For the data points there are given target outputs $\{y(x^{\{i\}})\}_{i=1}^k \in \mathbb{R}^{n_L}$. So the quadratic cost function $\text{Cost} : \mathbb{R}^s \to \mathbb{R}$ we want to minimize has the form

$$\text{Cost}(p) = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2. \tag{19}$$

The *gradient* or *steepest descent* method is a classical optimization method. The aim of the method is to find a vector v that minimizes the cost function. It proceeds iteratively by computing a sequence of vectors in $\mathbb{R}^s$ and tries to produce such sequence that converges to the vector v. Our starting point is the vector p and we want to find perturbation Δp such that the next vector $p + \Delta p$ shows improvement from the vector p. Assume Δp is small and we can ignore terms of order $\|\Delta p\|^2$ and higher. Then the Tailor series expansion gives

$$\text{Cost}(p + \Delta p) \approx \text{Cost}(p) + \sum_{r=1}^{s}\frac{\partial\,\text{Cost}(p)}{\partial p_r}\Delta p_r, \tag{20}$$

where the $\partial\,\text{Cost}(p)/\partial p_r$ denotes the partial derivative of the cost function with respect to the rth parameter. We put all the partial derivatives into a vector $\nabla\text{Cost}(p) \in \mathbb{R}^s$, the gradient, so that

$$\nabla\text{Cost}(p)_r = \frac{\partial\,\text{Cost}(p)}{\partial p_r}. \tag{21}$$

Then the equation (20) becomes

$$\text{Cost}(p + \Delta p) \approx \text{Cost}(p) + \nabla\text{Cost}(p)^T\Delta p. \tag{22}$$

To minimize the value of the cost function we use the Cauchy-Schwarz inequality. The inequality states that for any $f, g \in \mathbb{R}^s$, we have $|f^T g| \leq \|f\|_2\|g\|_2$. By looking at the equation (22), choosing Δp to lie in the direction of $-\nabla\text{Cost}(p)$ we will reduce the value of the cost function by making $\nabla\text{Cost}(p)^T\Delta p$ as negative as possible. Equation (22) is only an approximation for small Δp so it is wise to proceed with small steps in that direction. Then we will write

$$p \to p - \eta\nabla\text{Cost}(p). \tag{23}$$

The η is a small stepsize known as the *learning rate*. This equation defines the steepest descent method; we choose an initial vector and iterate with (23)

until a stopping criterion is met. Since our cost function (19) is the sum of individual terms also the partial derivative $\nabla\text{Cost}(p)$ is a sum of individual partial derivatives over the training data. More precisely, let

$$C_{x^{\{i\}}} = \frac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2, \tag{24}$$

and then from (19) we get

$$\nabla\text{Cost}(p) = \frac{1}{k}\sum_{i=1}^{k} \nabla C_{x^{\{i\}}}(p). \tag{25}$$

In steepest descent method computing the gradient vector in every iteration can be expensive when having a large number of parameters and a large number of training points. If we replace the mean of the gradients over all training points by the gradient at a randomly chosen single training point we get a much cheaper alternative. This method is called *stochastic gradient method*. So we pick only one training point at a time by randomly choosing an integer i uniformly from $\{1, 2, 3, \ldots, k\}$ and then update $p \to p - \eta\nabla C_{x^{\{i\}}}(p)$. As the iteration proceeds the method sees more training points, but after each step, the index i and the used training point is put back to the training set. So the same point has the same odds as any other point to be chosen at the next step.

An alternative way, that we use in our coding in section 3.3, is to not put the used training point back to the training set, but to go through each of the k training points in random order. Doing so is called as completing an *epoch*. The idea is to put integers $\{1, 2, 3, \ldots, k\}$ into a new order $\{n_1, n_2, n_3, \ldots, n_k\}$ and update $p \to p - \eta\nabla C_{x^{\{n_i\}}}(p)$ in this new order.

We want to compromise between stochastic gradient and the steepest descent method. We will use a small sample average by choosing $m \ll k$ integers $n_1, n_2, n_3, \ldots, n_m$ uniformly randomly from $\{1, 2, 3, \ldots, k\}$ and then updating

$$p \to p - \eta\frac{1}{m}\sum_{i=1}^{m} \nabla C_{x^{\{n_i\}}}(p). \tag{26}$$

Here the set $\{x^{\{n_i\}}\}_{i=1}^m$ is known as a *minibatch*. In our code, in section 3.3, we have defined it in variable called "batch size". We pick a j such that $k = j \cdot m$ and split the training set into j minibatches and cycle through them.

2.3.2 Backpropagation

Now that we know what stochastic gradient method is, we can use it to train a neural network. We switch back to the weight matrices $W^{[l]}$ and bias vectors $b^{[l]}$ from the general vector p of parameters and we want to compute the partial derivatives of the cost function. The cost function in (19) is a linear combination of individual terms that runs over the training data. The same applies for its partial derivatives. So our focus now is to calculate the individual partial derivatives. For a fixed training point we drop the dependence on $x^{\{i\}}$ in (24) and write

$$C = \frac{1}{2}\|y - a^{[L]}\|_2^2. \tag{27}$$

C depends on weights and biases only through $a^{[L]}$.

To make things little bit easier, let's introduce two new sets of variables. First off

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l} \quad \text{for } l = 2, 3, \ldots, L. \tag{28}$$

Here an element $z_j^{[l]}$ of $z^{[l]}$ is the *weighted input* for neuron j at layer l. With this new notation we can write equation (4) as

$$a^{[l]} = \sigma(z^{[l]}) \quad \text{for } l = 2, 3, \ldots, L. \tag{29}$$

Second, let $\delta^{[l]} \in \mathbb{R}^{n_l}$, often called *error*, be defined as

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} \quad \text{for } 1 \leq j \leq n_l \quad \text{and} \quad 2 \leq l \leq L. \tag{30}$$

Calling $\delta^{[l]}$ an error is little ambiguous [5], but $\delta_j^{[l]} = 0$ is a useful goal, since then all partial derivatives are zero and the cost function is at its minimum.

Next, we want to define the Hadamard, or componentwise, product of two vectors: If $x, y \in \mathbb{R}^n$, then $x \circ y \in \mathbb{R}^n$ is defined by $(x \circ y)_i = x_i y_i$.

Lemma 1. *Following results are consequence of the chain rule.*

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^{[L]} - y), \tag{31}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \quad \textit{for } \ 2 \leq l \leq L-1, \tag{32}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]} \quad \textit{for } \ 2 \leq l \leq L, \tag{33}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \quad \textit{for } \ 2 \leq l \leq L. \tag{34}$$

Lemma 1 is proved in [5, p. 870]. The output $a^{[L]}$ of the network can be evaluated with equations (28) and (29) from a *forward pass* by computing $a^{[1]}$, $z^{[2]}$, $a^{[2]}$, $z^{[3]}$, $a^{[3]}$, $\ldots$, $a^{[L]}$ in order. After this we are able to compute $\delta^{[L]}$ with the equation (31). And with the help of the equation (32) we are able to compute also $\delta^{[L-1]}, \delta^{[L-2]}, \ldots, \delta^{[2]}$ in a *backward pass*. With (33) and (34) we get the partial derivatives we originally wanted.

So first we compute all the outputs of the network and from these outputs we calculate the partial derivatives from the last layer L back to the layer 1. This method of computing gradients is known as *backpropagation*.

2.3.3 Activation function ReLU

In a neural network every neuron has an activation function. Its job is to check whether an incoming data should activate computations in a neuron and give an output to the next neuron or not. We have activation functions in neural networks because without them networks would only be linear regression models. Linear equations are easy to solve, but they have a limited capacity to solve complex problems. Activation functions are usually non-linear for this reason. They make neural networks learn more complex problems.

There are many suitable functions that can be used as an activation function, such as a sigmoid function, a quadratic cost function or a rectified linear unit,

ReLU. The choice is application-specific. There can also be a different activation function for each layer l separately. Our choice of activation function is ReLU

$$\sigma(x) = \begin{cases} 0, & \text{for } x \leq 0 \\ x, & \text{for } x > 0, \end{cases} \tag{35}$$

for all of the layers. There are studies [4], [11], which show that a neural network with ReLU as an activation function increases sparsity within the network which can explain improved performance and accelerated training.

3 Operator recurrent neural network for the matrix inversion

3.1 Matrix inversion

Definition 3. Let $GL(n, \mathbb{R})$ denote the general linear group, the set of all $n \times n$ invertible matrices of real numbers with matrix multiplication as the group operation. That is

$$GL(n, \mathbb{R}) = \big\{ A \in \mathbb{R}^{n \times n} \mid \text{A invertible} \big\}.$$

Now we form a mapping $F \colon GL(n, \mathbb{R}) \to GL(n, \mathbb{R})$, $F(A) = A^{-1}$ for matrix inversion. In section 3.3 we will try to teach this mapping F to a neural network.

3.2 Motivating operator recurrent networks

This section is adapted from [6, p. 59-63].

For an integer $n > 0$ suppose we have a data set

$$\{(X_j, y_j) \mid j = 1, \ldots, s\}, \tag{36}$$

where every $X_j \in \mathbb{R}^{n \times n}$ is an invertible matrix and $y_j \in \mathbb{R}^n$ is a vector. Our problem is to construct a function f with a graph $\{X, y = f(X)\}$. We want this graph to fit closely to the given data set. Assume we also know the algebraic relationship of the dataset to be

$$Xy = h, \tag{37}$$

where $h \in \mathbb{R}^n$ is a fixed vector. Now we can construct the function f as $f(X) = X^{-1}h$, so the actual problem we are having is how to apply the inversion of the matrix X to some particular vector h.

This problem is different from a linear inverse problem. In the linear inverse problem we have a data set consisting of pair of vectors $\{(x_j, y_j)\}$ with linear relationship $x = Ay$ with fixed matrix A. Now, the problem is to construct the linear map $f(x) = A^{-1}y$. A neural network with ReLU activation function is suitable for the job, but we want to investigate whether the operator recurrent network is suitable to solve this problem and under which conditions.

Theorem 4. *Let f_θ be an operator recurrent network on $\mathbb{R}^{n\times n}$ with layerwise outputs h_l, $l = 0, \ldots, L$. Then for each l, there exists a countable collection of polynomial regions $\{U_i^l\}$ in $\mathbb{R}^{n\times n}$ satisfying:*

1. *This collection partitions $\mathbb{R}^{n\times n}$; that is, $U_i^l \cap U_j^l = \emptyset$ for every $i \neq j$, and $\bigcup \bar{U_i^l} = \mathbb{R}^{n\times n}$.*
2. *Every open ball $B \subset \mathbb{R}^{n\times n}$ only nontrivially intersects U_i^l for finitely many i.*
3. *The restriction of h_j to each U_i^l is an operator polynomial of degree at most l, applied to h_0.*

Theorem 4 is proved in [6]. With the help of this theorem we gather that operator recurrent network equals to piecewise matrix polynomial. Now the question to ask is how we approximate the matrix inversion problem with piecewise matrix polynomials.

