
Department of Physics

FYS-STK4155 - Applied Data Analysis and Machine Learning

Project 3

# Estimating the diffusion coefficient of an MRI tracer using Physics Informed Neural Networks

*Authors:*
Jouval Somer, Stig Patey, Celine Marie Solberg & Ole Petter Maugsten

December, 2022

# Table of Contents

# List of Figures

# List of Tables

**Abstract**

TBD. This project tackles the problem of estimating the diffusion coefficient of a tracer spreading from the Cerebrospinal Fluid (CSF) and into the brain parenchyma. A Physics Informed Neural Network (PINN) was used to calculate the diffusion coefficient. The network was fed a set of 2D simulated MRI-like sliced brain images of the tracer concentration, and the 2D diffusion equation. The tuning of the network parameters was done on a quarter brain slice whose edges were smoothened-out. The optimal parameters were then used when running the full brain slice, both smoothened and unsmoothened, in the neural network. This brought about a diffusion coefficient that converged towards $1.010^{-4}\text{mm}^2/\text{s}$.

# 1 Introduction

Advancements in the emerging field of biological fluid mechanics has substantial motivation both in enhancing general clinical intuition (Stein et al. 2006), and to enable better patient specific diagnosis and treatment.

However, biological systems often involve complex and tortuous structures which spawn challenges for traditional numerical methods that require careful meshing of the geometries and making assumptions of the boundary conditions (Zapf et al. 2022). In addition, the applied general nonlinear partial differential equations (PDEs) like the Navier-Stokes equations are often unwantedly simplified which may lead to compromises and limit the complexity of the model, thus, leading to sub-par results. More over, shortcomings in our knowledge of the governing PDEs may also confine our models further.

Potential unknown biological processes also add to the uncertainty of predicting the correct behaviour of the fluid.

A method for solving such complex problems and a potential solution to the challenges listed above are physics-informed neural networks (PINNs) (Raissi et al. 2019). These artificial neural networks (ANNs) solve supervised learning tasks by combining data-driven machine learning (ML) methods as well as utilising prior knowledge about the physics of a system, in the form of PDEs and or boundary/initial condition, as a regularization agent that constrains the space of admissible solutions (Raissi et al. 2019). In addition, as large amounts of data that is usually need for ML tasks, it is sometimes difficult to acquire for various reasons, especially in the medical field. The physical constraints may be used to compensate for sparse data-sets. This combination may also lead to the discoveries of PDEs or unknown parameters and constants within them.

## 1.1 Motivation, Application Overview and Problem Statement

The cells in our brain, like any other cell in the body, produce metabolic waste. This waste must be cleared by some means, in order for the organ to function properly (Kaur et al. 2021), but the fact that the brain lacks a lymphatic circulation has puzzled scientists.

Alzheimer's disease is one of modern medicine's biggest challenges, yet surprisingly little is known about it. In 2015, it was the sixth leading cause of death in USA (Association et al. 2018). It is suggested that a leading cause of Alzeheimer's is the accumulation of metabolic waste proteins in the brain (Spires-Jones and Hyman 2014).

The idea of an alternate waste clearance system to the lymphatic one has been around since the mid 70's (Cserr and Ostrach 1974), but the hypothesis did not take off until the *glymphatic system* first was discovered in 2012 by the Danish neuroscientist, Maiken Nedergaard (Iliff et al. 2012). This system is a rapidly growing area of research. The hypothesis addresses whether the flow is governed only by diffusive transport, or if there is also convective (bulk) flow involved.

In hopes of getting a better understanding of the glymphatic system and to demonstrate the potential of PINNs we want to, in this study, apply a physics-informed neural network model to investigate the diffusive properties of a tracer molecule observed via MRI imaging over a period of 48 hours. We will for simplicity use synthetically made simulated MRI-like images with no noise alongside the diffusion equation to try to predict the concentration and estimate the diffusion coefficient of the tracer.

**Note:** The lecturer of FYS-STK4155 (M. Hjort-Jensen) was consulted for this project and he stated by email 09.12.22 that our project scope with ANN and PDE with Automatic Differentiation combined to a PINN was approved as well within the intent of including *at least two methods* for this project.

# 2 Theory

In this section we will describe the theory related to the glympathic system, which is our area of interest. We will then describe the methods we employ.

## 2.1 The Glymphatic System

The glymphatic system, first discovered by (Iliff et al. 2012), is a waste clearing system that utilizes a network of spaces surrounding blood vessels in the brain and cerebrospinal fluid (CSF) as means of transportation. One of its responsibilities is to extract the metabolic waste from within the brain and out to the lymphatic system.

Although it is believed that this system employs several means of fluid flow, for example both diffusion and pressure driven flow, we will assume only diffusive flow.

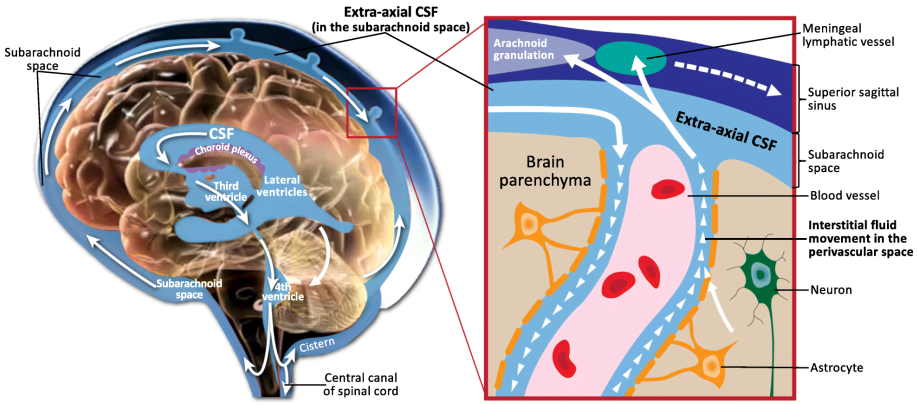


Figure 1: Diagram of CSF flow and interaction layers between different CSF compartments. IF in perivascular space in light blue and IF in brain parenchyma in beige (Shen 2018)

