Solution of the Poisson Equation with Neural Networks

Juliette Wegner

03.06.2023

Contents

1 Introduction **1**

2 Basics **3**
2.1 Poisson Equation 3
2.1.1 Solving the Poisson Equation numerically 3
2.2 Neural Networks 4
2.2.1 Structure 4
2.2.2 Training Neural Networks 5
2.2.3 Initializing Neural Networks 7
2.2.4 Hyperparameters 7
2.3 Transfer Learning 9

3 Results **10**
3.1 Laplace Equation 10
3.1.1 1D 10
3.1.2 2D 10
3.1.3 Comparing 1D and 2D 11
3.2 Poisson Equation 13
3.2.1 Testing stability with regard to noise 13
3.2.2 Transfer Learning 17
3.2.3 Transfer Learning and Noise 21

4 Summary and outlook **27**

5 Bibliography **29**

A Appendix **30**
A.1 Average Training for certain f 30
A.2 Using different values of λ in the loss function 30
A.3 An interesting outlier 33

1 Introduction

Examining the dynamics of a plasma, one needs to calculate the forces on each particle at each time step. Using particle-particle interaction, the number of calculations scale quadratic with the number of particles since each particle produces a force on every other particle due to long range interactions (e.g. Coulomb forces). For this reason one uses particle-in-cell (PIC) methods, using super particles (a collection of real particles) which experience the same dynamics as individual particles, because this is only determined by the mass to charge ratio, which is identical for the super particles and the individual particles, see [8]. For many plasmas, e.g. RF or dusty plasmas, the dominant force is the Coulomb force and electrostatic approximation can be used. Instead of calculating the sum of all Coulomb forces acting on every (super) particle, one can introduce the Maxwell equations in electrostatic approximation to determine the Lorentz force

$$F = qE$$

where q is the electrical charge and E is the electrical field. The solution of the electric field is done by introducing a mesh, where the charges of the particles are mapped to the mesh nodes, which allows to solve the Poisson equation

$$\nabla^2\Phi = -\frac{\rho}{\epsilon_0}$$

where Φ is the potential, ρ is the charge density and ϵ_0 is the dielectric constant of vacuum. The electrical field is then given via

$$E = -\nabla\Phi.$$

The electric field on the grid has then to be mapped back to the position of the super particles, which are then pushed by these forces. Then, the new charge distribution determines new potentials and electric fields, and so on. The big advantage of the PIC approach is the replacement of the particle-particle forces, which scale quadratic with the number of particles, by the solution of the macro potential field in a continuum representation, which is computationally much faster.

Therefore, Poisson solvers are very important parts of PIC codes. One of those is the finite difference method which will yield a linear system of equations of the form $A_h u = b$ where A_h is a sparse matrix dependent on the grid resolution, u are the values of the solution at the grid points and b is the right hand side at the grid points. In the case of PIC this would be the (scaled) charge density mapped to the grid. Now the matrix A_h has to be

inverted once and then the back solve can be used in every time step for an altered right hand side.

Neural networks have been used in a wide range of applications, one of which is solving differential equations [5]. In this thesis we will investigate if neural networks are an alternative for solving the Poisson equation, specifically for PIC.

After introducing the basics of neural networks, we will mainly look at the one-dimensional Poisson equation. The first part concentrates on the Laplace equation. To be able to get an estimate for the scaling of neutral network solvers for higher dimensional cases, we compare a neural network approach for solving the Laplace equation in one and two dimension. PIC requires the solution of the Poisson equation, therefore we analyse how noise affects the accuracy and run time of the neural network when solving the Poisson equation. In the PIC application, only the charge densities change from one time step to the other and this in a rather smooth way. This allows to make use of transfer learning to reduce the solver time. We will investigate how transfer learning can improve the run time, meaning that we use the neural network from the previous step as a starting point for the current one. In the end, we will combine both approaches and investigate the stability of transfer learning when the right hand side has noise. Finally, the work is summarised and an outlook is given.

2 Basics

In this chapter we introduce shortly the basics for this thesis.

2.1 Poisson Equation

The Poisson equation with Dirichlet boundary conditions is given by

$$\begin{aligned} \Delta u &= f \quad \text{on } \Omega \\ u &= g \quad \text{on } \partial\Omega \end{aligned}$$

where $\Omega \subset \mathbb{R}^n$, $\partial\Omega$ is its boundary, $f \in C(\Omega)$, $g \in C(\partial\Omega)$ and $u \in C^2(\Omega) \cap C(\partial\Omega)$. We can decompose the solution u into two parts, one that satisfies the conditions on Ω and one that satisfies the boundary conditions. This will allow us to split the process of solving for u into two different neural networks like described in [7].

So let u_I be a solution of the Poisson equation with homogenous Dirichlet boundary conditions:

$$\begin{aligned} \Delta u_I &= f \quad \text{on } \Omega \\ u_I &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

and u_B a solution of the Laplace equation with Dirichlet boundary conditions:

$$\begin{aligned} \Delta u_B &= 0 \quad \text{on } \Omega \\ u_B &= g \quad \text{on } \partial\Omega. \end{aligned}$$

Then $u_I + u_B$ satisfies the original problem since

$$\Delta(u_I + u_B) = \Delta u_I + \Delta u_B = f + 0 = f \quad \text{on } \Omega$$

and

$$u_I + u_B = 0 + g = g \quad \text{on } \partial\Omega.$$

2.1.1 Solving the Poisson Equation numerically

One can solve the Poisson equation numerically by solving the system of linear equations given by $A_h u = f$ where A_h is a sparse matrix, u are the values of the solution on the grid and f is the right hand side on the grid. The index h is the step size and as such indicates the grid. We do not want to look too deeply into how the matrix is derived and how the solution is calculated, just reiterate some results to later compare our neural network to.

One important result given in [1] is about convergence: If the right hand side is uniformly continuous, then the solution derived by the finite difference method converges to the correct solution with decreasing step size. The Heine-Cantor theorem yields that on a compact set every continuous function is uniformly continuous, and in the use case of PIC we will always look at compact sets, so as long as the right hand side is continuous, the finite difference method converges tot the correct solution.

Using Gaussian elimination as described in [2] one can calculate matrices P, L and R where P is a permutation matrix, L is a lower triangular matrix and R is an upper triangular matrix such that $LR = PA_h$. This allows to solve $A_h u = f$ in linear time once P, L and R have been calculated. Since A_h stays the same over the whole process of PIC one only needs to calculate these three matrices once, so the time it takes to calculate them can be neglected. In practice the time it takes to calculate that back solve is in the magnitude of ms.

2.2 Neural Networks

2.2.1 Structure

Neural Networks consist of several layers: one input layer, a number of hidden layers and an output layer. We will number them from 0 for the input layer and k for the output layer. Every layer l has a number of neurons n_l.

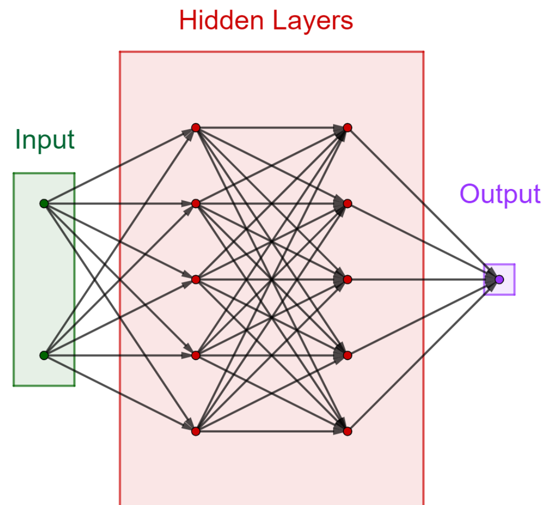


Figure 1: A neural network with $k = 3$ and $n_0 = 2, n_1 = n_2 = 5, n_3 = 1$.

Each layer is a vector $v_l \in \mathbb{R}^{n_l}$ of neurons and we propagate from one layer to the next via an affine-linear transformation and a non-linear activation function σ. We get

$$v_{l+1} = \sigma(W_{l+1}v_l + b_{l+1}) \quad \text{for } l = 0, \ldots, k-2$$

where $W_l \in \mathcal{R}^{n_{l+1} \times n_l}$ are called weight matrices and their entries weights, $b_l \in \mathbb{R}^{n_l}$ are called biases and $\sigma(v) = (\sigma(v_1), \ldots, \sigma(v_n))^T$ for $v \in \mathbb{R}^n$ with $\sigma : \mathbb{R} \to \mathbb{R}$. For the output layer we use

$$v_k = W_k v_{k-1} + b_k.$$

Common choices for the activation function are for example the rectified linear unit (ReLU)

$$\sigma(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{otherwise} \end{cases}$$

for classification problems or the logistic function

$$\sigma(x) = \frac{1}{1 + e^x}$$

as well as the hyperbolic tangent [6]. When choosing the activation function it is advantageous to consider properties that the neural network should have. If the neural network for example is supposed to approximate a differentiable function, it is advantageous to use a differentiable activation function.

2.2.2 Training Neural Networks

To train a neural network one uses known input output pairs and optimizes the weights and biases. For this we need a way to quantify how close the approximation of our neural network is to the exact output as well as an optimization procedure.