We can represent the inverse of matrix X with the matrix power series called Neumann series

$$X^{-1} = \sum_{k=0}^{\infty} (I - X)^k. \tag{38}$$

This equality holds for $\|I - X\| < 1$ and that is when the power series converges. If we truncate the power series we are able to approximate X^{-1} by a matrix polynomial, which can be represented by an operator recurrent network. If we happen to have some prior knowledge about spectral information, eigenvalues, of X, we are able to produce approximation with better properties and which also holds for other regions than $\|I - X\| < 1$. So for this, we assume that the matrices X_j are from a set U of orthogonally diagonalizable matrices whose eigenvalues are from a compact set K which does not contain some open neighbourhood of zero. This way it is guaranteed that all X_j and their inverses have uniformly bounded spectral norm. This means that every spectral norm

$$\|X_j\|_2 = \sqrt{\lambda_{max}(X_j^* X_j)} = \sigma_{max}(X_j), \tag{39}$$

where X_j^* is the conjugate transpose of X_j, λ_{max} is the largest eigenvalue and $\sigma_{max}(X_j)$ is the largest singular value of matrix X_j, is bounded by

$$\|X_j\|_2 = \sigma_{max}(X_j) \leq M, \quad M \in \mathbb{R}. \tag{40}$$

Next we introduce a lemma which tells that for matrix X in the set U it is possible to find a polynomial p such that $p(X)h$ approximates $X^{-1}h$ well. Our original problem was to find a function such that $f(X) = X^{-1}h$ and this lemma gives a good approximation for it. For the proof of the lemma we first need Mergelyan's Theorem.

Theorem 5. *(Mergelyan's Theorem) If K is a compact subset of the complex plane $\mathbb{C}$ such that $\mathbb{C} \setminus K$ is connected, and if f is a continuous complex function on K which is holomorphic in the interior of K, and if $\epsilon > 0$, then there exists a polynomial p such that $\|f(z) - p(z)\| < \epsilon$ for all $z \in K$.*

Proof. Can be found in [13]. □

Lemma 2. *Let U consist of the set of orthogonally diagonalizable matrices whose eigenvalues lie in a compact set $K \subset \mathbb{C}$ that does not contain zero, and assume that $\mathbb{C} \setminus K$ is connected. Then there exists a sequence of operator polynomials that approximate the function $X \mapsto X^{-1}$ uniformly on U.*

Proof. Since K does not contain zero, the complex function $z \mapsto 1/z$ is differentiable in K. This makes the function holomorphic on some open set containing K. Then the function is infinitely differentiable and locally equal to its own Taylor series, which means it is an analytic function. Since $\mathbb{C} \setminus K$ is connected the theorem 5 of Mergelyan helps us to construct a sequence of polynomials $\{p_i(z)\}$. This sequence uniformly approximates $z \mapsto 1/z$ on K. By the holomorphic functional calculus, we now have a sequence of operator polynomials $p_i(X)$, that uniformly approximates $X \mapsto X^{-1}$ on U. □

The function $f(z) = 1/z$ fulfils the requirements of Mergelyan's theorem. Next we construct a polynomial $p(z)$. First we take a geometric series

$$\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}.$$

Mark $1 - q = z$ and from here we get $q = 1 - z$. Next we approximate the infinite sum by a finite sum

$$\frac{1}{z} = \sum_{n=0}^{\infty} (1-z)^n \approx \sum_{n=0}^{N} (1-z)^n = p_N(z).$$

Polynomial $p_N(z)$ is the wanted polynomial that approximates the function $f(z) = 1/z$. The approximation holds only when z is close to 1, i.e. $|1 - z| < \epsilon$ for some $\epsilon > 0$.

Now we use holomorphic functional calculus to get a sequence of operator polynomials $\{p_i(X)\}$ that uniformly approximates $X \mapsto X^{-1}$ on U. Since the function $f(z)$ is analytic, we can write it as a power series

$$f(z) = \sum_{n=0}^{\infty} a_n z^n.$$

We replace the complex variable z with complex valued matrix $X \in \mathbb{C}^{n \times n}$

$$f(X) = \sum_{n=0}^{\infty} a_n X^n.$$

Let's remind ourselves about singular values and the operator norm of a matrix $A \in \mathbb{C}^{m \times n}$. A singular value of the matrix A is an eigenvalue $\lambda \in \mathbb{C}$ of a matrix $(A^*A)^{1/2}$. If we mark the singular values as σ_i, where $i = 1, 2, \ldots, \min\{m, n\}$, we can write the relation between the singular value and the eigenvalue as $\sigma^2(A) = \lambda(A^*A) = \lambda(AA^*)$. The biggest singular value σ of the matrix A is called the operator norm of A and is defined as

$$\|A\| = \max\{\sigma_i\} = \max\{\|Ax\| : x \in \mathbb{R}^{n \times 1}, \|x\| = 1\}.$$

The operator norm is possible to write using the eigenvalues in the following way. If the matrix A is symmetric, i.e. $A^* = A$, the eigenvalues of matrix $A^*A = A^2$ are λ^2, where λ is the eigenvalue of A. Singular values are $(\lambda^2)^{1/2} = |\lambda|$. Then

$$\|A\| = \max\{|\lambda| : \lambda \text{ is the eigenvalue of A, A is symmetric}\}.$$

This also applies to our function $\|f(X)\| = \max|f(\lambda)|$. We take a sequence of complex functions f_n such that $|(f_n - f)(z)| < \epsilon$ for $\forall z \in K$. From here we have

$$\|f_n(X) - f(X)\| = \max|f_n(\lambda) - f(\lambda)| < \epsilon, \quad \text{for some } \epsilon > 0.$$

Now the $f_n(X)$ is the sequence of operator polynomials we were looking for.

3.3 Coding for the matrix inversion

We have created a code, found in [8], that generates a neural network and data elements for training and testing the network. The code first teaches the network to do an inversion of a given matrix. The teaching is done in epochs with a training data set. The trained network is then tested with a test data set to see how well it learned to produce an inversion. After testing the trained network, the computer calculates the average loss which tells how close the network got to the actual inverse of the matrix. We repeat this cycle of training and testing the network multiple times which makes the network learn to do the inverse better. The whole idea is for the neural network to learn the inversion as well as possible. This means the error between the actual inverse and the inverse calculated by the neural network is wanted to be as small as possible. So we want the average loss to be small. In this section we work with 2×2 real valued matrices.

When searching the best learning rate we set the limit to 50 epochs just to see the beginning of the learning. After this we use a stopping criterion to define how far the network will be taught. The stopping criterion is when the average loss goes smaller than $9 \cdot 10^{-5}$. This limit is completely arbitrary. By setting the limit to a very small number the inverse calculated by the network is very close to the actual inverse. On the other hand teaching to any smaller average loss than $9 \cdot 10^{-5}$ would take too much time with the capacity we had.

3.3.1 Finding the best learning rate

The first thing we study is the learning rate (”lr” in the code). The goal is to find the learning rate that makes the neural network learn the quickest. With each different learning rate we use four new networks and observe the changes in the average loss. All networks have ten layers and randomly picked parameters at the beginning and they are independent from each other.

We create data sets and use them for all the networks so the results would be comparable. The training set contains 60 000 elements and testing set 10 000 elements. We train the networks for 50 epochs with each learning rate. Learning rates studied were $1 \cdot 10^{-1}, 10^{-2}, 10^{-3}$ and 10^{-4}. Learning rates bigger than $1 \cdot 10^{-1}$ made the networks unstable to learn and were excluded from the comparison. Each group of four networks with studied learning rates took on average 18 minutes each, so one network took 4–5 minutes to go through 50 epochs. The results are in tables 1–4 and in figures 2–5.

Viewing the results we can see there is some variation on how well and how quickly different networks learn. In general the networks learn well with all of the learning rates, but when learning rate goes smaller, learning gets slower for all the networks. In 50 epochs we can see that on average the networks learn the quickest with the learning rate $1 \cdot 10^{-1}$. We will be using this learning rate for all the runs from now on.

lr	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
0.1	0.001766	0.001554	0.001623	0.001542	0.001463
0.1	0.003488	0.002262	0.001613	0.001340	0.001219
0.1	0.001463	0.000659	0.000460	0.000333	0.000251
0.1	0.000987	0.000460	0.000344	0.000292	0.000261

Table 1: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-1}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 2.*

lr	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
0.01	0.003610	0.002943	0.002478	0.002132	0.001819
0.01	0.004851	0.003364	0.002559	0.001932	0.001417
0.01	0.006535	0.004404	0.003274	0.002701	0.002260
0.01	0.004984	0.002832	0.002144	0.001735	0.001458

Table 2: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-2}$*. See figure 3.*

lr	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
0.001	0.053339	0.020258	0.010881	0.007840	0.006353
0.001	0.058097	0.039330	0.030043	0.020909	0.014186
0.001	0.030381	0.011845	0.010171	0.009575	0.009121
0.001	0.050086	0.034006	0.024042	0.015542	0.010670

Table 3: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-3}$*. See figure 4.*

lr	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
0.0001	0.275042	0.169904	0.111861	0.073311	0.054786
0.0001	0.113693	0.080240	0.062574	0.053541	0.047360
0.0001	0.159241	0.073552	0.043187	0.034231	0.030052
0.0001	0.112535	0.101051	0.088696	0.073609	0.059076

Table 4: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-4}$*. See figure 5.*

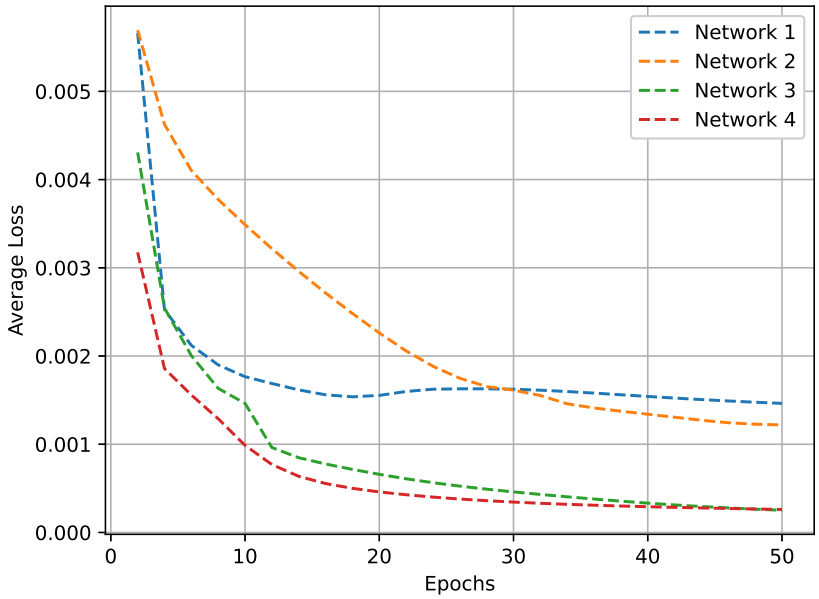

Figure 2: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-1}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 1.*