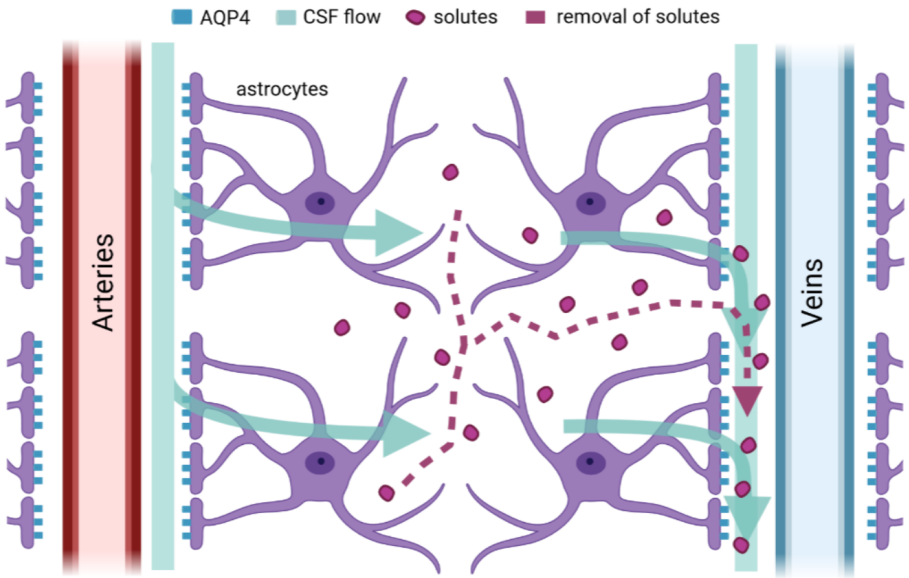


Figure 2: Close up view of the glymphatic system and CSF flow. The figure shows the CSF flowing along the arteries, through the AQP4 "feet" of the astrocytes and into the brain parenchyma (the substance of the brain, or the tissue that makes up the bulk of the brain) where the metabolic waste is "picked up" and extracted out along the veins (Franco Fernandez 2021).

## 2.2 The Diffusion Process

The governing equation for a diffusion process is the diffusion equation:

$$\frac{\partial c}{\partial t} = D\nabla^2 c, \tag{1}$$

where $c = c(x, y, t)$ is the concentration at position $(x, y)$ and time $t$, and $D$ is the diffusion coefficient, which we here assume is constant. This equation can then be used as the physical law we inform a PINN model studying diffusion processes.

## 2.3 ANNs

An Artificial Neural Network (ANN) is a type of machine learning algorithm which originally was inspired by the structure and function of the human brain. As described by the authors in an earlier project (Patey and Somer 2022) neural networks propagate information forward from neurons to neurons in layers, and then they learn by sending correction signals based on gradients backwards. After many learning iterations the model gets better, where the optimization is done with respect to minimising a convex cost function. The information flow forward is referred to as forward propagation. Here we have linear functions $z = Xw + b$ and then the output of a sum of such linear functions is the input of a non-linear activation function.

In a neural network this composition of functions takes form as an architecture with several nodes connected.

The final layer is called an output layer.

The information flow of gradients backwards is referred to as backwards propagation.

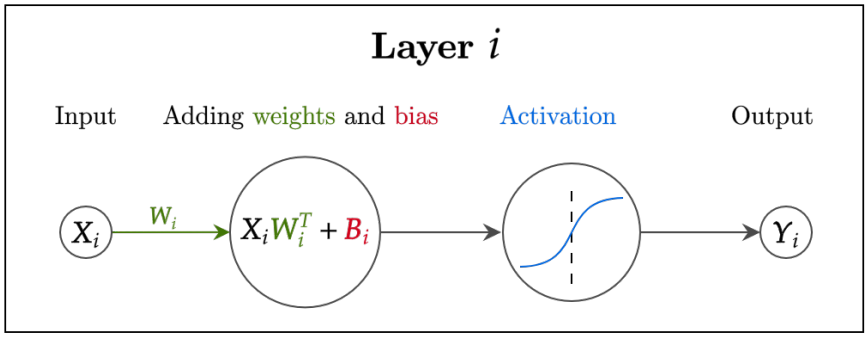


Figure 3: Illustration of a layer. It consists of a linear function (weights and biases) and a non-linear activation function.

In an ANN we have several functions that work in parallel. We call the first nodes the input layer nodes. The final outputs are the output layer nodes. In between we have what we refer to as hidden layers. Figure 4 shows an ANN with an input layer, two hidden layers and an output layer. The arrows pointing to each node in the figure illustrates forward propagation of data. The output of a node is an input to another node.

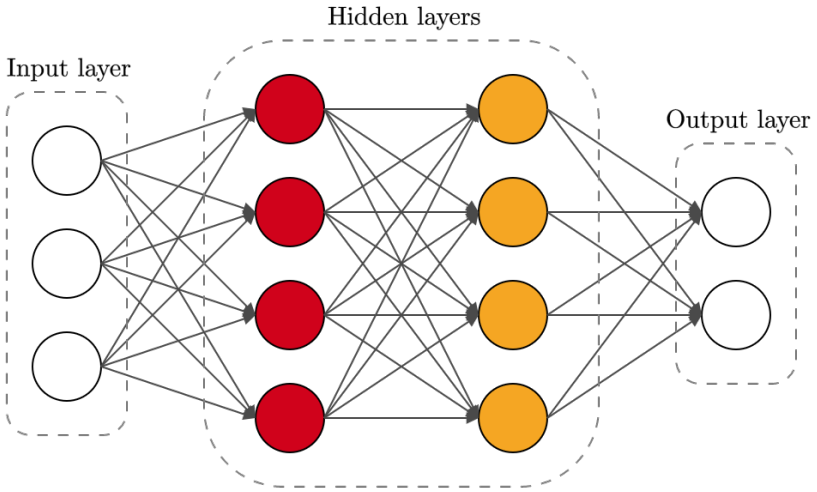


Figure 4: Illustration of an artificial neural network with two hidden layers.

Back propagation is illustrated in figure 5. By repeated use of the chain rule, the gradient is found going backwards. We will explore this in more detail in the next section of Automatic Differentiation.