To quantify how good the approximation is, we use a loss function. Commonly used is the mean squared error, which we will also use. Let u_{NN} be the solution given by the neural network while u is the exact solution. Let also Δ_Ω denote the grid on Ω and $\Delta_{\partial\Omega}$ the grid on $\partial\Omega$. Then our loss function will be described by

$$L(u_{NN}) = \frac{1}{|\Delta_\Omega \cup \Delta_{\partial\Omega}|} \sum_{x \in \Delta_\Omega \cup \Delta_{\partial\Omega}} (u_{NN}(x) - u(x))^2$$

This loss function requires knowledge of the actual solution which we do not have. So we have to find a different way to define the loss function using

only properties that we do know. These are the equations governing the solution, i.e. the Poisson equation, which includes the right hand side f as well as the boundary condition g. But there are different ways to encode our knowledge of these two functions.

The simplest way of working with the right hand side and the boundary conditions is to simply have one set of the two for a single neural network, which is the version used in [6]. It means that every neural network can only give us solutions for this specific pair of right hand side and boundary conditions and for every change in those we will have to train a new neural network which can be quite time consuming. But it will also help keep the structure of the neural network simple, since it will only have to approximate this one specific function. Because of this, we will use this version of neural networks.

For PIC we want to solve the Poisson equation for many different right hand sides. So what would happen if we gave the neural network extra inputs, i.e. the right hand side? We could once again split the problem into two: One solving the Laplace equation and one the Poisson equation with homogenous boundary conditions. We actually successfully implemented such a network for solving the Laplace equation in one dimension, although that is no surprise since the solution has an analytical form that is quite easy, see 3.1.1.

The more complicated matter is the network for solving the Poisson equation. For one, we will have to decide how the right hand side is given. Since we cannot change the number of inputs of our network, the right hand side will have to have a fixed resolution. But in PIC this is something we generally want, so it would not be the biggest problem. That would be how to train the network.

To train the network, we will have to either generate a lot of right hand sides to train the network on or use real right hand sides from data[1]. The sort of right hand sides we train with, will give the network a bias. Meaning if the right hand sides all have a certain form, then the network will be able to solve for those but might have problems with right hand sides that look very different. Once again, because we are in the use case of PIC that is actually not that big of a problem since we expect the right hand sides to have a certain form if we study a bounded plasma between two walls. But even here, we will give our network a bias.

Another problem is the complexity of the network. The network will have to be incredibly complex to be able to solve the Poisson equation even

[1] Using real data has its own set of problems, such as noise which we will later in 3.2.1 find to be a problem for even the simpler version of our neural network.

for a single class of right hand sides. This is because we are essentially training the network to behave as a numerical solver would, but without being able to incorporate a lot of the theoretical results that the numerical solvers are dependent on. So the standard version of the neural network that we proposed to use in 2.2.4 - with three hidden layers with ten neurons each - is way too simple for this task and we would have to find a structure complex enough for this problem.

Because of all of these difficulties we never actually managed to implement a working network that solves the Poisson equation for arbitrary right hand sides f, even in one dimension. But looking at this complex problem still helped develop how we work with the simpler version of the network.

For both of these versions of networks, we can define a loss function that only uses the Poisson equation to determine how good the network approximates the correct solution. This form of neural network is called a Physics Informed Neural Network (PINN), see [5]. Whether f and g are inputs of the network or are static parameters that stay the same during the whole of the training, does not actually change the loss function which is defined the following way:

$$L(u_{NN}) = \frac{\lambda}{|\Delta_\Omega|} \sum_{x \in \Delta_\Omega} (\Delta u_{NN}(x) - f(x))^2 + \frac{1-\lambda}{|\Delta_{\partial\Omega}|} \sum_{x \in \Delta_{\partial\Omega}} (u_{NN}(x) - g(x))^2$$

with $\lambda \in (0, 1)$. We will discuss how to choose λ further in 2.2.4.

With the loss function chosen we still need to chose a method for optimizing the weights and biases. We will be using ADAM since it requires no fine tuning of its parameters to be efficient. It also performs very well for optimizing different problems including neural networks, see [4].

2.2.3 Initializing Neural Networks

Before we can start to optimize the weights and biases of our neural networks, we will need starting values. The actual initial values are important since our optimisation procedures yield a local minimum instead of a global one and converge faster in a neighbourhood of the minimum, so it is advantageous to choose good initial weights and biases. This is further discussed in [3] which also describes the Xavier method to initialize the neural network which is what we will use.

2.2.4 Hyperparameters

When building our neural network there are a couple of parameters we have to choose that will not be optimized or even changed during training. One

of these is the λ used in the loss function. A naive approach would be to weight both the boundary and the interior the same, however [6] examined different values for λ for the Laplace equation with specific functions g and found that λ needs to be smaller the higher the frequency of g. [6] proposes a value of $\lambda = 5 \cdot 10^{-4}$. However, this was determined for one special case of the Laplace equation in two dimensions. We will be mainly looking at the one-dimensional case and as such the main work will be concentrated on solving the Poisson equation with homogenous boundary conditions. So we will have to determine a good value for λ ourselves and for simplicity choose $\lambda = \frac{1}{2}$ as a starting point.

Another hyperparameter are the number of layers and neurons in each layer. [6] examined how well different neural networks can approximate the eigenfunctions of the Laplace equation in two dimensions and found four hidden layers with 50 neurons each to be a good approximation. This depends on the complexity of the solution, and as such the right hand side f, so we will choose different network structures for different tasks. Mainly, we will be using a network with three hidden layers with ten neurons each since that will be enough to approximate one-dimensional functions well.

Similar to the number of layers and neurons, the choice of the activation function will influence the results quite a bit. We want the solution to be continuously differentiable, so we will choose an activation function that reflects that. For us that is going to be the hyperbolic tangent.

Next, there is also the issue of choosing an optimizer as well as the hyper parameters arising from it. We will be using the ADAM-algorithm to optimize the parameters in our networks. This will give us three more hyper parameters: the learning rate α, and the decay parameters β_1 and β_2. [6] determined that $\alpha = 5 \cdot 10^{-4}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are some good default choices for these. We will not be looking at choosing these more optimally.

Another hyper parameter we have to consider is the number of collocation points, i.e. how many inner points of Ω and how many boundary points we will generate each iteration. Once again, [6] looked at several different values for this and found 256 of each to do an adequate job, which will be our default unless otherwise stated.

The default hyperparameters we use, have mostly been taken from [6]. They determined the optimal values by looking at the two dimensional Laplace equation for one certain g. Since we will be also looking at the one dimensional Poisson equation and will use right hand sides that might vary quite a bit from their chosen g, these hyper parameters are not necessarily optimal for what we want to do. However, a full discussion and designation of the optimal hyper parameters would go beyond the scope of this Bachelor thesis, so we will content with the default parameters from [6].

One thing this section shows, is that the choice of hyperparamters can be quite important without a lot of theoretical guidelines for what is optimal. The hyperparameters can have quite a big influence on the output of the neural network. For example if the activation function is chosen to be ReLU, then the output will not be continuously differentiable but with other choices it will be. That is of course quite a big difference. Other hyperparameters will have much more subtle influences that will be hard to pinpoint. It is also not possible to look at a single hyperparameter in a vacuum since the choice of the others will result in changes. As already stated optimising these would go past the scope of this thesis, but it is still useful to keep in mind just how much these choices matter.

2.3 Transfer Learning

If we look at the application of solving the Poisson equation for PIC methods, then in every step, there is only a slight variation in the boundary conditions, so the idea is to use Transfer Learning. We train the neural network for the boundary conditions that arise at the beginning of our calculation. This might take longer, but will only have to be done once. Afterwards, for every step we start with the network that we calculated in the previous step and train this for the new - only slightly different - set of boundary conditions. This will be faster than training a completely new neural network and is further discussed in 3.2.2.

3 Results

Where did I use this again? Since we want to use neural networks for PIC, the solution is needed on a grid. Let us call this grid $\{x_i\}_{i=0,\dots,n}$. We can simplify this further, since the distance between neighbouring grid points is always the same. So let $h = \frac{b-a}{n}$, then $x_i = a + i \cdot h$.

3.1 Laplace Equation

3.1.1 1D

The Laplace equation in one dimension is given by

$$\begin{aligned} \frac{\partial^2}{\partial x^2} u &= 0 \quad \text{ on } [a,b] \\ u(a) &= u_a \\ u(b) &= u_b \end{aligned}$$

and as such the solution is given by

$$u(x) = \frac{x-a}{b-a}(u_b - u_a) + u_a.$$

Even though we can easily give the analytical solution, it is still useful to look at solving this with neural networks to get some benchmarks and it also allows us to inspect how the computing time scales with dimension.

It is sufficient to look at the domain $[0,1]$, since we can simply get solutions for different domains through scaling. We will also use the network structure proposed in 2.2.4, which will be three hidden layers with ten neurons each. The input will be one value, i.e. the point x and the output will be one value, the solution $u(x)$.

3.1.2 2D

In two dimensions the Laplace Equation is given by

$$\begin{aligned} \frac{\partial^2}{\partial x^2} u + \frac{\partial^2}{\partial y^2} u &= 0 \quad \text{ on } \Omega \\ u &= g \quad \text{ on } \partial\Omega \end{aligned}$$

where $\Omega \subset \mathbb{R}^2$ and $\partial\Omega$ is its boundary.

For simplicity, we will focus on $\Omega = [0,1]^2$. We can also simplify the boundary conditions to

$$
\begin{aligned}
u(0,y) &= g(y)\\
u(1,y) &= 0\\
u(x,0) &= 0\\
u(x,1) &= 0.
\end{aligned}
$$

A solution to the Laplace equation on Ω with general boundary conditions can then be obtained by solving the simplified problem for every one of the edges, rotating the solutions and then simply adding them all up.

Since the two-dimensional problem is more complex than the one-dimensional one, we will use a more complex neural network for this problem. It has three hidden layers with 50 neurons each. The input are two values representing the point in cartesian coordinates (x, y) and the output is one value, the solution $u(x, y)$.