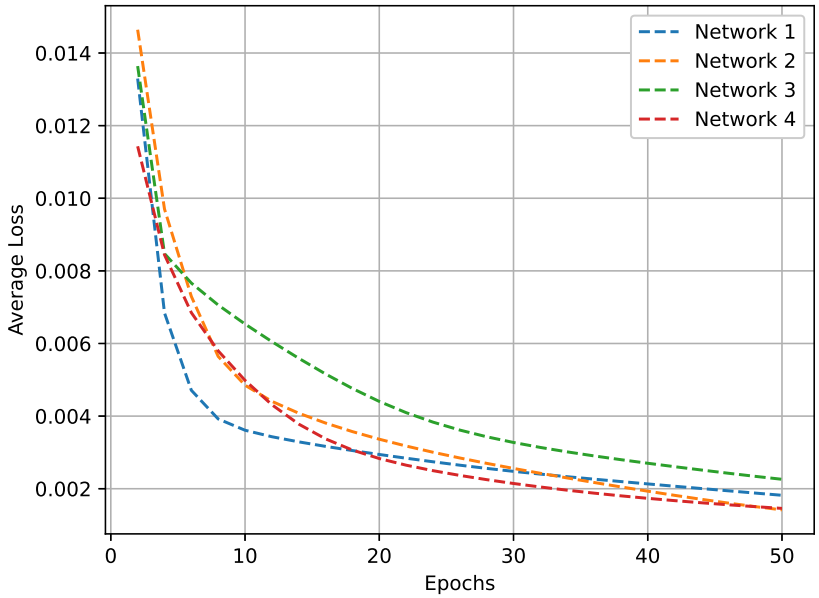

Figure 3: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-2}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 2.*

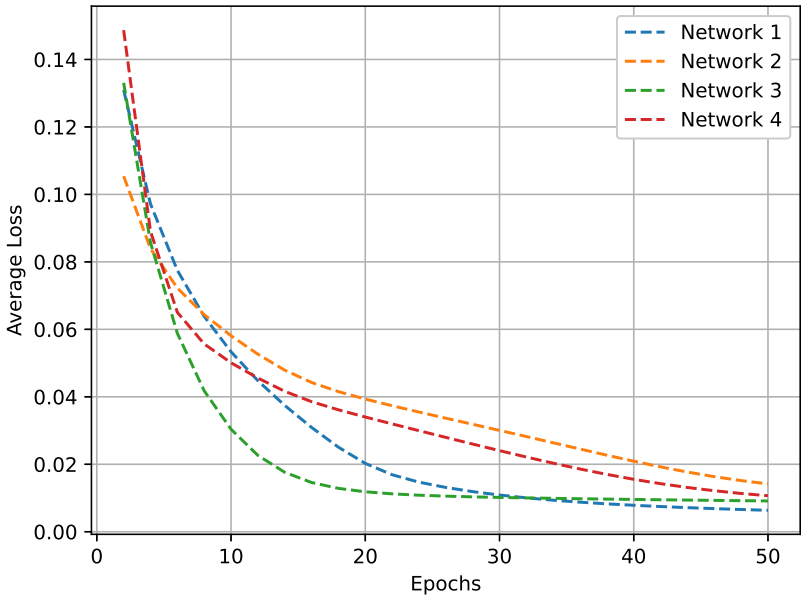

Figure 4: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-3}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 3.*

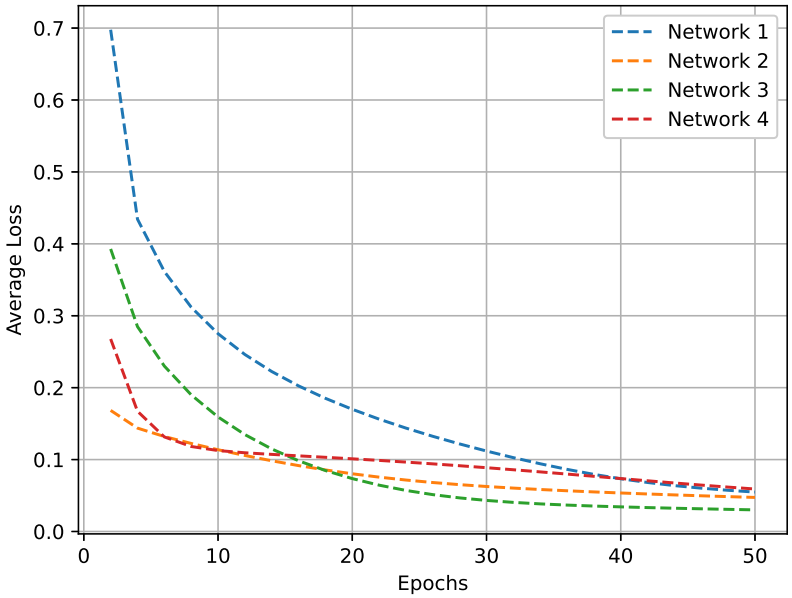

Figure 5: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-4}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 4.*

3.3.2 Teaching until the wanted small average loss

Now we know that a network learns the quickest with the learning rate $1 \cdot 10^{-1}$. Next up we shall see how quickly new untrained networks learn with this learning rate. We switch from 50 epochs to stopping criterion. We kept the data sets the same as before. It took 15–30 minutes on average for a network to learn until the stopping criterion. Results are in the table 5 and in figure 6.

Network	20 epochs	130 epochs	206 epochs	250 epochs	358 epochs
1	0.000890	0.000166	0.000109	0.000089	NaN
2	0.000357	0.000089	NaN	NaN	NaN
3	0.001488	0.000217	0.000134	0.000116	0.00009
4	0.000788	0.000148	0.000090	NaN	NaN

Table 5: *Development of the average loss. Four newly generated networks without ReLU layers trained until the stopping criterion when the average loss is smaller than* $9 \cdot 10^{-5}$. *See figure 6.*

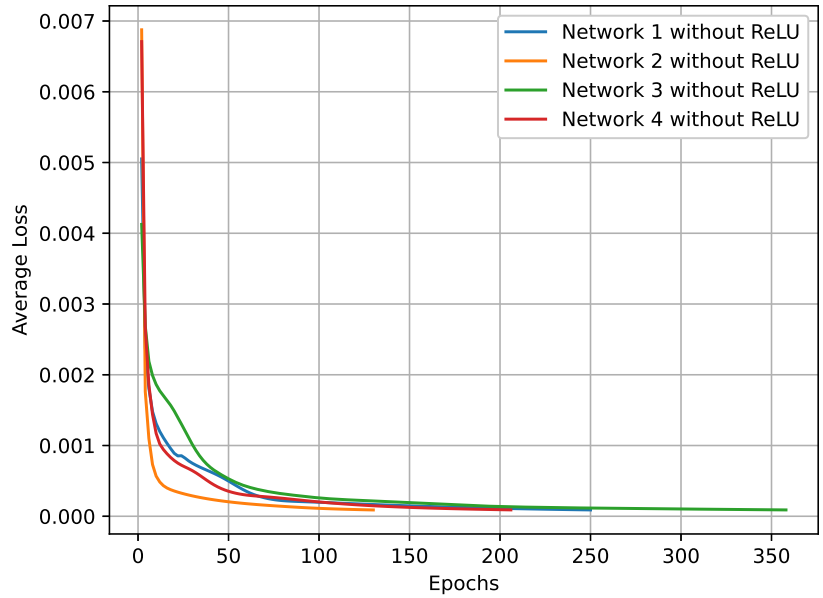

Figure 6: *Development of the average loss. Four newly generated networks without ReLU trained until the stopping criterion when the average loss is smaller than* $9 \cdot 10^{-5}$. *A plot from table 5.*

3.3.3 Adding ReLU layers

The next step is to add ReLU layers to the networks and see how that affects the learning. We update the creation of the neural network in the opnet.py-file by adding ReLU layers to the class OperatorNet. Now the code fully implements the network presented in definition 2.1 of [6]. The new implementation with ReLU layers doubles the amount of parameters so we decrease the number of layers down to five, but when using a network without ReLU we double the number to ten so the comparison is reasonable.

We create two groups of four networks, one group with ReLU layers and the other without. Now we can study the effect of ReLU on the learning. We keep the data sets the same as before and train the networks until the stopping criterion. The hypothesis is that the networks with ReLU learn faster than the networks without ReLU.

Results for networks without ReLU are in table 6 and in figure 7 and for networks with ReLU in table 7 and in figure 8. We can see NaN-values in the tables since the computer stops calculating after the network has reached the stopping criterion and there are no values after that.

For the networks without ReLU it took 10–35 minutes to reach the stopping criterion and for the networks with ReLU it took 10–73 minutes. Based on the results, we cannot see a clear improvement in the learning when ReLU is added. Compared to the networks without ReLU some networks with ReLU learn quicker, but some take much longer. This could be explained with the different initial states of the networks or the possibility that the optimization method may be unstable in networks with ReLU. Also it could be due to the used data sets since they are only a small sample of the whole data space and therefore do not contain all the possible input values. This means that the data sets may not represent the whole space so their distributions might not match with the whole data space. With other and maybe bigger data sets the results could be more even.

Network	20 epochs	124 epochs	190 epochs	200 epochs	396 epochs
1	0.000310	0.000089	NaN	NaN	NaN
2	0.000619	0.000126	0.000092	0.000090	NaN
3	0.000764	0.000187	0.000143	0.000138	0.00009
4	0.000598	0.000136	0.000089	NaN	NaN

Table 6: *Comparing the development of the average loss on networks without ReLU layers. Every network is taught until the stopping criterion when the average loss is smaller than* $9 \cdot 10^{-5}$*. See figure 7.*

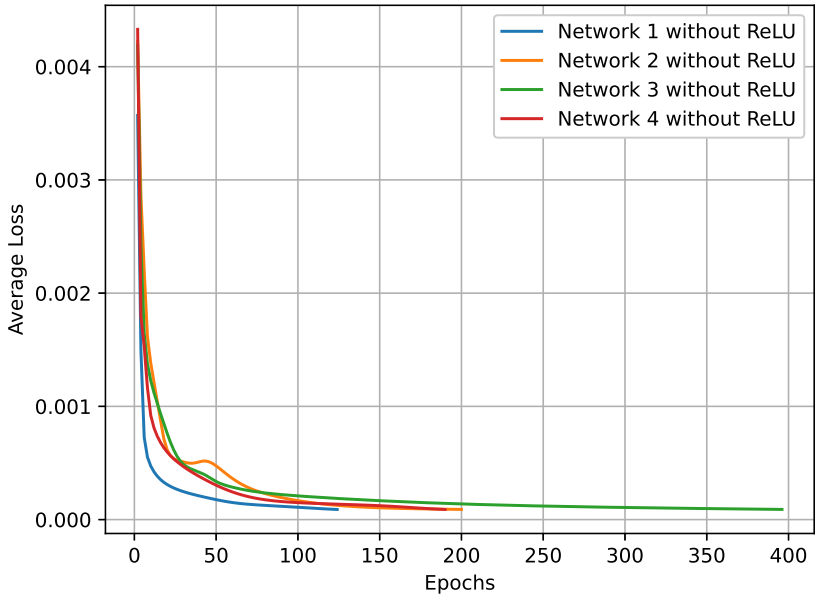

Figure 7: *Development of the average loss. A network without ReLU layers taught until the stopping criterion when the average loss is smaller than* $9 \cdot 10^{-5}$*. Data from table 6.*

Network	20 epochs	100 epochs	380 epochs	780 epochs	836 epochs
1	0.000565	0.000141	0.000090	NaN	NaN
2	0.000375	0.000089	NaN	NaN	NaN
3	0.000478	0.000188	0.000140	0.000087	NaN
4	0.001324	0.000294	0.000118	0.000092	0.00009

Table 7: *Comparing the development of the average loss on networks with ReLU layers. Every network is taught until the stopping criterion when the average loss is smaller than* $9 \cdot 10^{-5}$*. See figure 8.*