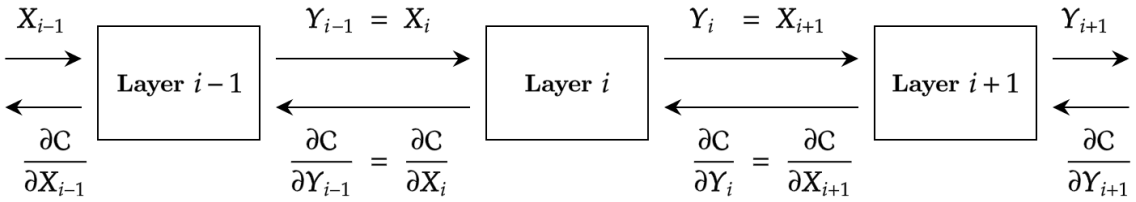


Figure 5: Forward and backward propagation. One layer's output becomes the next layer's input

## 2.4 Automatic Differentiation as Backpropagation

Automatic differentiation enables gradients of arbitrarily complicated loss functions to be computed accurately, and it differs from numeric and symbolic differentiation. In order to see this in context, let us first briefly describe numeric and symbolic differentiation.

With numeric differentiation we approximate the gradients by setting a small step size $h$, and calculate the approximate derivatives of $f'(x)$, using finite differences, for example by the forward difference method:

$$f'(x) \approx D_+(h) = \frac{f(x+h) - f(x)}{h}$$

Drawbacks with this method are the computational cost as well as errors associated with the approximation. For this reason numerical differentiation is not suitable for deep neural networks (Ketkar and Santana 2017).

Symbolic differentiation is limited to using specifically defined rules to differentiate a mathematical expression. For more complicated functions this can lead to very large and complicated expressions for the derivatives and inefficient code.

The limitations with these methods will be magnified as we try to calculate higher derivatives, and the partial derivatives of multivariate functions, which is what we need for our gradient methods in neural networks.

Automatic differentiation overcome these issues and hence it is the chosen method for calculating the gradients in ANNs.

Automatic differentiation uses the property that functions can be expressed as compositions of elementary functions where the derivative functions are known. We can then apply the chain rule repeatedly on these composite functions.

Secondly it uses the property that we can evaluate the derivatives for a given set of input values. By this we can avoid closed form mathematical expressions such as those provided in symbolic differentiation. With automatic differentiation we can also add computational procedures such as if-else statements and for loops. This provides enhanced flexibility in code implementation, whilst maintaining a fast and accurate calculation.

For neural networks we will be use back propagation which is one form of automatic differentiation. It belongs to the class of techniques referred to as reverse mode accumulation (Goodfellow et al. 2016).

For backwards propagation given a function f(x) and an input value x, automatic differentiation computes the derivative df/dx by applying the chain rule to the sequence of operations that define the function f. This involves starting with the output of the function and working backwards through the sequence of operations to compute the derivatives of each intermediate step.

Mathematically, let f be a function that maps input values x to output values y, and let g be a sequence of elementary functions (such as addition, multiplication, and exponentiation) that define f. Then the derivative of f with respect to x can be written as:

$$df/dx = (dg_n/dx) * (dg_{(n-1)}/dg_n) * ... * (dg_2/dg_3) * (dg_1/dg_2)$$

where $dg_i/dg_j$ is the derivative of $g_i$ with respect to $g_j$. By applying the chain rule repeatedly, the derivative of f with respect to x can be computed by propagating derivatives backwards through the sequence of operations that define f. This process is what we refer to as backpropagation.

### 2.4.1 Pytorch Autograd

One of the first implementations of automatic differentiation was in FORTRAN (Bischof et al. 1992). Since then, automatic differentiation has been incorporated into many other programming languages and libraries, including PyTorch, TensorFlow, and Theano. In the following we will describe the Pytorch version.

PyTorch is an open-source machine learning library for Python, based on the Torch library. It provides a wide range of algorithms for deep learning and performs automatic differentiation. PyTorch is widely used in natural language processing, computer vision, and other areas of artificial intelligence. It is developed and maintained by Facebook's AI research group (Paszke et al. 2019).

PyTorch's autograd package with automatic differentiation computes gradients automatically during the backward propagation of a neural network. The autograd package defines a set of functions that can be used to compute gradients of a function with respect to its inputs. These functions are called autograd functions, and they have the ability to keep track of the operations performed on their inputs during the forward pass of the network. When the backward pass is computed, these functions use the recorded operations to compute the gradients of the output with respect to the inputs.

## 2.5 PINNs

Neural networks are data greedy, which means they require a large amount of data to be trained properly. For many scientific and engineering applications such amounts of data may not be readily available. On the other hand we may have information on the physical laws for our study of interest. The method of PINNs is based on embedding such prior knowledge about the physical systems into a neural network and by this constrict the space of possible solutions to only those feasible, respecting the physical laws. With this we may be able to use less data. A simple example could be to inform the network about the law of energy conservation when we study the motion of a pendulum.

### 2.5.1 Inverse Problem

In physics-informed neural networks (PINNs), an inverse problem refers to the process of using data or observations to infer the underlying physical parameters or processes that generate the data. This is in contrast to a forward problem, which involves using a set of known physical parameters to predict the resulting data or observations (Raissi et al. 2019).

Inverse problems are common in many fields, including physics, engineering, and geoscience. They can be challenging to solve because they often involve finding the solution to a system of nonlinear equations that may have multiple solutions or may be ill-posed. PINNs can be used to address these challenges by combining the power of neural networks as universal approximators with physical principles and constraints to solve inverse problems in a more efficient and accurate manner (Lu and Karniadakis 2020).

### 2.5.2 Cost Function for PINNs

In order for a PINN model to learn we need to define a cost function that the network can optimize on, minimizing a total cost. We here refer to the *cost* function as the mean sum of the loss for all data points, where the *loss* function is the loss for a single data point.

For PINN the loss function will have two main terms, one for the network's prediction error on the concentration data at $(x, y, t)$, and one for the residual error on the D in the diffusion process at a collocation point.

Collocation points are points at which the function is evaluated in order to approximate the solution to a partial differential equation (PDE). For a PINN, collocation points are used to evaluate the residual of the PDE at specific locations in the domain of the PDE. We will refer to these points as PDE points. The selection of these points can influence the accuracy of the final solution. These residual based refinement methods are used to discard points where the residual is low and add points where the residual is high. This is done to "absorb" more information where it is needed. This presses may occur in every iteration.