3.1.3 Comparing 1D and 2D

We want to compare how efficient our neural network is in solving the Laplace equation in one and two dimension. This will give us some basic reference point for how the run time scales with dimension. But as already discussed, the Laplace equation in one dimension has an easy analytical solution and as such is trivial to solve. We have also already stated that the main complexity will be in solving the Poisson equation, so it is important to keep in mind that this really is only a rough estimate for run time scaling.

When solving the Laplace equation on $[0, 1]$ with the boundary conditions being uniformly distributed values in $[-10, 10]$, we get a 95% confidence interval for the number of iterations of $[5389, 6256]$ and for the time in seconds we get $[23.3, 27.1]$. Which is surprisingly high considering how easy the solution should be. That is because for every solution, the network starts with arbitrary values that generally do not approximate linear functions well.

As already mentioned in 2.2.2 we also implemented a neural network that takes the boundary conditions as well as the point as inputs and returns the solution at the point. This of course makes the run time trivial compared to the half a minute the simpler version takes, since the network now only needs to compute a handful of multiplications and additions as well as the activation function a few times. But in one dimension the Laplace equation has an easy analytical solution anyway, so we cannot extrapolate that such a network could also be trained for two or more dimensions.

Getting an estimate of the training time for the network in two dimensions is of course quite dependent on the form of the boundary condition at $x = 0$. More complex boundary conditions will take longer to train. So to get a feeling for the training time for different levels of complexity, we will look at boundary conditions of the form $\sin(\omega y)$, where the frequency ω dictates the complexity. Table 1 shows the 95% confidence intervals of both the iterations and time in seconds it took to train the network for different ω. It is obvious that training takes longer for more complex boundary conditions. We can see that with every increase of the frequency the training time more than doubles.

Table 1: 95% confidence intervals for the number of iterations it took to train as well as the training time for different boundary conditions of the form $\sin(\omega y)$

ω	iterations	time [s]
2π	[7067, 7916]	[50.1, 56.2]
4π	[18408, 20790]	[130.7, 147.7]
6π	[41848, 47064]	[298.3, 335.6]

Comparing the values from table 1 with the average training time for the one dimensional Laplace equation shows that training for two dimensions takes significantly longer than it takes for training in one dimension. Even the simplest boundary condition we looked at resulted in more than double the training time. For the more complex boundary conditions the training time is more than twelve times as high. While just two reference points are not enough to get a full feeling for the scaling, we do see just how dependent the training time is on the complexity of the boundary conditions.

3.2 Poisson Equation

The Poisson equation in one dimension is given by

$$\begin{aligned}\frac{\partial^2}{\partial x^2}u &= f \quad \text{on } [a,b]\\ u(a) &= u_a\\ u(b) &= u_b\end{aligned}$$

and as such the analytical solution is obtained by simply integrating the right hand side f.

Analogous to before in 3.1.1, we can restrict the domain $[a,b]$ to $[0,1]$ without loss of generality. To see this let us look at the modified problem

$$\begin{aligned}\frac{\partial^2}{\partial \tilde{x}^2}v &= \tilde{f}(\tilde{x}) \quad \text{on } [0,1]\\ v(0) &= u_a\\ v(1) &= u_b\end{aligned}$$

where $\tilde{x} = \frac{x-a}{b-a}$ and as such $x = \tilde{x} \cdot (b-a) + b$. Since we want to solve our original problem, we want $v(\tilde{x}) = u(x)$ to be satisfied. This yields

$$\begin{aligned}f(x) =& \frac{\partial^2}{\partial x^2}u(x)\\ =& \frac{\partial^2}{\partial x^2}v(\tilde{x})\\ =& \frac{\partial^2}{\partial \tilde{x}^2}v(\tilde{x}) \cdot \left(\frac{\partial \tilde{x}}{\partial x}\right)^2\\ =& \frac{\tilde{f}(\tilde{x})}{(b-a)^2}.\end{aligned}$$

This shows that we can solve the Poisson Equation on any interval, if we can solve it on $[0,1]$.

The neural network will be the same as in 3.1.1, meaning three hidden layers with ten neurons each, one input and one output.

We can also assume the right hand side f to have an approximate shape to mimic a plasma in front of two material walls. The charge density will be approximately 0 in the middle of Ω and increase toward the boundaries.

3.2.1 Testing stability with regard to noise

Since we want to use neural networks in an application that works with data, we have to expect there to be noise. So in this section, we will investigate how

much noise there can be on the right hand side f before the neural network cannot calculate the correct solution anymore. Since the desired use case is a bounded system with a plasma in between two material walls (e.g. in RF discharges), we choose to investigate this on

$$f(x) = 1 - \cosh(x - 0.5)$$

on $[0, 1]$ with boundary conditions $u(0) = 0 = u(1)$ We approximate the charge density for this system by assuming that it is 0 in the middle of our interval and increases the closer we get to the border. This expresses the fact, that in the bulk plasma quasi-neutrality exists, otherwise electric fields will be created to produce this condition. In front of a (metallic) wall, charge separation on the scale of several Debye lengths is possible and needed to reduce the large electron fluxes (due to the small mass) and to enlarge ion fluxes to get steady-state conditions with zero currents. This results in a potential profile which drops in narrow zones close to the walls from a positive bulk value to zero. This can be approximated by a negative cosinus hyperbolicus. Since we also know, that the right hand side is given by the negative charge density, we simply get a cosinus hyperbolicus. The analytical solution to this is given by

$$u(x) = \frac{1}{2}x(x-1) - \cosh(x - \frac{1}{2}) + \cosh(\frac{1}{2}).$$

The right hand side $\tilde{f}$ with noise will be given by

$$\tilde{f}(x_i) = f(x_i) + \varepsilon_i \quad \text{with } \varepsilon_i \overset{i.i.d.}{\sim} N(0, \sigma^2)\ \forall x_i \in \Delta_\Omega.$$

We will investigate the efficiency and accuracy of the network for different values of σ. We will choose these to be $1, 5, 10, 15$ and 20 percent of the maximal amplitude of the right hand side. For comparison we will also look at the network when training for a right hand side without noise.

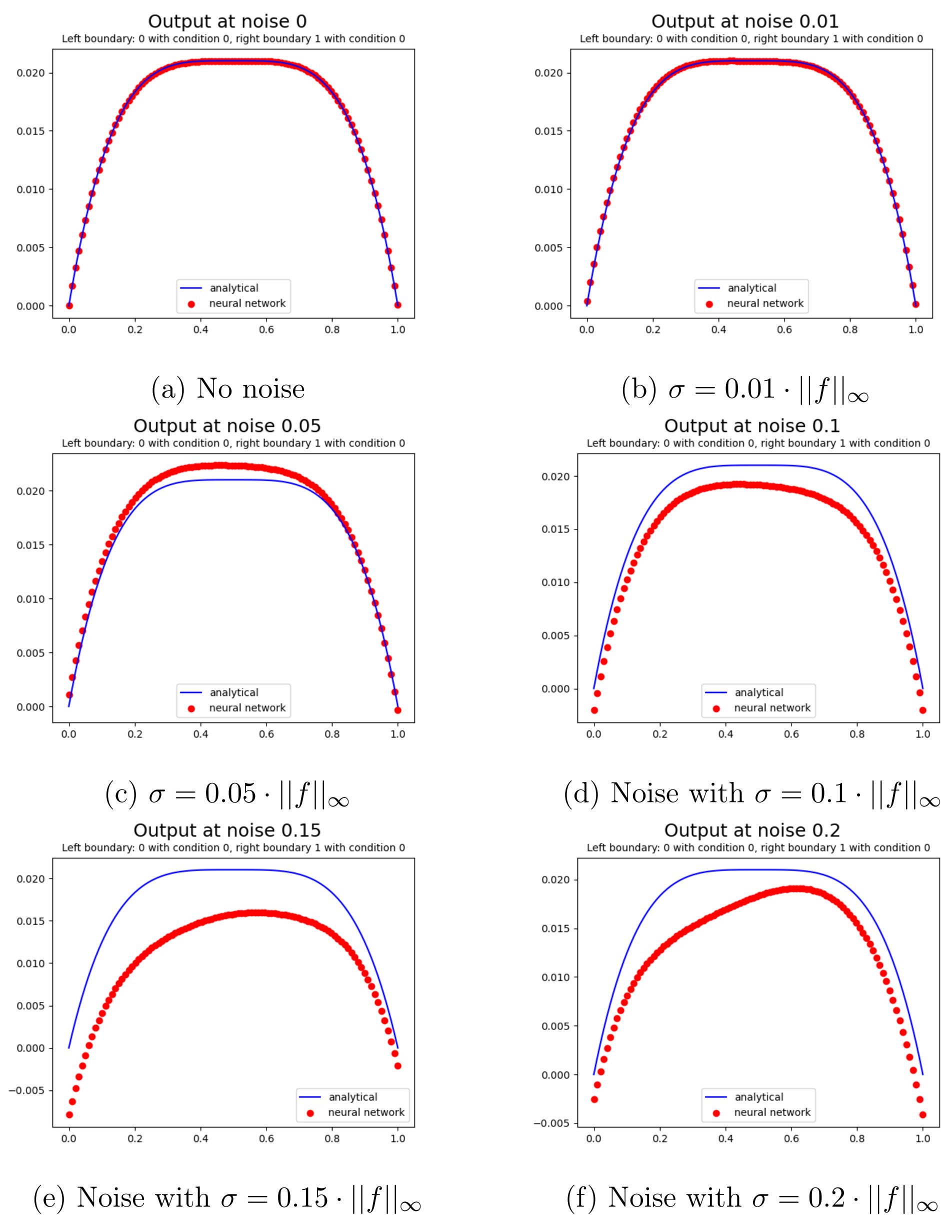
In table 2, we can see that as soon as we get any kind of noise, the training only terminates when the maximum number of iterations is reached, since the loss does not reach the required threshold. Instead the loss steadily increases for greater σ and the error in comparison to the analytical solution does as well. An interesting thing we can see in the results here is that the number of iterations, and so the time training took as well, is a lot smaller than expected (see appendix A.1) when training for a right hand side without noise. This shows that if we get lucky with the initial values of the weights and biases, training might be a lot faster than expected. However, the opposite can also hold true. This stochastic component of the neural network makes it a lot less predictable than the deterministic numerical solver.