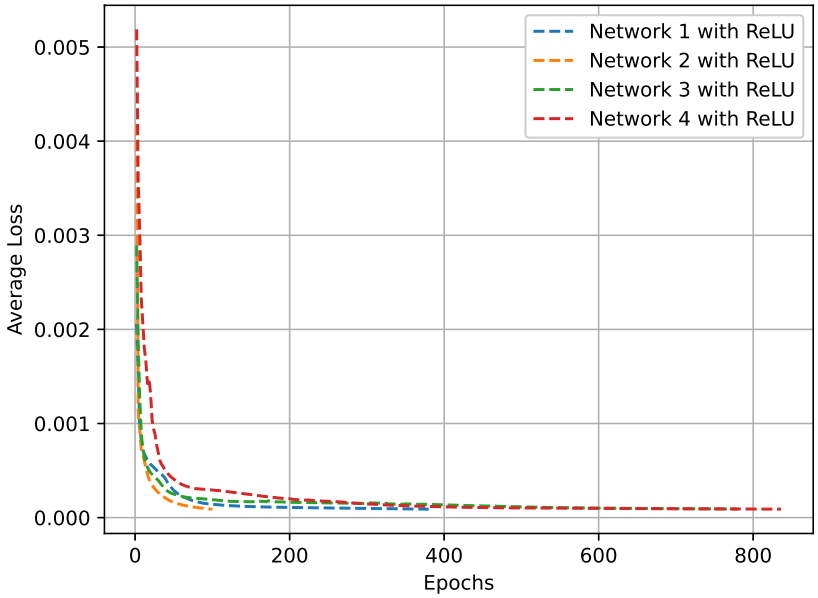

Figure 8: *Development of the average loss. Four networks with ReLU layers taught until the stopping criterion when the average loss is smaller than* $9 \cdot 10^{-5}$*. Data from table 6.*

3.3.4 Changing the sign of eigenvalues

Next we want to study what is the impact of the signs of the eigenvalues of the matrices for the learning of a neural network. Until now the eigenvalues have been picked from the positive interval $[1/2, 3/2]$. We changed the interval to negative $[-3/2, -1/2]$ at *simple_inversion_data.py*, where the data sets are created. Then we created new data sets where the training set contains 60 000 elements and the testing set contains 10 000 elements. Next, two groups of four networks, one with ReLU and one without, were created and trained until the stopping criterion. The assumption is that the networks with ReLU learn faster.

Results for networks without ReLU are in table 8 and in figure 9 and for networks with ReLU in table 9 and in figure 10. For a network without ReLU it took an average of 15–30 minutes to learn, and 25 minutes up to an hour for a network with ReLU. Based on the time networks used learning, the negativity of eigenvalues brought a challenge and surprisingly networks without ReLU handled it better.

Next we modify the code such that every time it creates new data set, it randomly picks a sign for the eigenvalues. It is possible that both eigenvalues are positive or negative, or that one is positive and the other is negative. We create a new data sets. The training set contains 60 000 elements and testing set 10 000 elements. Again, we compare the learning of two groups of four networks, one with ReLU and one without. In this case the used learning rate $1 \cdot 10^{-1}$ is unstable with the networks with ReLU layers. Five out of eight networks did

not give numerical values. Therefore, we had to change the learning rate to $9 \cdot 10^{-2}$. We taught all of the networks until the stopping criterion.

Results for networks without ReLU are in table 10 and in figure 11, and for networks with ReLU in table 11 and in figure 12. All the networks learned slower than in the case when the signs of the eigenvalues are the same. Especially networks with ReLU took a long time to reach the stopping criterion. At the quickest one network with ReLU took 30 minutes, but the others took 60, 75 and 80 minutes. For networks without ReLU it took 20–30 minutes for each to reach the stopping criterion. Based on the results, this change in data made learning more difficult for the networks in general, but again networks without ReLU learned faster.

Overall, the learning in figures 10–12 is very uneven for some networks. This may be due to differences in the initial states and the optimization may also be a bit unstable in these cases. This is an interesting phenomenon that we did not have the time to study further.

Network	50 epochs	126 epochs	184 epochs	236 epochs	430 epochs
1	0.000203	0.000089	NaN	NaN	NaN
2	0.000830	0.000474	0.000212	0.000149	0.00009
3	0.000279	0.000275	0.000154	0.000089	NaN
4	0.000247	0.000135	0.000090	NaN	NaN

Table 8: *Studying the development of the average loss on networks without ReLU layers until the stopping criterion when the eigenvalues of the matrices are negative. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 9.*

Network	100 epochs	264 epochs	278 epochs	574 epochs	676 epochs
1	0.000352	0.000135	0.000129	0.000090	NaN
2	0.000718	0.000267	0.000253	0.000111	0.00009
3	0.000283	0.000097	0.000089	NaN	NaN
4	0.000148	0.000090	NaN	NaN	NaN

Table 9: *Studying the development of the average loss on networks with ReLU layers until the stopping criterion when the eigenvalues of the matrices are negative. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 10.*

Network	1052 epochs	1142 epochs	1482 epochs	1630 epochs
1	0.000115	0.000109	0.000090	NaN
2	0.000090	NaN	NaN	NaN
3	0.000120	0.000112	0.000095	0.00009
4	0.000102	0.000090	NaN	NaN

Table 10: *Studying the development of the average loss on networks without ReLU layers until the stopping criterion when the signs of the eigenvalues of the matrices are randomly picked. The learning rate is* $1 \cdot 10^{-1}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 11.*

Network	500 epochs	1804 epochs	3652 epochs	4686 epochs	5356 epochs
1	0.000495	0.000090	NaN	NaN	NaN
2	0.000374	0.000194	0.000090	NaN	NaN
3	0.000760	0.000252	0.000160	0.000081	NaN
4	0.000271	0.000168	0.000127	0.000108	0.00009

Table 11: *Studying the development of the average loss on networks with ReLU layers until the stopping criterion when the signs of the eigenvalues of the matrices are randomly picked. The learning rate is* $9 \cdot 10^{-2}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 12.*

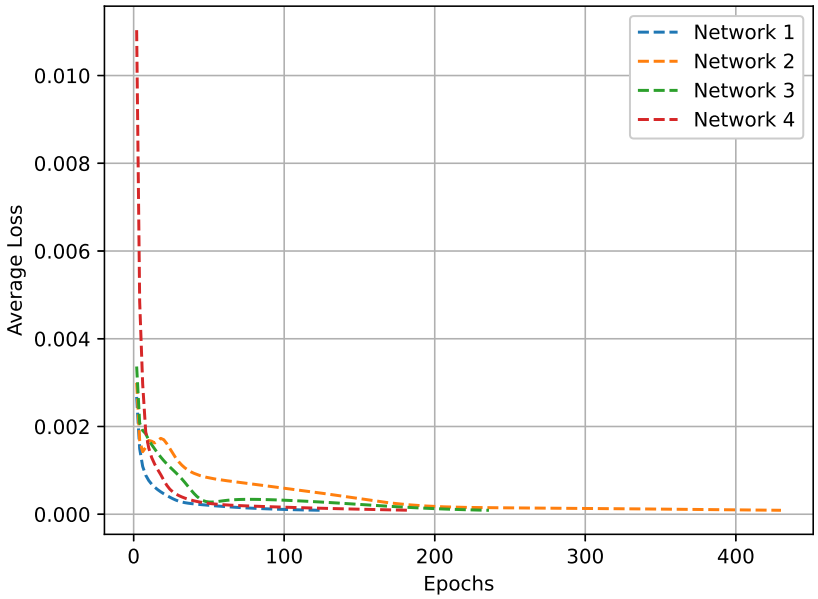

Figure 9: *Learning of the networks without ReLU layers until the stopping criterion when the eigenvalues of the matrices are negative. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. Data from table 8.*

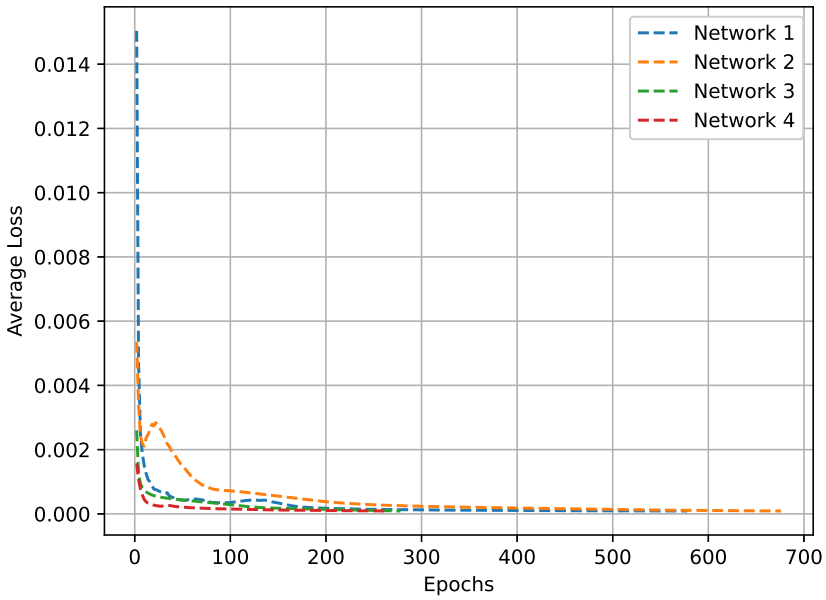

Figure 10: *Learning of the networks with ReLU layers until the stopping criterion when the eigenvalues of the matrices are negative. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. Data from table 9.*

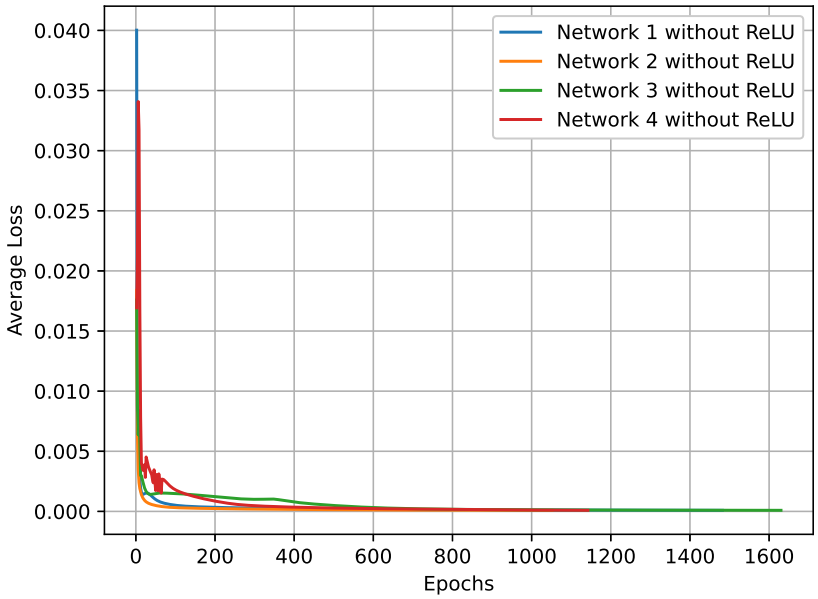

Figure 11: *Learning of the networks without ReLU layers until the stopping criterion when the signs of the eigenvalues of the matrices are randomly picked. The learning rate is* $1 \cdot 10^{-1}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. Data from table 10.*