Finally these two loss terms will be summed, and alternatively also scaled with a weight, such that we can scale their relative importance.

First we use the diffusion equation to define the residual function $r(x, y, t)$ as:

$$r(x, y, t) := \frac{\partial c}{\partial t} - D\nabla^2 c. \tag{2}$$

Here the residual is zero if D is accurately predicted.

Taking the $L_p$ norm and with the mean sum over all the PDE points we then define the corresponding residual cost function as:

$$\text{Cost}_\text{r} = \frac{1}{N_r} \sum_{i,j,k}^{N_r} \| r(x_i, y_j, t_k) \|_{L_p}, \tag{3}$$

where $N_r$ is the number of PDE-points.

We then define the concentration data loss as:

$$\text{Cost}_\text{c} = \frac{1}{N_c} \sum_{i,j,k}^{N_c} \| c_{nn}(x_i, y_j, t_k) - c_d(x_i, y_j, t_k) \|_{L_p}, \tag{4}$$

where $c_{nn}(x_i, y_j, t_k)$ is the neural network-predicted concentration at $(x_i, y_j, t_k)$, and $c_d(x_i, y_j, t_k)$ is the actual data sample concentration at the same point. $N_c$ is the number of data-points. $L_p$ refers to the norm.

Note that the boundary conditions are not included in the PINN loss function as they are often not required for inverse problems (Zapf et al. 2022). The initial conditions are simply the first image at $t = 0$.

The total cost is given by

$$\text{Cost}_\text{tot} = w_\text{d} \text{Cost}_\text{c} + w_\text{r} \text{Cost}_r, \tag{5}$$

where $w_\text{d}$ is the weight for the concentration data loss and $w_\text{r}$ is the weight for the residual loss. Without loss of generality we can set $w_\text{d} = 1$ and only scale the relative importance of this two cost terms by the weight $w_\text{r}$. This weight is typically referred to as a scaling factor for the penalty term. The unbiased choice is $w_\text{r} = 1$. It will also act as a regularization term for the PINN.

If this scaling factor is too small, the penalty term may have little effect on the final weights of the network. On the other hand, if the scaling factor is too large, the penalty term may dominate the cost function and cause the network to over-constrain the predicted outputs.

In general, it is therefore relevant to tune this weight in order to balance the trade-off between enforcing the physical constraints and minimizing the overall error between the predicted and true outputs. This may involve some trial and error, as the appropriate weight can depend on the specific problem at hand and the characteristics of the training data.

The standard PINN (Raissi et al. 2019) uses equal weights for the PDE and boundary loss terms, and there is recent research in alternative methods to scale the weights (Sukumar and Srivastava 2022).

The norm used can also be selected differently for these two terms in order to let outliers have different importance. An example could be to use the L2-norm for the neural network loss and L1-norm for the PDE loss term. This can affect the stability of the training of a PINN (Zapf et al. 2022).

With this cost function we can then create an optimization problem where the parameters of the neural networks $c_{nn}(x, y, t)$ can be learned by minimizing the total cost.

### 2.5.3 The PINN Network Architecture

The figure below shows the complete PINN architecture which consists of the ANN part and the PDE part. The output of the ANN is the input of the PDE. The total cost is the sum of the cost of the concentration and the weighted cost of the residual.

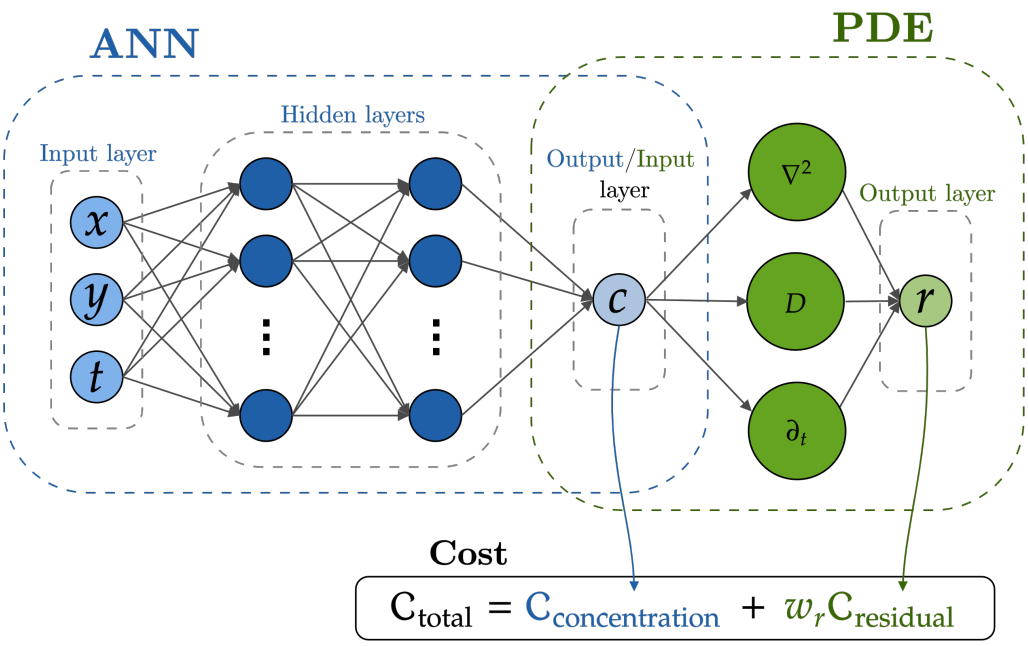


Figure 6: Illustration of the PINN. It consists of an artificial neural network (ANN) and a partial differential equation (PDE). The output of the ANN is the input of the PDE. The total cost is the sum of the cost of the concentration and the weighted cost of the residual.

## 2.6 Optimizers

There exist several optimizer algorithms, each with different strengths and weaknesses. The best optimizer depends on the problem at hand. For our PINN model two optimizers are of interest, ADAM and L-BFGS. We will here provide details of those two.

### 2.6.1 ADAM

The authors have experience with the ADAM optimizer in a previous project, where the properties of ADAM is investigated in more detail (Patey and Somer 2022). One of the benefits with ADAM is that due to the stochastic nature of SGD with ADAM it is less prone to get stuck in a local minima. It may however need longer time to converge than the next optimizer we will explore, the L-BFGS.