Table 2: Results of training the neural network with noises of different magnitude. Compared are the number of iterations and the time it took to train the network, as well as the loss at the end of training and the mean squared error of the computed solution compared to the analytical one. After 10000 iterations the training automatically finishes, regardless of the loss. The resolution of the right hand side was 100.

$\frac{\sigma}{\|\|f\|\|_\infty}$	Time [s]	Iterations	Loss	MSE
0.00	85.34	992	$1.014 \cdot 10^{-5}$	$1.397 \cdot 10^{-9}$
0.01	862.34	10000	$4.367 \cdot 10^{-5}$	$2.096 \cdot 10^{-8}$
0.05	884.59	10000	$1.029 \cdot 10^{-3}$	$9.196 \cdot 10^{-7}$
0.10	876.49	10000	$3.917 \cdot 10^{-3}$	$4.923 \cdot 10^{-6}$
0.15	879.55	10000	$1.344 \cdot 10^{-2}$	$3.832 \cdot 10^{-5}$
0.20	874.28	10000	$3.839 \cdot 10^{-2}$	$1.441 \cdot 10^{-5}$

As we can see, for $\sigma = 0.15 \cdot ||f||_\infty$ we actually get the biggest error. This can be explained with figure 2(e), where we can see that while the neural network approximates the general shape of the analytical solution - hence the loss being small - it does not satisfy the boundary conditions of $u(0) = 0 = u(1)$. This discrepancy is especially big at the left boundary of the interval. The bigger σ gets, the likelier this is to happen and as we can see, even for smaller σ the boundary conditions are not fulfilled as exactly as we want them to be. Bigger σ means that the noise has a bigger influence and as such the loss on the interior of the interval increases. If the loss on the interior is big, it will not make a (big) difference if the loss on the boundary also increases and as such the network does not penalize shifting the function as much as it otherwise would. Specifically in figure 2(e), we can see that the left hand side of the function is shifted by almost 0.01, which will result in a loss of only around 10^{-4}. Which is considerably smaller than the total loss, which is greater than 10^{-2} as shown in table 2. Now, this is were one would normally start optimizing the hyperparameter λ since it has a very direct influence on the loss function that could help combat this. As already stated in 2.2.4, we will not be optimize hyperparamters in this thesis, but to further investigate the noise we did look at different λ to increase the significance of the boundary conditions and it did not change the core problem, see appendix A.2.

The results shown in table 2 and figure 2 are representative of the process and the results are reproducible over several different generations of noise. Specifically, the network always seems to have problems reaching the loss



(a) No noise

(b) $\sigma = 0.01 \cdot ||f||_\infty$

(c) $\sigma = 0.05 \cdot ||f||_\infty$

(d) Noise with $\sigma = 0.1 \cdot ||f||_\infty$

(e) Noise with $\sigma = 0.15 \cdot ||f||_\infty$

(f) Noise with $\sigma = 0.2 \cdot ||f||_\infty$

Figure 2: Training the neural network for right hand sides with different levels of noise in comparison to the analytical solution.

threshold as soon as there is any noise and the approximation of the right hand side gets continually worse with bigger σ. For 1% it always seems to be approximating quite well. 10% does seem to be a threshold after which the boundary conditions are not approximated well anymore.

This shows that in order for our network to be able to get the correct solution, the noise has to be very small as the error increases quickly with more noise as seen in table 2. In practice, this means that we need to add more particles to the PIC simulation. While more particles will not increase the run time of the solver, and in fact might even decrease it, it will increase the run time of other steps in the PIC simulation. Specifically it will increase the run time of the pusher, i.e. it will take longer to map the forces to the particles and then move these particles. In order to be efficient, the run time of both the pusher and solver should be about equal. This would be quite difficult to achieve with the neural network, especially since training with noise only seems to terminate after a maximum number of iterations is reached.

3.2.2 Transfer Learning

As discussed in 2.3, in the use case of PIC, we will have to solve the Poisson Equation for many different right hand sides f. We do however have the additional information that in every step the current right hand side is not too different from the previous one. We now want to use this to try and save time when training the neural network.

Once again we work on the interval $[0, 1]$ and use the boundary conditions $u(0) = 0 = u(1)$, as in 3.2.1. Now however, we need a family of right hand sides that change over time. We make the ansatz

$$f(x,t) = 1 - \cosh\left(\lambda(t)(x - \frac{1}{2})\right)$$

with some factor $\lambda(t)$. This yields the analytical solution

$$u(x,t) = \frac{1}{2}x(x-1) - \frac{1}{\lambda(t)^2}\left(\cosh\left(\lambda(t)(x-\frac{1}{2})\right) - \cosh(\frac{\lambda(t)}{2})\right).$$

We will now specifically look at two different $\lambda(t)$: $\lambda_1(t) = 1 + \frac{t}{4}$ and $\lambda_2(t) = 1 + t$. It is obvious that λ_2 will yield bigger changes between time steps than λ_1.

As seen in figure 3 the network is able to approximate the analytical solution quite well. Tables 3 and 4 support this, as they show the error to be consistently small. While, especially in table 4, the error does grow

Table 3: Results of Transfer Training for $\lambda_1(t)$. The Relative Loss and Relative MSE columns are the respective non-relative columns divided by $||f||_\infty$ for ease of comparison.

t	Time [s]	Iterations	loss	relative loss	MSE	relative MSE
0	89.04	1019	$1.012 \cdot 10^{-5}$	$9.911 \cdot 10^{-6}$	$2.985 \cdot 10^{-9}$	$2.923 \cdot 10^{-9}$
1	171.00	1968	$1.575 \cdot 10^{-5}$	$9.757 \cdot 10^{-6}$	$2.436 \cdot 10^{-8}$	$1.509 \cdot 10^{-8}$
2	265.88	3053	$2.320 \cdot 10^{-5}$	$9.839 \cdot 10^{-6}$	$3.805 \cdot 10^{-8}$	$1.614 \cdot 10^{-8}$
3	206.58	2380	$3.207 \cdot 10^{-5}$	$9.828 \cdot 10^{-6}$	$7.850 \cdot 10^{-9}$	$2.406 \cdot 10^{-9}$
4	148.54	1711	$4.254 \cdot 10^{-5}$	$9.791 \cdot 10^{-6}$	$4.598 \cdot 10^{-8}$	$1.058 \cdot 10^{-8}$
5	118.13	1358	$4.961 \cdot 10^{-5}$	$8.828 \cdot 10^{-6}$	$3.829 \cdot 10^{-8}$	$6.814 \cdot 10^{-9}$
6	91.16	1052	$7.048 \cdot 10^{-5}$	$9.917 \cdot 10^{-6}$	$2.474 \cdot 10^{-7}$	$3.481 \cdot 10^{-8}$
7	103.44	1205	$8.815 \cdot 10^{-5}$	$9.981 \cdot 10^{-6}$	$2.072 \cdot 10^{-7}$	$2.346 \cdot 10^{-8}$
8	198.22	2278	$1.057 \cdot 10^{-4}$	$9.767 \cdot 10^{-6}$	$2.111 \cdot 10^{-7}$	$1.951 \cdot 10^{-8}$
9	195.00	2245	$1.273 \cdot 10^{-4}$	$9.718 \cdot 10^{-6}$	$1.116 \cdot 10-7$	$8.521 \cdot 10^{-9}$
10	146.46	1686	$1.468 \cdot 10^{-4}$	$9.342 \cdot 10^{-6}$	$3.374 \cdot 10^{-8}$	$2.147 \cdot 10^{-9}$

Table 4: Results of Transfer Training for $\lambda_2(t)$. The Relative Loss and Relative MSE columns are the respective non-relative quantities divided by $||f||_\infty$ for ease of comparison.

t	Time [s]	Iterations	Loss	Relative Loss	MSE	Relative MSE
0	153.29	1763	$9.119 \cdot 10^{-6}$	$8.932 \cdot 10^{-6}$	$1.439 \cdot 10^{-8}$	$1.410 \cdot 10^{-8}$
1	315.14	3636	$4.322 \cdot 10^{-5}$	$9.947 \cdot 10^{-6}$	$2.656 \cdot 10^{-8}$	$6.114 \cdot 10^{-9}$
2	191.97	2221	$9.245 \cdot 10^{-5}$	$8.545 \cdot 10^{-6}$	$1.644 \cdot 10^{-8}$	$1.519 \cdot 10^{-9}$
3	471.39	5430	$2.051 \cdot 10^{-4}$	$9.281 \cdot 10^{-6}$	$2.319 \cdot 10^{-6}$	$1.049 \cdot 10^{-7}$
4	344.91	3964	$3.830 \cdot 10^{-4}$	$9.327 \cdot 10^{-6}$	$1.390 \cdot 10^{-5}$	$3.385 \cdot 10^{-7}$
5	230.88	2662	$7.062 \cdot 10^{-4}$	$9.735 \cdot 10^{-6}$	$5.761 \cdot 10^{-6}$	$7.941 \cdot 10^{-8}$
6	368.04	4233	$1.136 \cdot 10^{-3}$	$9.116 \cdot 10^{-6}$	$1.571 \cdot 10^{-5}$	$1.261 \cdot 10^{-7}$
7	787.49	9073	$1.965 \cdot 10^{-3}$	$9.338 \cdot 10^{-6}$	$1.488 \cdot 10^{-5}$	$7.069 \cdot 10^{-8}$
8	868.78	10000	$5.858 \cdot 10^{-3}$	$1.664 \cdot 10^{-5}$	$1.138 \cdot 10^{-5}$	$3.230 \cdot 10^{-8}$
9	720.86	8285	$5.539 \cdot 10^{-3}$	$9.457 \cdot 10^{-6}$	$5.971 \cdot 10^{-5}$	$1.019 \cdot 10^{-7}$
10	868.64	10000	$7.426 \cdot 10^{-1}$	$7.649 \cdot 10^{-4}$	$2.952 \cdot 10^{-4}$	$3.041 \cdot 10^{-7}$

continually with time, when accounting for the increased amplitude of both the right hand side f as well as the solution u, there are no significant changes at least in the case of $\lambda_1(t)$. In the case of $\lambda_2(t)$ the error does grow by one magnitude, but it is still small.