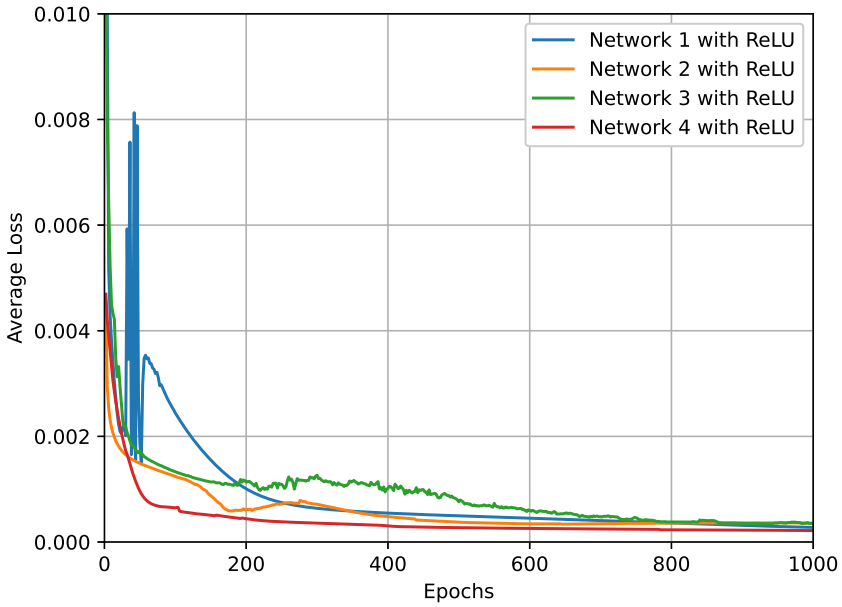

Figure 12: *Learning of the networks with ReLU layers until the stopping criterion when the signs of the eigenvalues of the matrices are randomly picked. The learning rate is* $9 \cdot 10^{-2}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. Data from table 11.*

4 The inverse problem of the wave equation

In this chapter we move from square matrices to the one dimensional wave equation. First we address the general form of the equation and prove there is a general solution for it [12]. Then we look at the inverse problem of the wave equation and prove there is a unique solution for it [10].

4.1 The general form and solution of the wave equation

The wave equation in one dimensional space has the form

$$\partial_t^2 u - c^2 \partial_x^2 u = 0, \quad -\infty < x < \infty, \tag{41}$$

where $t > 0$ is the time, x is the variable of one dimensional space and $c \in \mathbb{R}^+$ is the non-zero wave speed.

To obtain the general solution to this equation let's first define two variables as

$$\xi = \xi(x,t) = x - ct, \tag{42}$$
$$\eta = \eta(x,t) = x + ct. \tag{43}$$

With the help of these *characteristic coordinates* we can write our function u in a new way as

$$u(x,t) = \tilde{u}(\xi(x,t), \eta(x,t)) = \tilde{u}(\xi, \eta) \tag{44}$$
$$= u\left(\frac{\xi+\eta}{2}, \frac{\xi-\eta}{2c}\right) = u(x(\xi,\eta), t(\xi,\eta)), \tag{45}$$

where $x = (\xi + \eta)/2$ and $t = (\xi - \eta)/2c$. Now we want to differentiate the function u using the chain rule. First we differentiate twice with respect to x

$$\partial_x u = \partial_\xi \tilde{u} \partial_x \xi + \partial_\eta \tilde{u} \partial_x \eta = \partial_\xi \tilde{u} + \partial_\eta \tilde{u} = (\partial_\xi + \partial_\eta)\tilde{u}, \tag{46}$$

where $\partial_x \xi = 1 = \partial_x \eta$.

$$\begin{aligned} \partial_x^2 u &= \partial_x \partial_x u = \partial_x (\partial_\xi + \partial_\eta)\tilde{u} \\ &= (\partial_\xi + \partial_\eta)(\partial_\xi + \partial_\eta)\tilde{u} \\ &= (\partial_\xi^2 - 2\partial_{\xi\eta} + \partial_\eta^2)\tilde{u}. \end{aligned} \tag{47}$$

Next we differentiate twice with respect to t

$$\partial_t u = \partial_\xi \tilde{u} \partial_t \xi + \partial_\eta \tilde{u} \partial_t \eta = c\partial_\xi \tilde{u} - c\partial_\eta \tilde{u} = c(\partial_\xi + \partial_\eta)\tilde{u}, \tag{48}$$

where $\partial_t \xi = c$ and $\partial_t \eta = -c$.

$$\begin{aligned} \partial_t^2 u &= \partial_t \partial_t u = \partial_t c \left(\partial_\xi + \partial_\eta\right) \tilde{u} \\ &= c \left(\partial_\xi + \partial_\eta\right) c \left(\partial_\xi + \partial_\eta\right) \tilde{u} \\ &= c^2 \left(\partial_\xi^2 - 2\partial_{\xi\eta} + \partial_\eta^2\right) \tilde{u}. \end{aligned} \tag{49}$$

Now substituting these into the wave equation we get

$$\begin{aligned} \partial_t^2 u - c^2 \partial_x^2 u &= c^2 \partial_\xi^2 \tilde{u} - c^2 2\partial_{\xi\eta} \tilde{u} + c^2 \partial_\eta^2 \tilde{u} - c^2 \partial_\xi^2 \tilde{u} - c^2 2\partial_{\xi\eta} \tilde{u} - c^2 \partial_\eta^2 \tilde{u} \\ &= -c^2 4 \partial_{\xi\eta} \tilde{u} = 0. \end{aligned} \tag{50}$$

So the problem comes down to the equation

$$-c^2 4\partial_{\xi\eta}\tilde{u} = 0. \tag{51}$$

We know that $-4c^2 \neq 0$ so it has to be

$$\partial_{\xi\eta}\tilde{u} = 0. \tag{52}$$

We can solve $\tilde{u}$ by integrating twice. From the equation (52) we observe that derivative of $\tilde{u}$ with respect to ξ does not depend on η and vice versa. So by deriving some function $g(\eta)$ with respect to ξ we have

$$\partial_\xi g(\eta) = 0 \tag{53}$$

and from here we can denote

$$\partial_\eta \tilde{u}(\xi, \eta) = g(\eta). \tag{54}$$

Then integrating with respect to η we get

$$\int \partial_\eta \tilde{u}\, d\eta = \int g(\eta)\, d\eta + F. \tag{55}$$

The function g is a derivative of some function G so we have $g(\eta) = G'$. F is a constant with respect to η so we can write it as a function of ξ as $F(\xi)$. And we get the general form of the solution to the one dimensional wave equation

$$\tilde{u}(\xi, \eta) = F(\xi) + G(\eta) \tag{56}$$

$$\Rightarrow u(x, t) = F(x + ct) + G(x - ct), \tag{57}$$

where $F, G \in C^2(\mathbb{R})$ are arbitrary functions.

4.2 Piecewise wave speed function

In the previous section we assumed that the wave speed c was a constant, see equation (41). In this section we consider the case where the wave speed is a one-dimensional piecewise constant function $c = c(x)$. That is how we use it in our code [8].

If we start with the wave speed being $c = 1$ for all $(x, t) \in \mathbb{R}^2$, our wave equation has the form

$$\partial_t^2 u - \partial_x^2 u = 0. \tag{58}$$

Let's suppose we know that $u(x, t) = f(x - t)$ and $\partial_t u(x, t) = g(x - t)$ when $t < 0$. By chapter 4.4 we know that with these initial conditions the wave equation (58) has a solution that is unique. It follows from the fact that the wave propagation has finite speed.

Now, if our wave speed is a piecewise constant function

$$c(x) = \begin{cases} 1, & x < x_0 \\ 1/2, & x > x_0, \end{cases} \tag{59}$$

it has discontinuity point at $x = x_0$. When a wave comes to this point it splits into two waves, reflected and transmitted wave. To get the general solution in

this case, we will start with amplitudes a_j and delays d_j. Lets construct the function u considering the change in the wave speed. Before discontinuity point, when $x < x_0$, our function equals to $u(x,t) = f(x-t)$ and is zero elsewhere. After the splitting there is a reflected wave travelling in the area of $x < x_0$ with some amplitude a_1 and some delay d_1 which are different from the amplitude and delay of the original wave. We get

$$\begin{aligned} u(x,t) &= f(x/c - t) + a_1 f(-x/c - t - d_1) \\ &= f(x-t) + a_1 f(-x - t - d_1), \quad \text{when } x < x_0. \end{aligned}$$

After the splitting there is also a transmitted wave travelling in the area of $x > x_0$ with amplitude a_2 and delay d_2, so we get

$$u(x,t) = a_2 f(x/c - t - d_2) = a_2 f(2x - t - d_2), \quad \text{when } x > x_0.$$

All in all our solution function has the form of

$$u(x,t) = \begin{cases} f(x-t) + a_1 f(-x-t-d_1), & x < x_0 \\ a_2 f(2x - t - d_2), & x > x_0. \end{cases} \tag{60}$$

To make this function and its derivative continuous everywhere, we have to require u and $\partial_x u$ to be continuous at $x = x_0$ and $t > 0$ for all functions f. So the solution function $u(x,t)$ and its partial derivatives must be equal at this point, so we have

$$\begin{cases} f(x_0 - t) + a_1 f(-x_0 - t - d_1) = a_2 f(2x_0 - t - d_2) \\ \partial_x (f(x_0 - t) + a_1 f(-x_0 - t - d_1)) = \partial_x (a_2 f(2x_0 - t - d_2)). \end{cases} \tag{61}$$

And from here we get the conditions for the parameters a_j and d_j, $j = 1, 2$, as

$$\begin{cases} x_0 - t = -x_0 - t - d_1 = 2x_0 - t - d_2 \\ 1 + a_1 = a_2 \\ 1 - a_1 = 2a_2. \end{cases}$$

Solving this system of equations we get that $a_1 = -1/3$ and $a_2 = 2/3$. Parameters d_j depend on x_0 since $d_1 = -2x_0$ and $d_2 = x_0$.

Example 1. We take the function (59) and put $x_0 = 1$. Then conditions for parameters a_j and d_j takes the form of

$$\begin{cases} 1 - t = -1 - d_1 = 2 - d_2 \\ 1 + a_1 = a_2 \\ 1 - a_1 = 2a_2. \end{cases}$$

Parameters a_j stay the same as they don't depend on x, but d_j will have more precise forms now, $d_1 = -2$ and $d_2 = 1$.

4.3 Initial and boundary values

It is usual to have some prior knowledge about the wave equation problem. If we have information about the beginning point, we have an initial value problem. If we also have information about the ending point, we have a boundary value problem.

In the initial value problem we have information only about the beginning state at time $t = 0$. For example we could have a given $f \in C^3(\mathbb{R}^n)$ and $g \in C^2(\mathbb{R}^n)$ and the problem would be

$$\begin{cases} \partial_t^2 u - c^2 \partial_x^2 u = 0 & \\ u(x,0) = f & \text{in } \mathbb{R}^n \\ \partial_t u(x,0) = g & \text{in } \mathbb{R}^n. \end{cases} \tag{62}$$

If the dimension $n \geq 3$, the initial problem (62) can be solved by Poisson method of spherical means, and if $n = 2$ by Hadamard method of descent.

An example of a boundary value problem is a string that is attached from its end points $x = 0$ and $x = L$. The string vibrates and the function $u(x,t)$ denotes the vertical displacement at the time t of the point $x \in (0, L)$. We know the shape and the speed of the string at the beginning, at time $t = 0$, and they are given as $f, g \in C^2[0, L]$. Then we can write the boundary value problem as

$$\begin{cases} \partial_t^2 u - c^2 \partial_x^2 u = 0 & \text{in } (0,L) \times \mathbb{R} \\ u(0,t) = u(L,t) = 0 & \text{in } \mathbb{R} \\ u(x,0) = f,\ \partial_t u(x,0) = g & \text{in } (0,L). \end{cases} \tag{63}$$

The functions f and g have to satisfy the compatibility conditions

$$f(0) = f(L) = g(0) = g(L) = 0.$$

A solution to the boundary value problem can be found by using the solution of Cauchy problem [1, p. 185].