ADAM is named from adaptive moments (Kingma and Ba 2014).

The ADAM algorithm is here referred below from Algorithm 8.7 in (Goodfellow et al. 2016).

Algorithm **8.7** The ADAM algorithm

Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
(Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default: $10^{-8}$ )
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1 st and 2 nd moment variables $\boldsymbol{s} = \mathbf{0}, \boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
**while** stopping criterion not met do
  Sample a minibatch of $m$ examples from the training set $\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
  Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
  Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
  Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
  Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ ( operations applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
**end while**

ADAM optimization is very often the best choice (Géron 2017), but we will also note that for PINN the L-BFGS optimizer can be suitable. We will now discuss in more detail the properties of the L-BFGS.

### 2.6.2 L-BFGS

The L-BFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno) algorithm is a Quasi-Newton method for solving optimization problems, particularly those in which the objective function is convex and differentiable. It is an iterative method, meaning that it finds approximate solutions by repeating a set of steps until convergence is reached. L-BFGS has been the method of choice for PINN by some authors (Raissi et al. 2019).

Before introducing L-BFGS we need to define some underlying concepts, namely line search and the Wolfe condition. We will then provide the full BFGS algorithm. For further details the reader may refer to (Wright, Nocedal et al. 1999).

Line search is a method used in optimization to find efficiently a good step size (learning rate) $\alpha$ along the search direction vector $\mathbf{p}$ for an iterative optimization algorithm.

The iteration for step $k + 1$ is: $x_{k+1} = x_k + \alpha_k \mathbf{p}_k$.

The line search is inexact in the sense that there may be theoretically better step lengths, but it would take too much time to find it, so instead line search tries out a sequence of candidate values for $\alpha_k$, and terminating when certain conditions are satisfied.

Such a set of conditions suitable for Quasi-Newton methods are the Wolfe conditions.

The Wolfe conditions are the two conditions below:

1. The sufficient decrease condition: This condition requires that the objective function $f$ must decrease by a certain amount for a given step size. This ensures that the step is making progress towards the optimal solution. Mathematically this gives the inequality constraint:

$$f(x_{k+1}) = f\left(x_k + \alpha p_k\right) \leq f\left(x_k\right) + c_1 \alpha \nabla f_k^T p_k$$

For $c_1 \in (0, 1)$. A typical value for L-BFGS is $c_1 = 10^{-4}$. Geometrically the objective value

$f(x_{k+1})$ is then below the slope of the line of $l(\alpha)$ in figure 7.

The sufficient decrease condition is however not enough since this condition is satisfied for all sufficiently small values of $\alpha$, as we can see from figure 7. To avoid too short steps we introduce a second requirement, called the curvature condition.

2. The curvature condition:

This condition evaluates the gradients of $f_k$ and $f_{k+1}$ and impose the following inequality constraint:

$$\nabla f(x_{k+1})^T p_k = \nabla f\left(x_k + \alpha_k p_k\right)^T p_k \geq c_2 \nabla f_k^T p_k$$

For $c_2 \in (c_1, 1)$. A typical value for L-BFGS is $c_2 = 0.9$.

Assume this condition is not met. Then it means the new gradient is still steeper than the factored gradient on the right hand side, but then we can reduce f significantly by moving further to a larger step size in our line search. This explains the logic behind this condition, it avoids a too short step. This is seen geometrically as slope requirements in figure 7, where the interval of acceptable steps meeting both Wolfe conditions are shown.

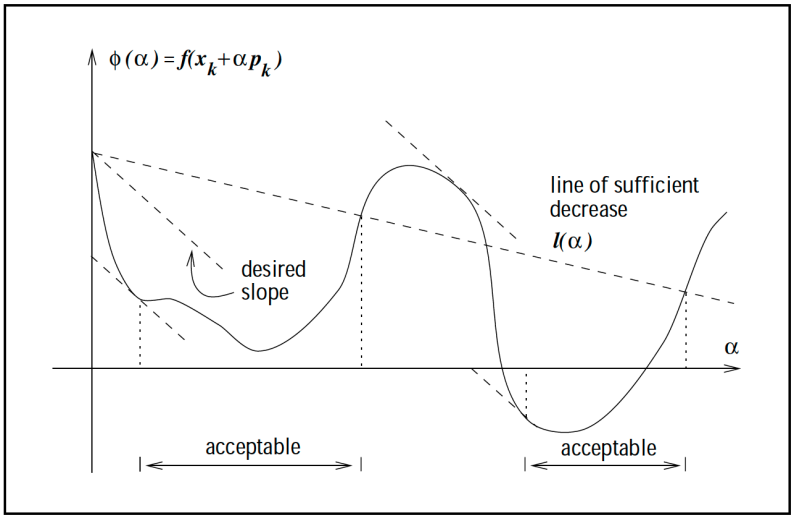


Figure 7: Wolfe conditions (Wright, Nocedal et al. 1999)

In addition to the Wolfe conditions we see that if we move too far the gradient could become positive, and we can limit our line search here by restricting the positive values, simply by taking the absolute values of the gradient in the curvature condition. This set of constraints is then referred to as the strong Wolfe conditions:

$$f\left(x_k + \alpha_k p_k\right) \leq f\left(x_k\right) + c_1 \alpha_k \nabla f_k^T p_k$$
$$\left|\nabla f\left(x_k + \alpha_k p_k\right)^T p_k\right| \leq c_2 \left|\nabla f_k^T p_k\right|,$$

For $c_1 \in (0, 1)$ and $c_2 \in (c_1, 1)$. We will use the strong Wolfe condition in the PyTorch implementation of L-BFGS, which uses the default values $c_1 = 10^{-4}$ and $c_2 = 0.9$.

We have discussed the step size $\alpha$. Let us then see how the direction vector $\mathbf{p}$ is computed.

Here BFGS uses an approximation $B_k$ of the true Hessian matrix H, which is the matrix of second-order partial derivatives of the objective function. The $B_k$ matrix describes approximately the curvature of the objective function, and the search direction is chosen to be the negative of the

direction of steepest descent, which is given by the equation $p_k = -B_k^{-1}\nabla f_k$, where $\nabla f_k$ is the gradient of the objective function.