Comparing tables 3 and 4 shows that $\lambda_2(t)$ seems to require more iterations to finish training. For $\lambda_1(t)$ most time steps take less than 2000 iterations to train. In general, most steps take about the same amount of iterations as it takes to train the case for $t = 0$ (see appendix A.1). If we compare the iterations it took to train for $\lambda_1(8)$ in transfer learning, versus training for it with a new network, see A.1, there is no significant difference. Although in this case the iterations for $\lambda_1(8)$ are on the higher end of the spectrum. In general, table 3 fails to show any significant advantage of transfer learning.

In comparison, for $\lambda_2(t)$ it takes significantly more iterations to satisfy the stopping condition. Most of the time it took more than 3000 iterations and the steps for $t = 8$ and $t = 10$ only terminated because the maximum number of iterations had been reached. The number of iterations for the steps where transfer learning is used is also always several times bigger than the iterations for the first step (see appendix A.1) and even if we compare it to the time it takes to train for $\lambda_2(2)$, the second time step does not save any time compared to just training directly.

The most important question however is how the neural network compares to numerically solving the problem. And the answer is quite unfavourably. Even the fastest step took more than a minute and more than 1000 iterations to train. Compared to the milliseconds it would take for a numerical solver to find the solution this is slower by several orders of magnitude. When looking at transfer learning specifically we wanted to utilise the knowledge we had about the right hand side to improve training time, but the improvement is at best negligible and at worst non-existent. With the numerical procedure on the other hand, we can use that the same mesh is used in every step and as such only need to calculate the time intensive step once, which gives the numerical solver a further advantage over the neural network. The numerical solver has one final advantage over the neural network: It always converges to the correct solution. For the neural network that would need to be proven.

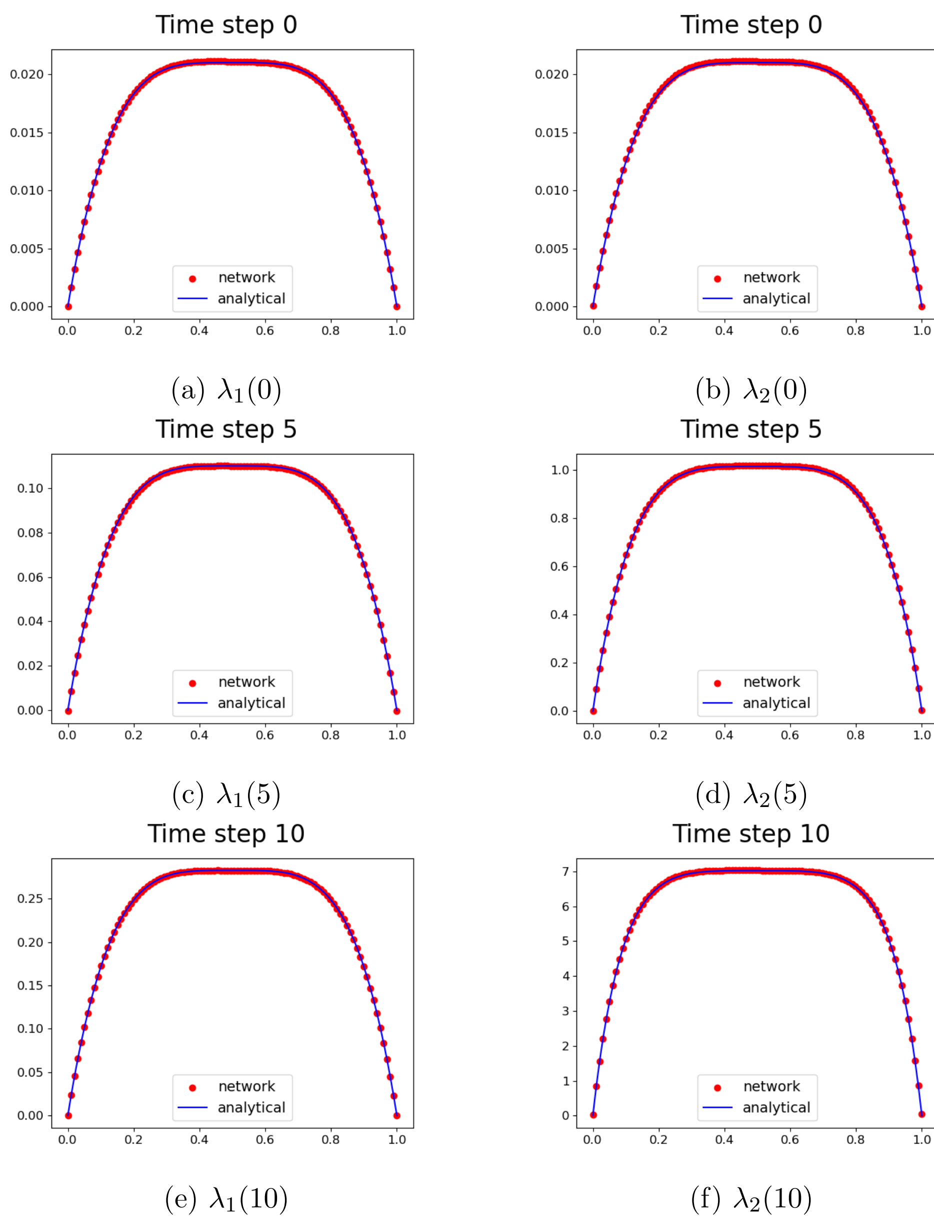


Figure 3: Using Transfer Learning for right hand sides that change at a different rate.

3.2.3 Transfer Learning and Noise

In the use case of PIC, we want to utilise transfer learning to help reduce the training time of the neural network. When working with real data however, we will also have noise. In this section we will investigate how those two things interact and if the noise destabilizes the transfer learning or if transfer learning is helpful in smoothing the noise.

We will use the same sort of right hand sides as in 3.2.2 with $\lambda_1(t)$ since that yielded better results than λ_2. So we get the right hand side

$$f(x,t) = 1 - \cosh\left((1+\frac{t}{4})(x-\frac{1}{2})\right)$$

and the analytical solution

$$u(x,t) = \frac{1}{2}x(x-1) - \frac{1}{(1+\frac{t}{4})^2}\left(\cosh\left((1+\frac{t}{4})(x-\frac{1}{2})\right) - \cosh(\frac{1+\frac{t}{4}}{2})\right).$$

For the right hand side with noise we will make the same ansatz as in 3.2.1, so

$$\tilde{f}(x_i) = f(x_i) + \varepsilon_i \quad \text{with } \varepsilon_i \overset{i.i.d.}{\sim} N(0,\sigma^2).$$

In accordance with the results in 3.2.1, we will test this for σ that are at most 10% of the maximum norm of the right hand side.

Table 5: Results of Transfer Training for $\lambda_1(t)$ without noise. The Relative Loss and Relative MSE columns are the respective non-relative columns divided by $||f||_\infty$ for ease of comparison.

t	Time [s]	Iterations	Loss	Relative Loss	MSE	Relative MSE
0	327.43	3769	$1.004 \cdot 10^{-5}$	$9.838 \cdot 10^{-6}$	$7.295 \cdot 10^{-9}$	$7.145 \cdot 10^{-9}$
1	207.17	2392	$1.570 \cdot 10^{-5}$	$9.726 \cdot 10^{-6}$	$5.177 \cdot 10^{-8}$	$3.207 \cdot 10^{-8}$
2	157.61	1791	$2.200 \cdot 10^{-5}$	$9.334 \cdot 10^{-6}$	$3.949 \cdot 10^{-8}$	$1.675 \cdot 10^{-8}$
3	173.56	1956	$3.011 \cdot 10^{-5}$	$9.227 \cdot 10^{-6}$	$7.404 \cdot 10^{-8}$	$2.269 \cdot 10^{-8}$
4	263.81	3031	$4.317 \cdot 10^{-5}$	$9.937 \cdot 10^{-6}$	$5.016 \cdot 10^{-9}$	$1.155 \cdot 10^{-9}$
5	253.17	2864	$5.422 \cdot 10^{-5}$	$9.649 \cdot 10^{-6}$	$3.216 \cdot 10^{-8}$	$5.723 \cdot 10^{-9}$
6	213.25	2489	$7.032 \cdot 10^{-5}$	$9.893 \cdot 10^{-6}$	$1.563 \cdot 10^{-8}$	$2.199 \cdot 10^{-9}$
7	172.61	2012	$8.680 \cdot 10^{-5}$	$9.828 \cdot 10^{-6}$	$1.749 \cdot 10^{-8}$	$1.980 \cdot 10^{-9}$
8	125.72	1451	$1.071 \cdot 10^{-4}$	$9.902 \cdot 10^{-6}$	$1.851 \cdot 10^{-7}$	$1.711 \cdot 10^{-8}$
9	105.94	1223	$1.218 \cdot 10^{-4}$	$9.297 \cdot 10^{-6}$	$6.586 \cdot 10^{-8}$	$5.027 \cdot 10^{-9}$
10	92.49	1042	$1.558 \cdot 10^{-4}$	$9.917 \cdot 10^{-6}$	$1.106 \cdot 10^{-7}$	$7.039 \cdot 10^{-9}$