4.4 The inverse problem and its unique solution

We denote $\mathbb{R}^+ := (0, \infty)$. The boundary value problem

$$\begin{cases} \partial_t^2 u - c(x)^2 \partial_x^2 u = 0, & \text{in } (0, 2T) \times \mathbb{R}^+ \\ \partial_x u(0,t) = f(t) & \\ u(x,0) = \partial_t u(x,0) = 0 & \end{cases} \tag{64}$$

where $f \in C_0^\infty(\mathbb{R}^+)$ and the wave speed $c(x)$ is strictly positive on $\overline{\mathbb{R}^+}$, has a unique solution $u^f \in H^1((0,2T) \times \mathbb{R}^+)$, [10]. Define the Neumann-to-Dirichlet operator as

$$\Lambda : L^2(0,2T) \to L^2(0,2T),\ \Lambda f = u^f(0,t). \tag{65}$$

Next we define the space of velocity functions. Let $C_0, C_1, S, m > 0$ and then

$$\begin{aligned} \mathcal{D}(\mathcal{A}) := \{ & c \in L^\infty(\mathbb{R}^+) \mid C_0 \leq c(x) \leq C_1, \\ & ||c||_{C^2(\mathbb{R}^+)} \leq m, c - 1 \in C_0^2(0,S) \}, \end{aligned} \tag{66}$$

and also set $T > \dfrac{S}{C_0}$. Let E be a Banach space and define

$$\mathcal{L}(E) = \{A : E \to E \mid \text{A is linear and continuous}\}. \tag{67}$$

Let $X = L^\infty(\mathbb{R}^+)$ and $Y = \mathcal{L}(L^2(0, 2T))$ and now define the direct map

$$\mathcal{A} : \mathcal{D}(\mathcal{A}) \subset X \to Y,\ \mathcal{A}(c(x)) = \Lambda. \tag{68}$$

Our inverse problem is to recover the wave speed $c(x)$ when we know the boundary measurements Λ. So we have $\mathcal{A}^{-1}(\Lambda) = c(x)$. The continuity of (65) and (68) is proved in [10].

We start solving the problem with the fact that the wave propagation has finite speed. This means that $u(x, t) = 0$ for $x \geq r(t)$, with some smooth and increasing function $r(t) \in \mathbb{R}$. We want to find the optimal $r(t)$ that satisfies the given requirements. Set the measurement error as

$$\mathcal{E}(x, t) = c^{-2}|\partial_t u(x, t)|^2 + |\partial_x u(x, t)|^2 \tag{69}$$

and the energy of the wave above the function $r(t)$ as

$$E(t) = \frac{1}{2}\int_{r(t)}^{\infty} \mathcal{E}(x, t)dx, \tag{70}$$

which can also be written as

$$E(t) = E(0) + \int_0^t \partial_t E(s)ds. \tag{71}$$

We will show below that $\partial_t E(t) \leq 0$. Then also $E(t) \leq 0$. This implies that $E(t) = 0$, which gives us $u(x, t) = 0$ for $x \geq r(t)$. So our goal is to find $r(t)$ such that $\partial_t E(t) \leq 0$.

Lets investigate the $\partial_t E(t)$ a little more and calculate it open using Leibniz integral rule

$$\begin{aligned}
\partial_t E(t) &= \frac{1}{2}\int_{r(t)}^{\infty} \partial_t \mathcal{E}(x, t)dx && (72)\\
&= -\frac{1}{2}\partial_t r(t)\mathcal{E}(r(t), t) + \frac{1}{2}\int_{r(t)}^{\infty} \partial_t \mathcal{E}(x, t)dx && (73)\\
&= -\frac{1}{2}\partial_t r(t)\mathcal{E}(r(t), t) + \int_{r(t)}^{\infty} c^{-2}(x)\partial_t u\partial_t^2 u + \partial_x u\partial_{tx} u\ dx. && (74)
\end{aligned}$$

Integrating by parts the integral in (74) we get

$$\begin{aligned}
&\int_{r(t)}^{\infty} c^{-2}(x)\partial_t u\partial_t^2 u + \partial_x u\partial_{tx} u\ dx && (75)\\
&= \int_{r(t)}^{\infty} c^{-2}(x)\partial_t u\partial_t^2 u\ dx + \int_{r(t)}^{\infty} \partial_x u\partial_{tx} u\ dx && (76)\\
&= \int_{r(t)}^{\infty} c^{-2}(x)\partial_t u\partial_t^2 u\ dx + [\partial_x u\partial_t u]_{r(t)}^{\infty} - \int_{r(t)}^{\infty} \partial_x^2 u\partial_t u\ dx && (77)\\
&= \int_{r(t)}^{\infty} (c^{-2}(x)\partial_t^2 u - \partial_x^2 u)\partial_t u\ dx + [\partial_x u\partial_t u]_{r(t)}^{\infty}. && (78)
\end{aligned}$$

The integral in (78) equals to zero since u is the solution of the wave equation. For the second term, to keep the argument simple, let's assume that since the finite speed of wave propagation the boundary terms vanish at infinity. This is proven in [2, p. 84] at theorem 6, but only for a constant wave speed. If we assume that there exists a maximum of $c(x)$, then the the boundary terms have to vanish at infinity. So we can mark $\tilde{c} = \max(c(x))$ and then the proof holds for our situation too and the second term in (78) vanishes at infinity. Now our equation in (72) shrinks down to

$$\partial_t E(t) = -\frac{1}{2}\partial_t r(t)\mathcal{E}(r(t), t) - (\partial_x u \partial_t u)|_{x=r(t)}. \tag{79}$$

We want the first and second terms to cancel each other out. If we have

$$0 \leq (x+y)^2 = x^2 + 2xy + y^2 \tag{80}$$

$$\Rightarrow -xy \leq \frac{1}{2}(x^2 + y^2) \tag{81}$$

and with $a \in \mathbb{R}^+$

$$\frac{\sqrt{a}}{\sqrt{a}}(-xy) = -(\frac{x}{\sqrt{a}})(\sqrt{a}y) \leq \frac{1}{2}(a^{-1}x^2 + ay^2), \tag{82}$$

then we can use this to our second term in (79) as follows

$$-(\partial_x u \partial_t u)|_{x=r(t)} \leq \frac{1}{2}(a^{-1}|\partial_x u|^2 + a|\partial_t u|^2).$$

We are at a point $x = r(t)$. The first and second term cancel each other only when $a^{-1} = r'$ and $a = r'c^{-2}$. Then

$$\begin{aligned} & r'(t)c^{-2} = a = 1/r'(t) \\ & \Leftrightarrow (r'(t))^2 = c^2(r(t)) \\ & \Leftrightarrow r'(t) = c(r(t)). \end{aligned} \tag{83}$$

Now we have an equation for $r(t)$. We will solve this by first defining the following function

$$\rho(x) = \int_0^x \frac{1}{c(y)} dy.$$

Function $\rho(x)$ is strictly increasing and therefore it has inverse function. We can set

$$r(t) = \rho^{-1}(t), \tag{84}$$

since this satisfies the equation (83) as we can see

$$r'(t) = \frac{1}{\rho'(r(t))} = \frac{1}{1/c(r(t))} = c(r(t)). \tag{85}$$

With this choice of $r(t)$ we have $\partial_t E \leq 0$ and the finite speed of propagation holds.

Next we will give a control problem that we will use as a tool to solve the inverse problem. The idea of the control problem is that we can control the solution. First let us define an inner product for functions in L^2 space as

$$(\phi, \psi)_{L^2(\mathbb{R}^+, c^{-2})} = \int_0^\infty \phi(x)\psi(x)c^{-2}(x)dx. \tag{86}$$

Proposition 1. *For any function $\phi \in L^2([0,T])$ with $\phi(x) = 0$ for $x > r(T)$, there exists $f \in L^2([0,T])$ such that*

$$u^f(x,T) = \phi(x). \tag{87}$$

Proof. The proof is introduced in [9] but in more general situation. With the help of chapter 1.2 in [9] we can write the wave equation (64) the following way

$$\partial_t^2 u - m^{-1} g^{-1/2} \partial_x (m g^{-1/2} \partial_x u) + cu = 0. \tag{88}$$

To get our form of the wave equation from (88) we want $m^{-1}g^{-1/2} = c^2$ and $mg^{-1/2} = 1$. By solving these equations we get that $m = 1/c$ and $g = 1/c^2$. Now the proof of Lemma 1.15 in [9] holds for our situation. □

Next let us define a function W as follows

$$\begin{aligned} W(t,s) :=& (u^f(\cdot,t), u^f(\cdot,s))_{L^2(\mathbb{R}^+, c^{-2})} \\ =& \int_0^\infty u^f(x,t) u^f(x,s) c^{-2}(x) dx. \end{aligned} \tag{89}$$

Proposition 2. *Operator Λ determines function W.*

Proof. For the simplicity we write $u = u^f$. By using the integration by parts we can see that the boundary terms at infinity are vanishing. First, since u is the solution of the wave equation we have that $\partial_t^2 u = c^2 \partial_x^2 u$ and we can write

$$\begin{aligned} (\partial_t^2 - \partial_s^2) W(t,s) &= \partial_t^2 W(t,s) - \partial_s^2 W(t,s) \\ &= (\partial_t^2 u(\cdot,t), u(\cdot,s))_{L^2(\mathbb{R}^+, c^{-2})} - (u(\cdot,t), \partial_s^2 u(\cdot,s))_{L(\mathbb{R}^+, c^{-2})} \\ &= (c^2 \partial_x^2 u(\cdot,t), u(\cdot,s))_{L^2(\mathbb{R}^+, c^{-2})} - (u(\cdot,t), c^2 \partial_x^2 u(\cdot,s))_{L(\mathbb{R}^+, c^{-2})} \\ &= (\partial_x^2 u(\cdot,t), u(\cdot,s))_{L^2(\mathbb{R}^+)} - (u(\cdot,t), \partial_x^2 u(\cdot,s))_{L(\mathbb{R}^+)}. \end{aligned}$$

And integrating by parts we get

$$\begin{aligned} &= [u(\cdot,s)\partial_x u(\cdot,t)]_0^\infty - \int_0^\infty \partial_x u(\cdot,s) \partial_x u(\cdot,t) dx \\ &- [u(\cdot,t)\partial_x u(\cdot,s)]_0^\infty + \int_0^\infty \partial_x u(\cdot,s) \partial_x u(\cdot,t) dx \\ &= [u(\cdot,s)\partial_x u(\cdot,t)]_0^\infty - [u(\cdot,t)\partial_x u(\cdot,s)]_0^\infty. \end{aligned}$$

The function u is zero after some $0 < T < \infty$ so at infinity the function u is also zero. Thus

$$(\partial_t^2 - \partial_s^2) W(t,s) = u(0,t)\partial_x u(0,s) - u(0,s)\partial_x u(0,t).$$