For notation we define the following vectors:

$$s_k = x_{k+1} - x_k = \alpha_k p_k, \quad y_k = \nabla f_{k+1} - \nabla f_k$$

The **BFGS update rule** which updates $B_{k+1}$ can be shown to be (Wright, Nocedal et al. 1999):

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}.$$

With this update rule we arrive at the following algorithm:

**BFGS Method Algorithm**(Wright, Nocedal et al. 1999):

Given starting point $x_0$, convergence tolerance $\epsilon > 0$, Hessian approximation $B_0$; $k \leftarrow 0$;

**while** $\|\nabla f_k\| > \epsilon$;

Compute search direction

$$p_k = -B_k^{-1}\nabla f_k;$$

Set $x_{k+1} = x_k + \alpha_k p_k$ where $\alpha_k$ is computed by line search satisfying Wolfe conditions.

Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$;

Compute $B_{k+1}$ by the **BFGS update rule** above.

$k \leftarrow k + 1$;

**end (while)**

Each iteration can be performed at a cost of $O\left(n^2\right)$ arithmetic operations (plus the cost of function and gradient evaluations).

The algorithm is robust, and its rate of convergence is superlinear, which is fast enough for most practical purposes. Even though Newton's method converges quadratically, its cost per iteration is usually higher, because of the need to calculate second derivatives and solution of a linear system.

We have described the BFGS algorithm, the limited-memory version of it, L-BFGS, stores only the $m$ most recent correction pairs $(s_i, y_i)$, and discards the other vector pairs.

The main idea is that curvature information from only the most recent iterations are likely to be relevant to the actual behavior of the Hessian at the current iteration. Discarding the rest makes the L-BFGS algorithm relatively fast. The choice of $m$ is problem dependent, but values between 3 and 20 often produce satisfactory results. The main weakness of the L-BFGS method is that it converges slowly on ill-conditioned problems (Wright, Nocedal et al. 1999).

# 3 Method and Implementation

## 3.1 Data Handling

### 3.1.1 The Synthetic data

The data set in this project consists of 20 synthetic MRI images ranging from 0 to 48 hour with an interval of 2.4 hours between. Both are optimized at the same time. By letting $D$ be a tunable parameter and minimizing the total loss, we may get an estimate of $D$.

The synthetic data was provided by doctoral research fellow Bastian Zapf from the Department of Mathematics, UiO. The data was made by solving equation 1 using the finite element method (FEM) with boundary and initial conditions 6, 8 and with a diffusion coefficient $D_{init} = 0.36\text{mm}^2\,\text{h}^{-1} = 10^{-4}\text{mm}^2\,\text{s}^{-1}$ (Magdoom et al. 2019). For a more comprehensive description of how the data was made see (Zapf et al. 2022).

$$c(x,t) = g(x,t) \quad \text{on } \partial\Omega \times (0,T), \tag{6}$$
$$c(x,0) = 0 \quad \text{in } \Omega, \tag{7}$$

were,

$$g(t) = \begin{cases} 2t/T & \text{for } 0 \leq t \leq T/2 \\ 2 - 2t/T & \text{for } T/2 \leq t \leq T \end{cases} \tag{8}$$

## 3.2 Scaling

It is desirable to work a with governing equation on dimensionless form as it can among other things reduce the dimensionality of the parameter space. To do this, we need to scale the physical domain of the problem. The images of the cross section shows us the spacial domain $x \times y$ of the brain. The brained spanned 129mm in the x-direction and 162mm in the y-direction. We decided to train the network on a fourth of this domain, essentially splitting both the x- and y-axis into two equal parts. To scale the diffusion equation, we need to define characteristic lengths $X$ and $Y$. Let us take these lengths to be the size of the domain in the x-direction $X = 162\text{mm}/2 = 81\text{mm}$ and the size of the domain in the y-direction $X = 129\text{mm}/2 = 64.5\text{mm}$. We can now define dimensionless lengths $x^* = x/X$ and $y^* = y/Y$. There are twenty images, all of which are spaced 2.4 hours apart, resulting in a total timespan of 48 hours. Let us take this timespan to be our characteristic timelength $T = 48h = 172800s$. We can now define a dimensionless time $t^* = t/T$. By substituting our definitions of the dimensionless variables into the governing equation, we obtain the dimensionless diffusion equation

$$\frac{\partial c}{\partial t^*} = D\frac{T}{XY}\left(\frac{\partial^2}{\partial x^{*2}} + \frac{\partial^2}{\partial y^{*2}}\right)c. \tag{9}$$

Note that we could introduce a characteristic concentration here, but we save this scaling for later as it does not affect the dimensionless number $d = DT/XY$ which governs the diffusive transport. When the PINN is learning, it is actually this dimensionless number that it will try to estimate. The diffusion coefficient is later determined by upscaling the dimensionless number by the characteristic lengths $D = d \cdot XY/T$.

Scaling the concentration was not necessary for finding a dimensionless physical domain or determining the dimensionless number, but it is important for the PINN as we obtained better results by centering the data. We therefore ended up scaling the concentration to values between -1 and 1.

### 3.2.1 Train and Test Data

As for all machine learning methods we need to train the model on some data and test on other data. A conventional way is to split the train and test data into 80% and 20% of the total data, respectively. For PINNS, the case is a bit different. As far as we know there is no conventional way of splitting the train and test data. By consulting with Bastian Zapf (previously mentioned), the choice landed on using 2 out of 20 pictures as test data. The two pictures were picture number 8 and 16, corresponding to the times 16.8 hours and 33.6 hours, respectively.

Note that the loss is evaluated differently for train and test data:

$$\text{total loss train} = \text{train data loss} + \text{weight} \times \text{PDE train loss} \tag{10}$$

$$\text{total loss test} = \text{test data loss} \tag{11}$$

## 3.3 Implementing The PINN

The neural network was constructed using a class that inherited from Pytorch's torch.nn.Module, and was defined as follows,

```
""" The Neural Network """
NN_PINN = Net(num_hidden_units=32, num_hidden_layers=5,\
          inputs=3, inputnormalization=True).to(device)
```

Thus, the network consisted of 5 hidden layers and 32 hidden units/nodes in each layer.