Table 6: Results of Transfer Training for $\lambda_1(t)$ with $\sigma = 0.01||f||_\infty$. The Relative Loss and Relative MSE columns are the respective non-relative columns divided by $||f||_\infty$ for ease of comparison. Every single step only terminated after 10,000 iterations, so the columns depicting iterations and training time have been emitted.

t	loss	relative loss	MSE	relative MSE
0	$3.410 \cdot 10^{-5}$	$3.307 \cdot 10^{-5}$	$7.222 \cdot 10^{-8}$	$7.004 \cdot 10^{-8}$
1	$8.182 \cdot 10^{-5}$	$5.022 \cdot 10^{-5}$	$4.460 \cdot 10^{-6}$	$2.738 \cdot 10^{-6}$
2	$2.102 \cdot 10^{-4}$	$8.857 \cdot 10^{-5}$	$1.709 \cdot 10^{-6}$	$7.202 \cdot 10^{-7}$
3	$3.351 \cdot 10^{-4}$	$1.023 \cdot 10^{-4}$	$3.571 \cdot 10^{-7}$	$1.089 \cdot 10^{-7}$
4	$6.267 \cdot 10^{-4}$	$1.444 \cdot 10^{-4}$	$3.938 \cdot 10^{-7}$	$9.077 \cdot 10^{-8}$
5	$8.357 \cdot 10^{-4}$	$1.490 \cdot 10^{-4}$	$3.144 \cdot 10^{-6}$	$5.608 \cdot 10^{-7}$
6	$1.979 \cdot 10^{-3}$	$2.779 \cdot 10^{-4}$	$1.143 \cdot 10^{-6}$	$1.605 \cdot 10^{-7}$
7	$2.259 \cdot 10^{-3}$	$2.543 \cdot 10^{-4}$	$1.183 \cdot 10^{-6}$	$1.331 \cdot 10^{-7}$
8	$3.103 \cdot 10^{-3}$	$2.824 \cdot 10^{-4}$	$1.668 \cdot 10^{-6}$	$1.518 \cdot 10^{-7}$
9	$3.853 \cdot 10^{-3}$	$2.912 \cdot 10^{-4}$	$2.441 \cdot 10^{-5}$	$1.845 \cdot 10^{-6}$
10	$8.600 \cdot 10^{-3}$	$5.449 \cdot 10^{-4}$	$4.789 \cdot 10^{-6}$	$3.034 \cdot 10^{-7}$

Table 7: Results of Transfer Training for $\lambda_1(t)$ with $\sigma = 0.05||f||_\infty$. The Relative Loss and Relative MSE columns are the respective non-relative columns divided by $||f||_\infty$ for ease of comparison. Every single step only terminated after 10,000 iterations, so the columns depicting iterations and training time have been emitted.

t	loss	relative loss	MSE	relative MSE
0	$7.169 \cdot 10^{-4}$	$6.811 \cdot 10^{-4}$	$7.245 \cdot 10^{-7}$	$6.884 \cdot 10^{-7}$
1	$2.590 \cdot 10^{-3}$	$1.566 \cdot 10^{-3}$	$5.208 \cdot 10^{-6}$	$3.147 \cdot 10^{-6}$
2	$5.095 \cdot 10^{-3}$	$2.168 \cdot 10^{-3}$	$1.543 \cdot 10^{-6}$	$6.562 \cdot 10^{-7}$
3	$9.560 \cdot 10^{-3}$	$2.729 \cdot 10^{-3}$	$5.995 \cdot 10^{-6}$	$1.711 \cdot 10^{-6}$
4	$1.184 \cdot 10^{-2}$	$2.625 \cdot 10^{-3}$	$7.826 \cdot 10^{-6}$	$1.735 \cdot 10^{-6}$
5	$1.977 \cdot 10^{-2}$	$3.472 \cdot 10^{-3}$	$1.423 \cdot 10^{-5}$	$2.500 \cdot 10^{-6}$
6	$4.353 \cdot 10^{-2}$	$6.271 \cdot 10^{-3}$	$2.004 \cdot 10^{-5}$	$2.887 \cdot 10^{-6}$
7	$3.691 \cdot 10^{-2}$	$3.980 \cdot 10^{-3}$	$4.998 \cdot 10^{-6}$	$5.389 \cdot 10^{-7}$
8	$8.294 \cdot 10^{-2}$	$6.934 \cdot 10^{-3}$	$3.408 \cdot 10^{-4}$	$2.849 \cdot 10^{-5}$
9	$1.182 \cdot 10^{-1}$	$8.845 \cdot 10^{-3}$	$3.034 \cdot 10^{-5}$	$2.270 \cdot 10^{-6}$
10	$1.557 \cdot 10^{-1}$	$9.095 \cdot 10^{-3}$	$2.549 \cdot 10^{-4}$	$1.489 \cdot 10^{-5}$

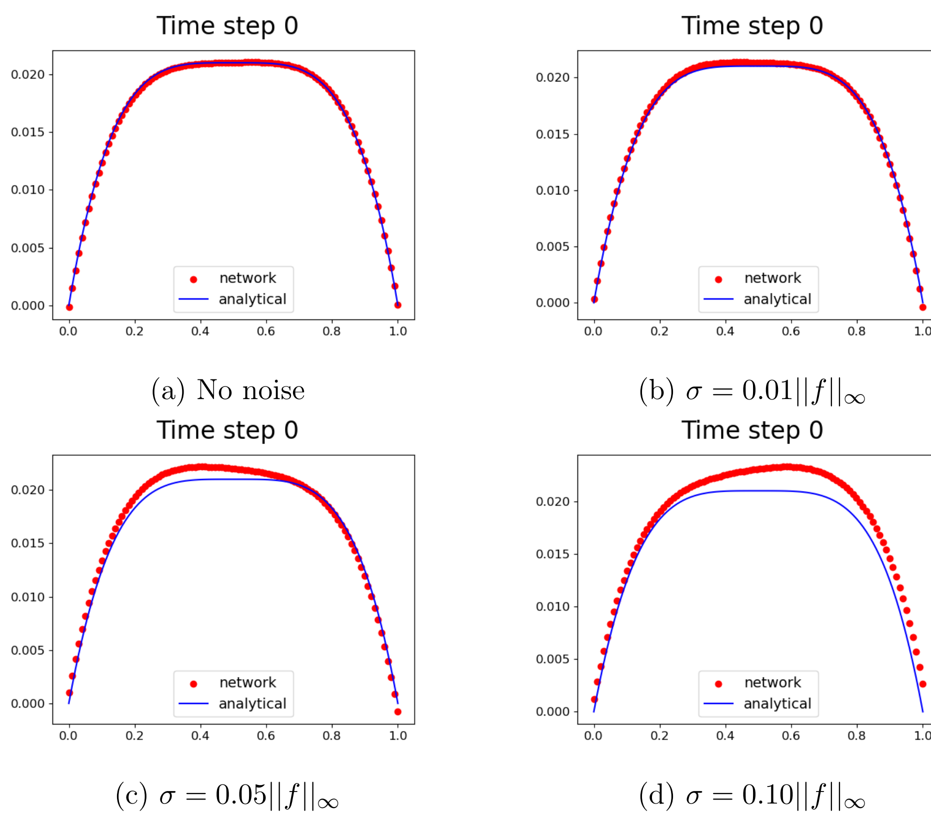
Table 8: Results of Transfer Training for $\lambda_1(t)$ with $\sigma = 0.10||f||_\infty$. The Relative Loss and Relative MSE columns are the respective non-relative columns divided by $||f||_\infty$ for ease of comparison. Every single step only terminated after 10,000 iterations, so the columns depicting iterations and training time have been emitted.

t	loss	relative loss	MSE	relative MSE
0	$2.827 \cdot 10^{-3}$	$2.545 \cdot 10^{-3}$	$3.136 \cdot 10^{-6}$	$2.823 \cdot 10^{-6}$
1	$8.461 \cdot 10^{-3}$	$4.550 \cdot 10^{-3}$	$4.711 \cdot 10^{-6}$	$2.533 \cdot 10^{-6}$
2	$1.390 \cdot 10^{-2}$	$5.391 \cdot 10^{-3}$	$8.481 \cdot 10^{-6}$	$3.290 \cdot 10^{-6}$
3	$4.111 \cdot 10^{-2}$	$1.128 \cdot 10^{-2}$	$1.041 \cdot 10^{-5}$	$2.858 \cdot 10^{-6}$
4	$8.828 \cdot 10^{-2}$	$1.750 \cdot 10^{-2}$	$3.518 \cdot 10^{-5}$	$6.975 \cdot 10^{-6}$
5	$1.122 \cdot 10^{-1}$	$1.833 \cdot 10^{-2}$	$7.377 \cdot 10^{-5}$	$1.205 \cdot 10^{-5}$
6	$1.893 \cdot 10^{-1}$	$2.705 \cdot 10^{-2}$	$1.240 \cdot 10^{-5}$	$1.771 \cdot 10^{-6}$
7	$2.906 \cdot 10^{-1}$	$2.757 \cdot 10^{-2}$	$7.960 \cdot 10^{-5}$	$7.551 \cdot 10^{-6}$
8	$4.260 \cdot 10^{-1}$	$3.606 \cdot 10^{-2}$	$3.216 \cdot 10^{-5}$	$2.722 \cdot 10^{-6}$
9	$5.703 \cdot 10^{-1}$	$3.614 \cdot 10^{-2}$	$3.010 \cdot 10^{-4}$	$1.908 \cdot 10^{-5}$
10	$7.987 \cdot 10^{-1}$	$4.604 \cdot 10^{-2}$	$7.016 \cdot 10^{-4}$	$4.044 \cdot 10^{-5}$

As soon as noise was introduced to the right hand side, the training only finished when reaching the maximum number of 10,000 iterations. This is consistent with what we found in 3.2.1. Table 5 is also consistent with the results from 2.3, both in the magnitude of errors as well as the number of iterations training took.