Since we have $u(x,0) = 0$ and $\partial_t u(x,0) = 0$ we get $W(0,s) = 0$ and $\partial_t W(0,s) = 0$. Now we see that W is the solution of the following initial value problem

$$\begin{cases} (\partial_t^2 - \partial_s^2) W(t,s) = f(t)\Lambda f(s) - f(s)\Lambda f(t) \\ W(0,s) = \partial_t W(0,s) = 0. \end{cases} \tag{90}$$

□

In similar way the operator Λ determines the inner product $(u^f(\cdot,t),1)_{L^2(\mathbb{R}^+,c^{-2})}$ as follows

$$\begin{aligned}\partial_t^2(u^f(\cdot,t),1)_{L^2(\mathbb{R}^+,c^{-2})} &= (\partial_t^2 u(\cdot,t),1)_{L^2(\mathbb{R}^+,c^{-2})}\\ &= (\partial_x^2 u(\cdot,t),1)_{L^2(\mathbb{R}^+)}\\ &= \int_0^\infty \partial_x^2 u(x,t)dx\\ &= [\partial_x u(x,t)]_0^\infty\\ &= -\partial_x u(0,t)\\ &= -\Lambda f(t).\end{aligned}$$

And hence we can recover also

$$\begin{aligned}&||u^f(\cdot,T)-1||^2_{L^2(\mathbb{R}^+,c^{-2})} - ||1||^2_{L^2(\mathbb{R}^+,c^{-2})}\\ &= ||u^f(\cdot,T)||^2_{L^2(\mathbb{R}^+,c^{-2})} - 2(u^f(\cdot,T),1)_{L^2(\mathbb{R}^+,c^{-2})}.\end{aligned} \tag{91}$$

Recall that u^f satisfies (64) where f appears as the source. We want to minimize equation (91). For this we use Proposition 1 and find such f that gives us u^f as

$$u^f(x,T) = \begin{cases}1, & x \le r(T)\\ 0, & otherwise.\end{cases} \tag{92}$$

This form of function u^f is a minimizer. Then we define a volume as

$$\begin{aligned}(u^f(\cdot,T),1)_{L^2(\mathbb{R}^+,c^{-2})} &= \int_{x<r(T)} 1\cdot c^{-2}(x)dx\\ &= \int_0^{r(T)} c^{-2}(x)dx =: V(T).\end{aligned} \tag{93}$$

When we differentiate this we obtain

$$V'(t) = r'(t)c^{-2}(r(t)) = \frac{c(r(t))}{c^2(r(t))} = \frac{1}{c(r(t))},$$

since $r'(t) = c(r(t))$. Denote that

$$c(r(t)) = \frac{1}{V'(t)}. \tag{94}$$

Now we have recovered $c(x)$ in "wrong" coordinates, the time travel coordinates.

To find $c(x)$ in right coordinates let's mark $c(r(t)) = g(t)$ and recall that

$$c(x) = c(\rho^{-1}(\rho(x))) = c(r(\rho(x))) = g(\rho(x)).$$

To solve this we first need to calculate $r(t)$. Recall from (85) that $r'(t) = c(r(t)) = g(t)$. Thus

$$r(t) = \int_0^t r'(s)ds = \int_0^t g(s)ds.$$

Then we get $\rho = r^{-1}$ and from there we can solve $c(x) = g(\rho(x))$.

5 Operator recurrent neural network for the inverse of the wave equation

The initial set-up is the same as in chapter 3.3 where we modified the code for the matrix inversion. Once again we want the neural networks to learn the inversion as well as possible. This means we want the error between the actual inverse and the inverse calculated by the neural network to be as small as possible. The code is improved with the help of [3].

In this section we concentrate on a subproblem related to the inverse problem of the wave equation to find the wave speed $c(x)$ as written in section 2 of [10]. We start by finding the best learning rate and for this we only want to see how the networks start to learn with different learning rates to get the idea. That is why we run 50 epochs with each network. After the best learning rate is found we switch to the stopping criterion. We set the same stopping criterion as in the matrix inversion, the average loss has to go smaller than $9 \cdot 10^{-5}$.

The data of the wave equation take more memory than in the case of matrix inversion, so we had to drop the amount of elements in the data sets to make it manageable for our computer. We generated a training set containing 10 000 elements and testing set containing 2 500 elements. We trained all of the networks with these data sets. This way the results are comparable.

5.1 Finding the best learning rate

Like in section 3.3.1 we are going to start with finding the best learning rate. With each of the learning rates we trained four new networks and observed the changes in the average loss. All of the networks have ten layers and randomly picked parameters at the beginning. These parameters are independent from each other. Each of the networks is taught 50 epochs.

We studied learning rates $1 \cdot 10^{-1}$,10^{-2},10^{-3} and 10^{-4}. All the other values of learning rate worked well except for $1 \cdot 10^{-1}$. This was unstable and did not produce any values. Everything between $1 \cdot 10^{-2}$ and $1 \cdot 10^{-1}$ was unstable and were removed from the comparison.

Results are in tables 12–14 and in figures 13–15. Training four networks with each of the learning rates took 20 minutes on average. So teaching one neural network with some learning rate takes approximately 5 minutes.

Based on the results there is some variation between how quickly networks learn. All the learning rates help networks to learn, but as in section 3.3.1 we also see here that as the learning rate gets smaller the learning gets slower for all the networks. In 50 epochs we see that on average the networks learn the quickest with the learning rate $1 \cdot 10^{-2}$.

Network	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
1	0.025264	0.013921	0.010738	0.009030	0.007798
2	0.069024	0.032439	0.018876	0.014757	0.012109
3	0.038485	0.022752	0.016397	0.012319	0.009525
4	0.056649	0.034920	0.025393	0.020453	0.017084

Table 12: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-4}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 13.*

Network	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
1	0.005307	0.003073	0.002180	0.001696	0.001399
2	0.006279	0.003001	0.002189	0.001788	0.001534
3	0.004777	0.002557	0.001904	0.001548	0.001321
4	0.007543	0.004039	0.002751	0.002141	0.001794

Table 13: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-3}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 14.*

Network	10 epochs	20 epochs	30 epochs	40 epochs	50 epochs
1	0.000806	0.000453	0.000328	0.000260	0.000218
2	0.000829	0.000471	0.000335	0.000264	0.000221
3	0.000978	0.000633	0.000473	0.000374	0.000308
4	0.000718	0.000450	0.000324	0.000250	0.000204

Table 14: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-2}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 15.*

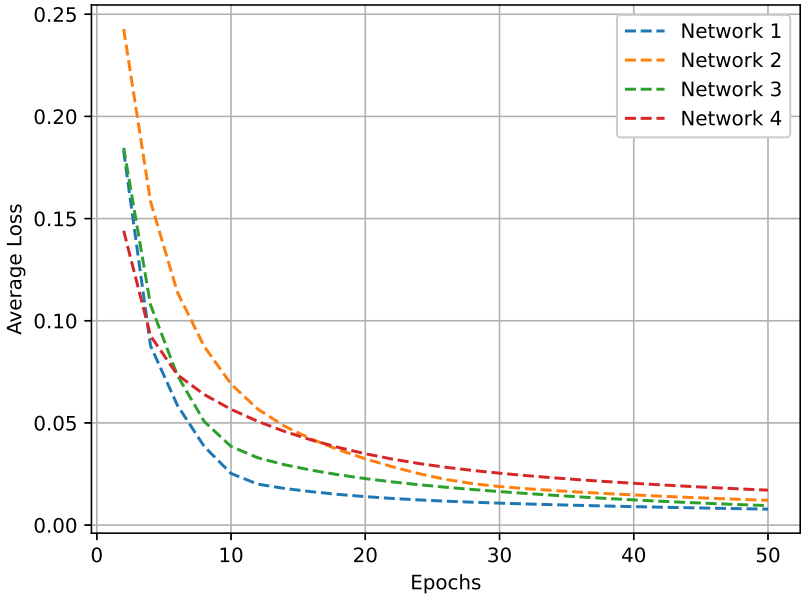

Figure 13: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-4}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 12.*

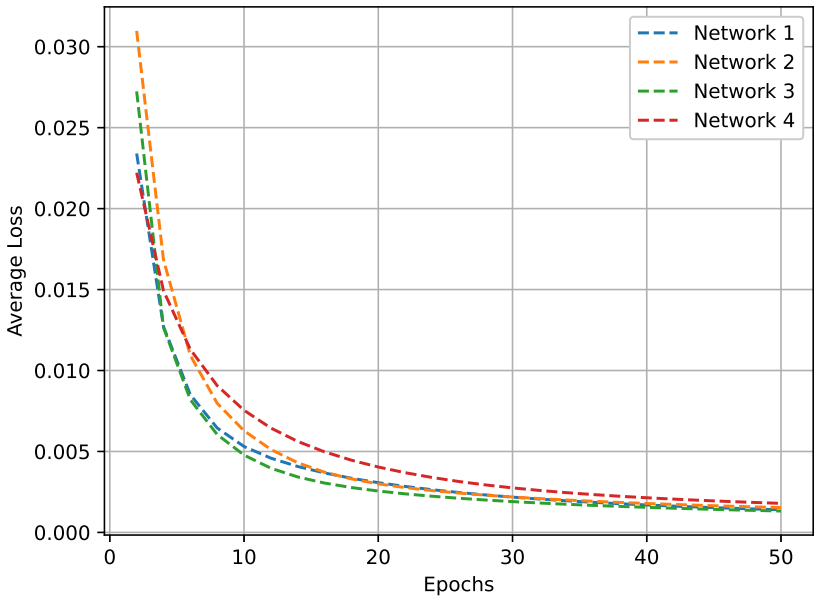

Figure 14: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-3}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 13.*

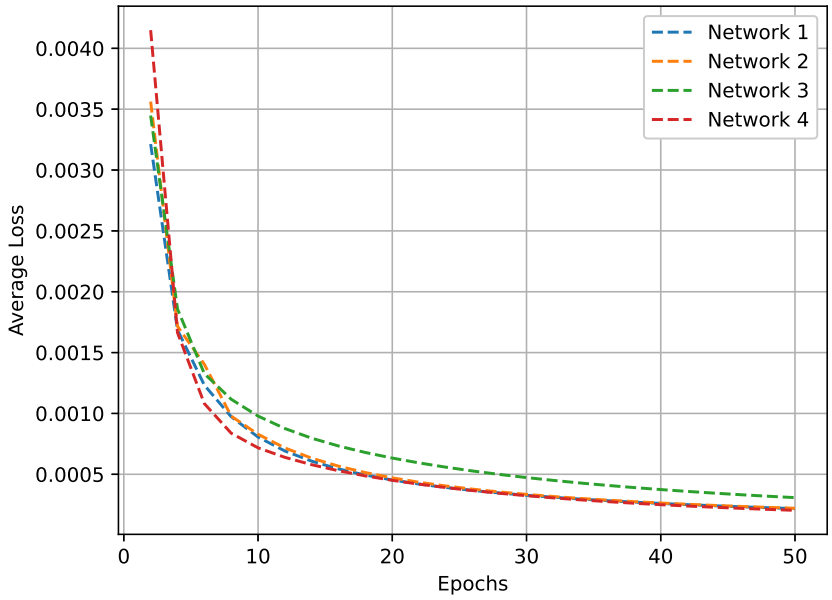

Figure 15: *Studying the development of the average loss on networks without ReLU when the learning rate is* $1 \cdot 10^{-2}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 14.*

5.2 Teaching until the wanted small average loss

We learned in the previous section that networks learn best with the learning rate $1 \cdot 10^{-2}$. We continue with that from now on. Next up we want to see how quickly an untrained network learns. We generate four new networks and teach them until the stopping criterion.