The total loss is defined as follows

```
# Forward pass for the data:
train_data_loss_value = data_loss(NN, train_input_list, train_data_list,
↪  loss_function_NN)

# Forward pass for the PDE
train_pde_loss_value = pde_loss_residual(train_pde_points, NN, D_param,
↪  loss_function_PDE)

train_total_loss = train_data_loss_value  + pde_w * train_pde_loss_value
```

where the data loss is defined as

```
def data_loss(nn, input_list, data_list, loss_function_NN):
    loss = 0.
    count = 0
    for input_, data in zip(input_list, data_list):
        count += 1
        predictions = torch.squeeze(nn(input_))
        loss = loss + loss_function_NN(predictions, data)
    return loss/count
```

and the PDE loss is defined as

```
def pde_loss_residual(coords, nn, D, loss_function_PDE):

        assert isinstance(D, torch.nn.Parameter), "D should be a parameter of
        ↪  the network."
        assert coords.shape[-1] == 3, "Array should have size N x 3."

        coords.requires_grad = True
        output = nn(coords).squeeze() # Forward pass of coords through the
        ↪  network

        ones = torch.ones_like(output)

        # Take the gradient of the output
        output_grad, = torch.autograd.grad(outputs=output,
                                           inputs=coords,
                                           grad_outputs=ones,
                                           create_graph=True)

        doutput_dt = output_grad[..., -1]
        doutput_dx = output_grad[..., 0]
        doutput_dy = output_grad[..., 1]

        ddoutput_dxx, = torch.autograd.grad(outputs=doutput_dx,
                                            inputs=coords,
                                            grad_outputs=ones,
                                            create_graph=True)

        ddoutput_dyy, = torch.autograd.grad(outputs=doutput_dy,
                                            inputs=coords,
                                            grad_outputs=ones,
                                            create_graph=True)

        ddoutput_dxx = ddoutput_dxx[..., 0]
        ddoutput_dyy = ddoutput_dyy[..., 1]

        laplacian = (ddoutput_dxx + ddoutput_dyy)

        residual = doutput_dt - D * laplacian

        assert output.shape == residual.shape

        return loss_function_PDE(residual, torch.zeros_like(residual))
```

We selected the residual points uniformly between $-1$ and 1.

The ADAM optimizer was implemented as follows

```
# List of the neural network parameters
params_ANN = list(NN_ANN.parameters())

# The ADAM optimizer
optimizerPINN = torch.optim.Adam([{'params':params_PINN,"lr":learning_rate_NN},
                                  {'params': D_param, 'lr': learning_rate_D}])
```

and the L-BFGS optimizer was implemented as follows

```
# List of the neural network parameters and the diff. coeff. D parameter
params = list(NN_PINN.parameters()) + [D_param]

# The L-BFGS optimizer
optimizer = torch.optim.LBFGS(params, max_iter=max_epochs,
                                      tolerance_grad=1e-8,
                                      tolerance_change=1e-12,
                                      line_search_fn="strong_wolfe")
```

Note that the network's parameters and the diff. coeff. D in ADAM are two separate objects whilst they are one object in L-BFGS.

## 3.4 Testing PINN on a Known Function

The PINN was tested on a gravity problem, where an object is thrown on a parabolic trajectory, and the PINN is tasked to estimate the gravitational constant. This was done to verify that the network works. The PINN is physics-informed with Newton's second law of motion

$$\frac{d^2y}{dt^2} = -g. \tag{12}$$

The only force acting on the object is the constant gravity with the gravitational acceleration $g = 9.81\text{m/s}^2$. We accordingly define the PDE-loss to be $\frac{d^2y}{dt^2} + g$. The object is thrown upwards with an initial velocity $v_0 = \pi\text{m/s}$ at $y = 0\text{m}$. The true trajectory was calculated for the object until 1 second had passed. During that one second, 100 measurements of the position were made by taking the height and adding some noise uniformly distributed between -0.1 and 0.1. These measurements were split at random into a training set of 80 points and a test set of 20 points. The PINN learned from the training set until it reached convergence, and the result is shown in figure 8.

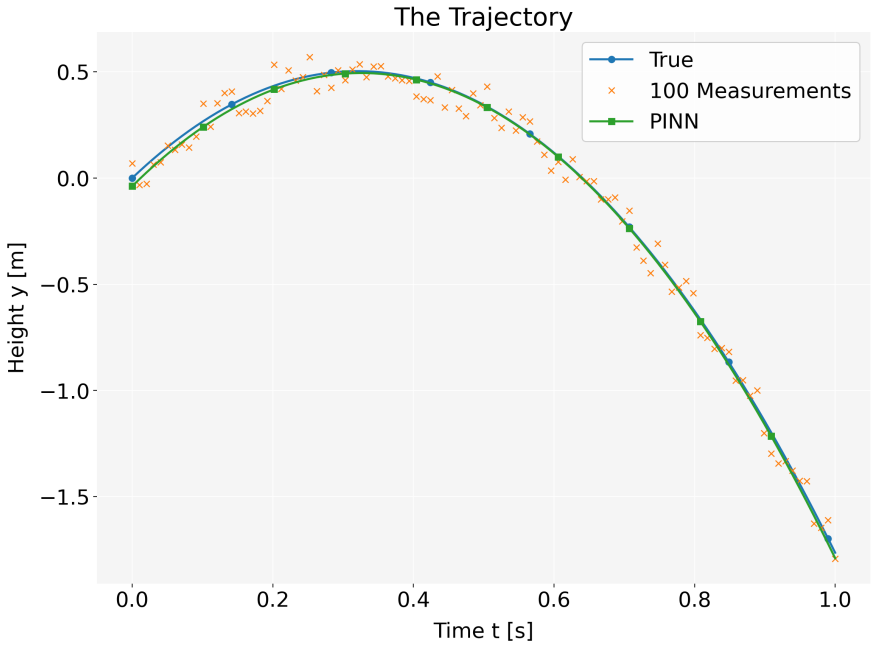


Figure 8: The figure shows true trajectory of the object and the 100 measurements made during the throw. It also shows the PINN's predicted trajectory.