When there is no noise table 5 shows a small error that stays in the same range for all steps. This does not hold true when we introduce noise to the right hand side. The error for different time steps fluctuates in up to two orders of magnitude and is generally greater for later steps. Also to note is that the error generally increases with increasing noise from table 5 to 8. This was to be expected from the results in 3.2.1.

Figures 4 and 5 show what we already saw through tables 5 to 8: Without noise the analytical solution gets approximated very well and the bigger the noise the worse the approximation. We can also see what the increased error for bigger noise results in: For a noise of 1% the analytical solution is still approximated quite well, while especially for 10% we get quite big differences.



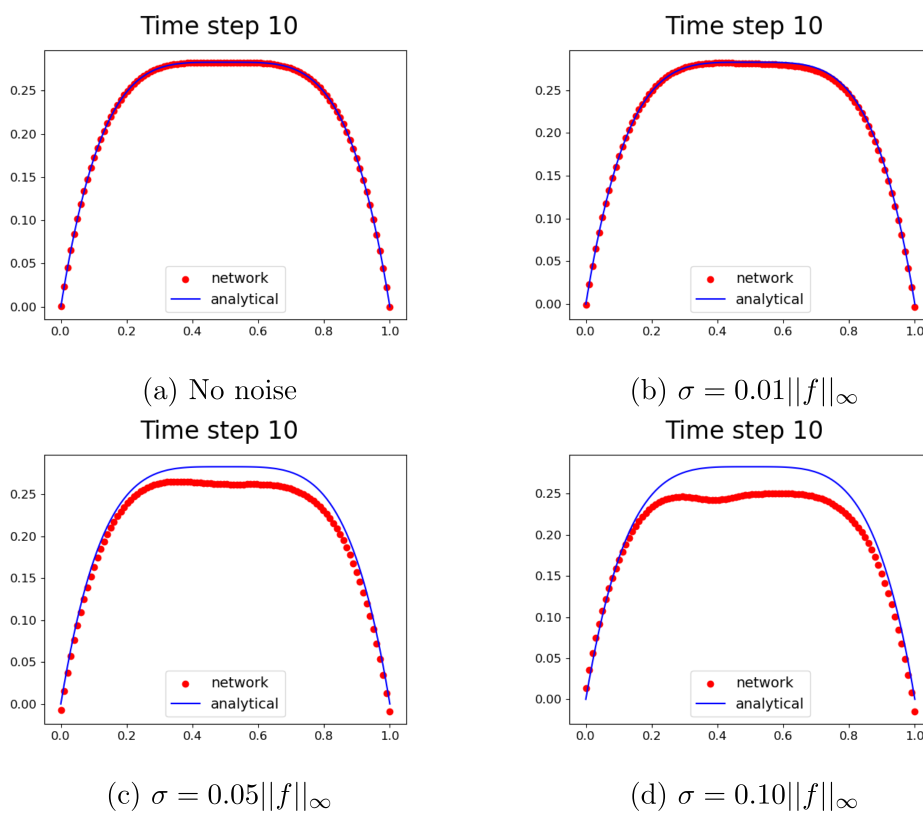
(a) No noise

(b) $\sigma = 0.01||f||_\infty$

(c) $\sigma = 0.05||f||_\infty$

(d) $\sigma = 0.10||f||_\infty$

Figure 4: The initial step of transfer learning when training with different levels of noise



(a) No noise

(b) $\sigma = 0.01||f||_\infty$

(c) $\sigma = 0.05||f||_\infty$

(d) $\sigma = 0.10||f||_\infty$

Figure 5: The tenth step of transfer learning when training with different levels of noise

This once again shows what we already saw in 3.2.1: The noise has to be very small for our neural network to be able to approximate the analytical solution well and decreasing the noise in the actual data would mean an increase in the run time. The use of transfer learning does not seem to significantly decrease the error, since the error for the different noise levels in this section is about the same as it was in 3.2.1 when there was no transfer learning used.

4 Summary and outlook

The research question of this thesis is **if neural networks could prove to be advantageous for solving the Poisson equation, specifically for PIC**. When comparing the time it takes to train the neural network versus the run time of a numerical solver the neural network is worse by several orders of magnitude. The neural network needs some minutes to train while the time it takes for numerical solvers is several 10s of milliseconds. So, **the approach fails to yield an advantage over numerical solvers**.

When investigating the influence of noise on the accuracy of the neural network as well as on the needed training time, increased levels of noise lead to increased errors and increased training time. Specifically, the training did not reach the required threshold for the loss to terminate as soon as noise is introduced. For noise that has a standard deviation of 1% of the amplitude of the right hand side the error is still quite small and the visual approximation is as good as for no noise. 10% does seem to be a threshold after which the boundary conditions are not approximated well anymore.

This shows that in order for our network to be able to get the correct solution, the noise has to be very small as the error increases quickly with more noise as seen in table 2. In practice, this means that we need to add more particles to the PIC simulation. While more particles will not increase the run time of the solver, and in fact might even decrease it, it will increase the run time of other steps in the PIC simulation. Specifically it will increase the run time of the pusher, i.e. it will take longer to map the forces to the particles and then move these particles. In order to be efficient, the run time of both the pusher and solver should be about equal. This would be quite difficult to achieve with the neural network, especially since training with noise only seems to terminate after a maximum number of iterations is reached.

Next the question was addressed if prior knowledge of the right hand side can speed up training through transfer learning. Not surprisingly, larger differences in right hand sides will result in longer run times to train the network. Independent of the amount of change, there was no significant reduction of run time using transfer learning compared to direct training.

Lastly, the combination of a right hand side with noise with transfer learning was studied. As before, the loss threshold could not be reached after noise was introduced to the right hand side. The error also steadily increased for increasing noise and stayed in the same ballpark as it had been when we did not use transfer learning. So transfer learning does not provide a way to smooth the noise, either.

Even though the neural networks we constructed did not perform favourably

when compared to numerical solvers, neural networks are still an interesting and useful tool when solving differential equations since they are relatively simple to implement compared to other numerical methods like finite volumes. Further optimisation should be possible with pre-trained neural networks for specific classes of right hand sides. Also, the development of faster hardware gives positive perspectives for such approaches.

5 Bibliography

[1] Dietrich Braess. *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer-Verlag, 2013.

[2] Roland W Freund and Ronald W Hoppe. *Stoer/Bulirsch: Numerische Mathematik 1*. Springer-Verlag, 2007.

[3] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[4] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[5] Stefano Markidis. "Physics-informed deep-learning for scientific computing". In: *arXiv preprint arXiv:2103.09655* (2021).

[6] Remco van der Meer. "Solving Partial Differential Equations with Neural Networks". MA thesis. Delft University of Technology, June 2019.

[7] Ali Girayhan Özbay et al. "Poisson CNN: Convolutional neural networks for the solution of the Poisson equation on a Cartesian mesh". In: *Data-Centric Engineering* 2.e6 (2021).

[8] David Tskhakaya et al. "The Particle-In-Cell Method". In: *Contributions to Plasma Physics* 47.8-9 (2007), pp. 563–594.

A Appendix

A.1 Average Training for certain f

In order to better categorise our results, we looked at the average time and number of iterations it took the neural network to train for a certain right hand side f.

We start with

$$f(x) = 1 - \cosh(x - 0.5)$$

since that is our starting point for approximating the real charge density. Training 100 times with a loss tolerance of 10^{-5} gives us a 95% confidence interval of $[1204, 1581]$ for the number of iterations the training took and one of $[103.2, 135.6]$ for the time trained in seconds.

Next we will look at

$$f(x) = 1 - \cosh(3(x - 0.5)).$$

This coincides with $\lambda_1(8)$ and $\lambda_2(2)$ from 3.2.2. For this f training 100 times with a loss tolerance of 10^{-5} gives a 95% confidence interval of $[1824, 2381]$ for the number of iterations and one of $[159.3, 207.9]$ for the time in seconds the training took. Comparing to the first f, both quantities have increased by about 50%. This might be explained by looking at both functions: The second one has a much higher curvature and as such will be harder to approximate.

A.2 Using different values of λ in the loss function

As seen in 3.2.1 the more noise there is, the likelier the network is to not be able to fulfill the boundary conditions accurately enough. If we decrease λ, we will increase the influence of the boundary condition on our loss, see 2.2.2. Since a full investigation of the influence of λ would go beyond the scope of this thesis, we only looked at one different value of λ.

We chose $\lambda = 0.05$ meaning that the significance of the loss on the interior will be dampened by about a factor 10^{-2} compared to the loss on the boundary. This was because of what we found in 3.2.1, where the loss on the boundary was smaller than the one on the interior by about factor 10^{-2}.