Results are in table 15 and in figure 16. The learning was quicker than expected. The first network took 200 epochs and 19 minutes to meet the stopping criterion. The second network took 130 epochs and 10 minutes. The third network took 166 epochs and 12 minutes. The last network took 200 epochs and 14 minutes.

Network	10 epochs	50 epochs	100 epochs	150 epochs	200 epochs
1	0.000918	0.000314	0.000173	0.000118	0.000089
2	0.000841	0.000212	0.000113	NaN	NaN
3	0.000826	0.000239	0.000143	0.000100	NaN
4	0.000584	0.000228	0.000148	0.000111	0.000090

Table 15: *Studying the development of the average loss on networks without ReLU layers until the stopping criterion. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figure 16.*

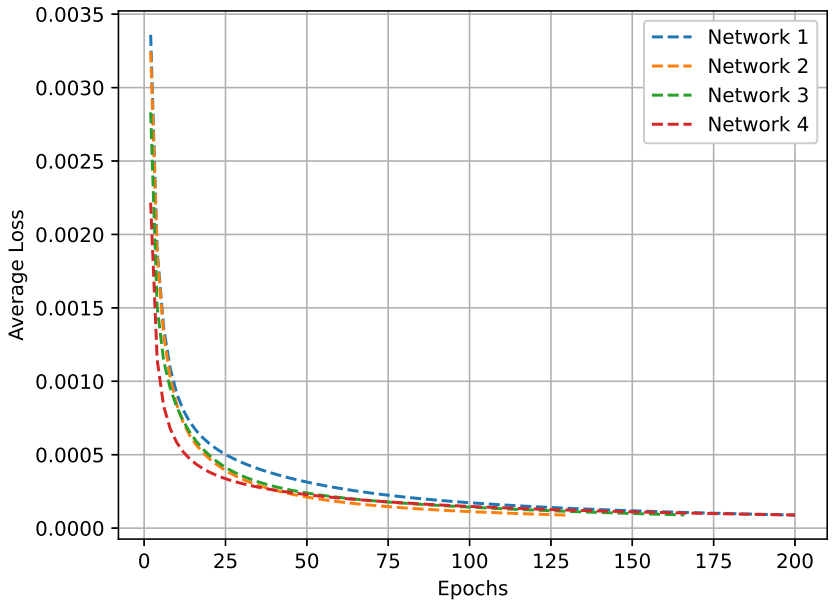

Figure 16: *Studying the development of the average loss on networks without ReLU layers until the stopping criterion. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 15.*

5.3 Compare networks with and without ReLU layers

Now we want to compare networks with and without ReLU layers. We want to see whether the ReLU layers help and speed up the learning. This turned out to be more complicated than we hoped. When teaching the network with ReLU layers with the learning rate of $1 \cdot 10^{-2}$ networks did not produce any values. We had to go down to the value of $8 \cdot 10^{-4}$ before calculations started to give any numerical values, but it was still unstable. Value $1 \cdot 10^{-4}$ is the most reliable and we continue with it with all of the networks with and without ReLU.

When adding the ReLU layers to a network it doubles the amount of layers. So always when we compare networks with and without ReLU we double the number of the layers in a network without ReLU. Now we have 5 layers but for networks without ReLU we have 10.

We generate four networks with ReLU layers and four networks without ReLU layers and teach them until the stopping criterion. We switched the stopping criterion from $9 \cdot 10^{-5}$ to $9 \cdot 10^{-4}$ since with the new learning rate networks learn much slower. The hypothesis is that ReLU does help the networks to learn faster.

Results are in the table 16 and in the figures 17 and 18. The networks without ReLU took 1034-1155 epochs and 69-77 minutes to reach the stopping criterion and the networks with ReLU took mainly around 526-634 epochs and 30-40 minutes. One network with ReLU took a little longer, 1070 epochs, to learn until the stopping criterion. Based on the results we can conclude that ReLU does help the networks to learn faster.

Network	50 epochs	150 epochs	1034 epochs	1068 epochs	1150 epochs
1 no ReLU	0.010811	0.004400	0.000922	0.000900	NaN
2 no ReLU	0.010590	0.003996	0.000963	0.000944	0.0009
3 no ReLU	0.008306	0.003540	0.000907	NaN	NaN
4 no ReLU	0.009749	0.004008	0.000900	NaN	NaN
1 with ReLU	0.004149	0.002435	NaN	NaN	NaN
2 with ReLU	0.005814	0.002156	NaN	NaN	NaN
3 with ReLU	0.009970	0.002576	0.000921	0.000900	NaN
4 with ReLU	0.003698	0.001606	NaN	NaN	NaN

Table 16: *A comparison of the development of the average loss on neural networks with and without ReLU layers until the stopping criterion. The learning rate is* $1 \cdot 10^{-4}$. *Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. See figures 17 and 18.*

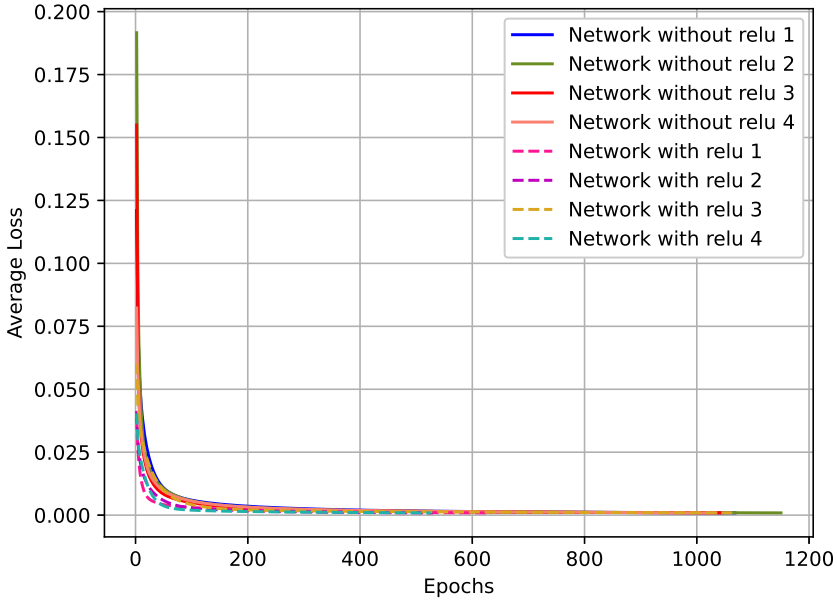

Figure 17: *A comparison of the development of the average loss on neural networks with and without ReLU layers until the stopping criterion. The learning rate is* $1 \cdot 10^{-4}$*. Each of the networks is new and structurally the same but the initial states are randomly picked and can vary. A plot from table 16.*

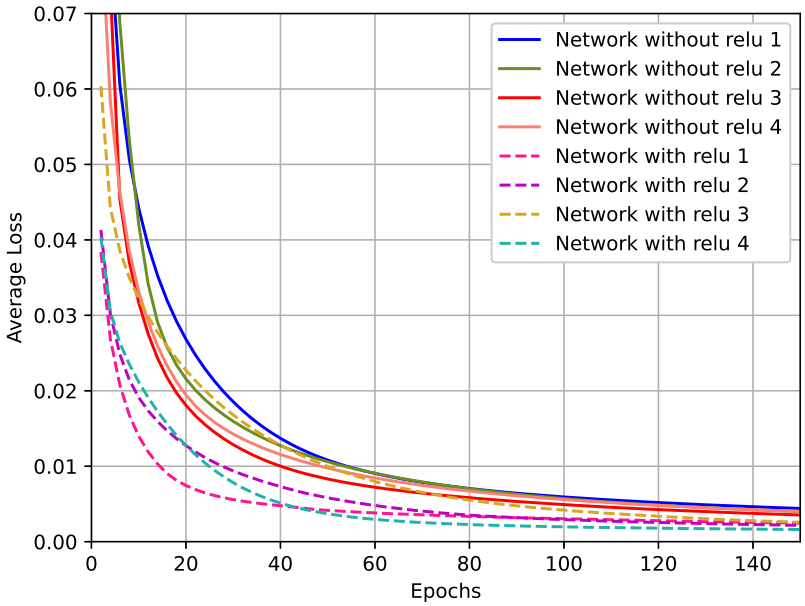

Figure 18: *Zoomed version of the figure 17. A comparison of the development of the average loss on neural networks with and without ReLU layers. A plot from table 16.*

6 Bibliography

References

[1] Emmanuele DiBenedetto. *Partial Differential Equations*. en. Birkhäuser Boston, 2010. ISBN: 978-0-8176-4551-9 978-0-8176-4552-6. DOI: 10.1007/978-0-8176-4552-6. URL: http://link.springer.com/10.1007/978-0-8176-4552-6.

[2] Lawrence C. Evans. *Partial differential equations*. eng. Second edition. Graduate studies in mathematics ; volume 19. Providence, R.I: American Mathematical Society, 2010. ISBN: 978-0-8218-4974-3.

[3] Līva Freimane. *liva-freimane/oprecnn*. URL: https://github.com/liva-freimane/oprecnn.

[4] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". en. In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15* (Jan. 2010), p. 9.

[5] Catherine F. Higham and Desmond J. Higham. "Deep Learning: An Introduction for Applied Mathematicians". en. In: *SIAM Review* 61.3 (Jan. 2019), pp. 860–891. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/18M1165748. URL: https://epubs.siam.org/doi/10.1137/18M1165748.

[6] Maarten V. de Hoop, Matti Lassas, and Christopher A. Wong. "Deep learning architectures for nonlinear operator functions and nonlinear inverse problems". en. In: *arXiv:1912.11090 [cs, math]* (Jan. 2022). arXiv: 1912.11090. URL: http://arxiv.org/abs/1912.11090.

[7] James E. Humphreys. *Linear algebraic groups*. eng. Graduate texts in mathematics 21. Springer, 1975. ISBN: 978-0-387-90108-4.

[8] Aili Joutsela. *ailijii/oprecnn*. 2022. URL: https://github.com/ailijii/oprecnn.

[9] Alexander Kachalov, Yaroslav Kurylev, and Matti Lassas. *Inverse boundary spectral problems*. eng. Monographs and surveys in pure and applied mathematics ; 123. Publication Title: Inverse boundary spectral problems. Boca Raton (FL): Chapman & Hall/CRC, 2001. ISBN: 1-58488-005-8.

[10] Jussi Korpela, Matti Lassas, and Lauri Oksanen. "Regularization strategy for inverse problem for 1+1 dimensional wave equation". en. In: *Inverse Problems* 32.6 (June 2016). arXiv:1509.04478 [math], p. 065001. ISSN: 0266-5611, 1361-6420. DOI: 10.1088/0266-5611/32/6/065001. URL: http://arxiv.org/abs/1509.04478.

[11] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". en. In: *Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28* (2013), p. 6.

[12] Yehuda Pinchover. *An Introduction to partial differential equations*. eng. University Press, 2005. ISBN: 978-0-521-61323-1.

[13] Walter Rudin. *Real and complex analysis*. eng. 3rd ed. McGraw-Hill, 1987. ISBN: 978-0-07-054234-1.

[14] Dmitry Yarotsky. *Error bounds for approximations with deep ReLU networks*. en. Number: arXiv:1610.01145 arXiv:1610.01145 [cs]. May 2017. URL: http://arxiv.org/abs/1610.01145.