In table 9 it is easy to see that this new λ does not decrease training time or the error, although it is interesting to note that the training with a noise of 1% does reach the threshold for the loss. This was to be expected, since when λ was bigger, we had already seen in table 2 that the loss for this level of noise was close to reaching the threshold and it is to be expected that the reason

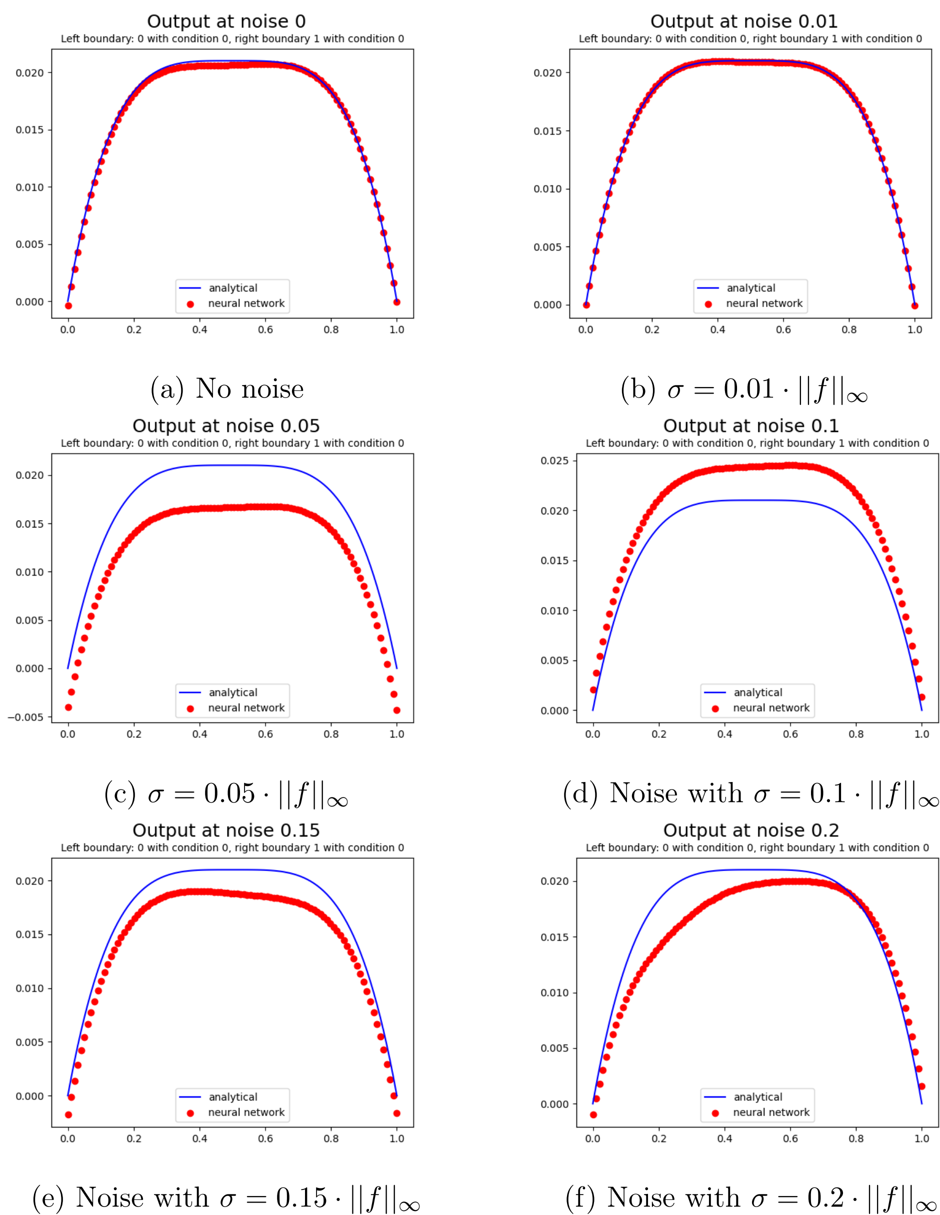
Table 9: Results of training the neural network with $\lambda = 0.05$.

$\frac{\sigma}{\|\|f\|\|_\infty}$	Time [s]	Iterations	Loss	Relative Loss	MSE	Relative MSE
0.00	59.85	687	$9.683 \cdot 10^{-6}$	$9.484 \cdot 10^{-6}$	$6.683 \cdot 10^{-8}$	$6.545 \cdot 10^{-8}$
0.01	264.80	3069	$1.011 \cdot 10^{-5}$	$9.884 \cdot 10^{-6}$	$5.688 \cdot 10^{-9}$	$5.564 \cdot 10^{-9}$
0.05	868.56	10000	$1.075 \cdot 10^{-4}$	$1.002 \cdot 10^{-4}$	$1.778 \cdot 10^{-5}$	$1.657 \cdot 10^{-5}$
0.10	866.45	10000	$3.466 \cdot 10^{-4}$	$3.199 \cdot 10^{-4}$	$9.326 \cdot 10^{-6}$	$8.608 \cdot 10^{-6}$
0.15	866.76	10000	$1.128 \cdot 10^{-3}$	$8.445 \cdot 10^{-4}$	$4.366 \cdot 10^{-6}$	$3.269 \cdot 10^{-6}$
0.20	868.25	10000	$3.692 \cdot 10^{-3}$	$2.497 \cdot 10^{-3}$	$5.005 \cdot 10^{-6}$	$3.385 \cdot 10^{-6}$

it did not reach the threshold is the loss on the interior of the interval rather than the one on the boundary. Since decreasing λ, decreases the significance of the loss on the interior, the training now terminates although the error is not significantly smaller.

Figure 6 also shows that the different λ did not significantly improve the situation. While the general shape of the analytical solution is approximated by every network, we do notice big differences, especially in figure 6(f). We can also see that there is still errors on the boundary for sufficiently big noise, for example in figure 6(c). We did manage to decrease the shifting by about half compared to figure 2, but overall the improvements are disappointing.

We can see that the error did decrease for the biggest two noise levels, but for smaller noise levels, especially for 5% it increased significantly. So we would want to choose λ dependent on the noise in the data, but to accurately calculate the level of noise is quite difficult.



(a) No noise

(b) $\sigma = 0.01 \cdot ||f||_\infty$

(c) $\sigma = 0.05 \cdot ||f||_\infty$

(d) Noise with $\sigma = 0.1 \cdot ||f||_\infty$

(e) Noise with $\sigma = 0.15 \cdot ||f||_\infty$

(f) Noise with $\sigma = 0.2 \cdot ||f||_\infty$

Figure 6: Training the neural network for right hand sides with different levels of noise in comparison to the analytical solution when $\lambda = 0.05$.

A.3 An interesting outlier

When training the neural network, we will always get different results depending on the random initialization of the weights and biases. This can both lead to better or worse results. A nice example of that is seen in table 10 which was obtained for Transfer Learning with $\lambda_1(t)$ and no noise. For the first few steps, the number of iterations it takes to train the network is about half of the iterations for training the first time (compare A.1), while for the last few steps it is significantly bigger. This shows just how dependent on random variables this method is.

Table 10: Results of Transfer Training for $\lambda_1(t)$. The Relative Loss and Relative MSE columns are the respective non-relative columns divided by $||f||_\infty$ for ease of comparison.

t	Time [s]	Iterations	loss	relative loss	MSE	relative MSE
0	90.70	1054	$9.326 \cdot 10^{-6}$	$9.134 \cdot 10^{-6}$	$3.862 \cdot 10^{-9}$	$3.783 \cdot 10^{-9}$
1	53.05	619	$1.597 \cdot 10{-}5$	$9.892 \cdot 10^{-6}$	$2.092 \cdot 10^{-8}$	$1.296 \cdot 10^{-8}$
2	126.46	1484	$2.340 \cdot 10^{-5}$	$9.924 \cdot 10^{-6}$	$2.451 \cdot 10^{-8}$	$1.040 \cdot 10^{-8}$
3	93.07	1086	$3.250 \cdot 10^{-5}$	$9.960 \cdot 10^{-6}$	$3.028 \cdot 10^{-8}$	$9.279 \cdot 10^{-9}$
4	77.23	905	$3.942 \cdot 10^{-5}$	$9.073 \cdot 10^{-6}$	$1.504 \cdot 10^{-8}$	$3.461 \cdot 10^{-9}$
5	59.53	692	$5.228 \cdot 10^{-5}$	$9.304 \cdot 10^{-6}$	$1.458 \cdot 10^{-7}$	$2.595 \cdot 10^{-8}$
6	56.39	663	$6.757 \cdot 10^{-5}$	$9.507 \cdot 10^{-6}$	$1.539 \cdot 10^{-7}$	$2.165 \cdot 10^{-8}$
7	177.06	2067	$8.604 \cdot 10^{-5}$	$9.742 \cdot 10^{-6}$	$2.465 \cdot 10^{-7}$	$2.791 \cdot 10^{-8}$
8	280.09	3271	$1.075 \cdot 10^{-4}$	$9.933 \cdot 10^{-6}$	$1.414 \cdot 10^{-7}$	$1.307 \cdot 10^{-8}$
9	231.73	2717	$1.296 \cdot 10^{-4}$	$9.895 \cdot 10^{-6}$	$6.443 \cdot 10^{-7}$	$4.918 \cdot 10^{-8}$
10	206.86	2422	$1.557 \cdot 10^{-4}$	$9.910 \cdot 10^{-6}$	$8.171 \cdot 10^{-8}$	$5.200 \cdot 10^{-9}$