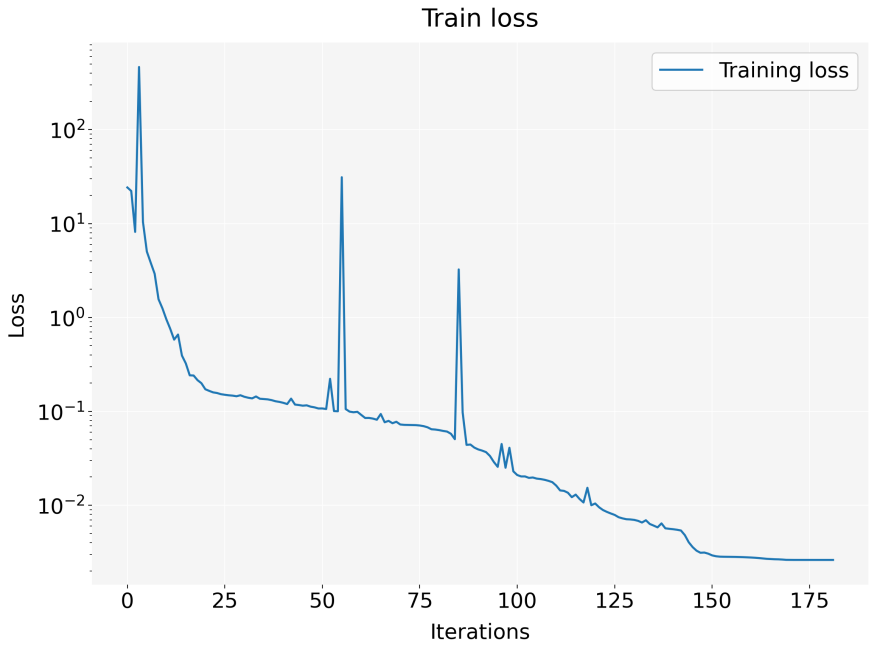


Figure 9: The figure shows the total loss from iteration to iteration. After about 180 iterations, the loss had converged.

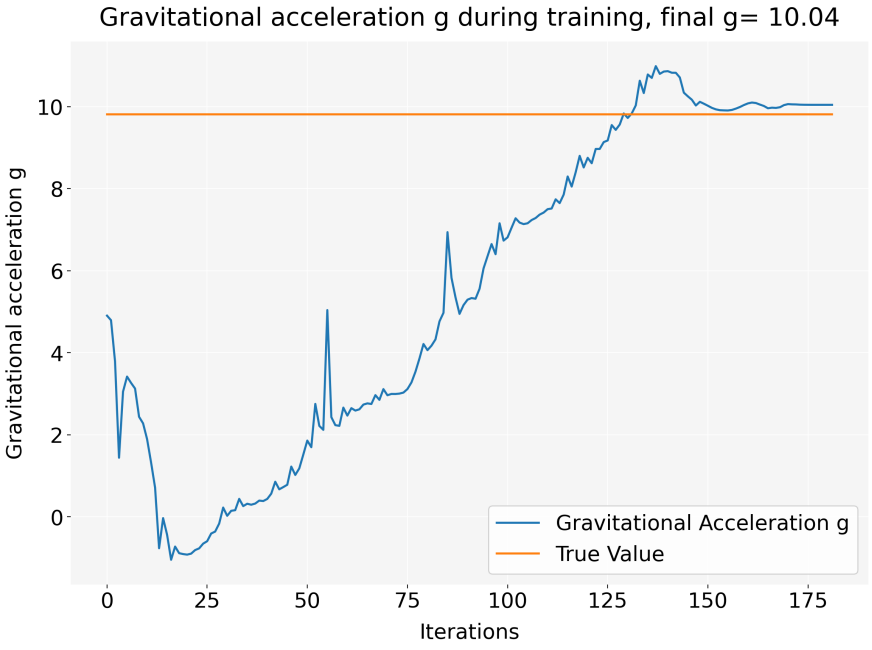


Figure 10: The figure shows the PINN's estimate of the gravitational constant $g$ from iteration to iteration. The initial guess was $g/2$. When convergence was reached, the acceleration was estimated to 10.04.

During the training of the network, there was a tradeoff between estimating the gravitational acceleration $g$ and predicting the trajectory of the object. There is an analogy to the diffusion problem here. When the PINN will be learning from the diffusion in the brain, there will be a tradeoff between estimating the diffusion coefficient and predicting the concentration of the tracer particles.

The results show that the PINN performs well for (at least) this type of physics problem, giving us a reason to proceed with this model.

## 3.5 Evaluating the Model

The model is evaluated by assessing the three main metrics

1. The data and PDE losses
2. The MRI concentration prediction images
3. The diffusion coefficient

Ideally, we want to find an estimation for the diffusion coefficient. In theory we do not know what this value is and therefore we can not only look at its convergence. We also need to look at the other metrics. If the data loss has converged and the predicted MRI concentration images look like the data at those timepoints, we can say that the neural network aspect of the PINN is reliable. Now, if the PDE loss and the diff. coeff. D also has converged at the same time we can with somewhat certainty say that we have found a candidate for the diff. coeff D.

This would be equivalent to

1. The total training loss,
2. The trajectory of the object,
3. The gravitational acceleration g,

in the gravity test.

## 3.6 Hyperparameters and Optimization

For our PINN we did a wide hyperparameter search first and then narrowed down on the most promising. Our search was based on some trial and error and heuristics. As described in the theory part, a unique feature of a PINN is the cost function with two terms, one for the neural net and one for the PDE residual. How these two terms are weighted was an important hyperparameter, as well as the selection of L1 or L2 norm on the PDE. Other hyperparameters the included the learning rate with or without an associated scheduler, the selection of either ADAM or L-BFGS optimizer, the number of hidden layers and neurons, the activation functions of the neurons and for the PDE the number of PDE collocation points. The selections involve trade offs between accuracy, stability and computational speed. ReLu or Leaky ReLu can not be used in PINNs involving second, or higher order PDE functions. There was a trade off between selecting the L1 or L2 norm and the weight for the PDE cost function.

| Hyperparameter | Wide Search | Medium Search | Narrow Search | Final Model |
|---|---|---|---|---|
| Hidden Layers (L) | 1,3 5,9 | 3,5,7 | 5 | 5 |
| Nodes in each L (N) | 16-128 | 32-64 | 32 | 32 |
| Epochs | 10-20000 | 20-100 | 20-40 | 25 |
| PDE points | 1e3 - 1e5 | 1e3 | 1e3 | 1e3 |
| PDE Weight ($W_r$) | 0-16 | 0-4 | 0.5-1.5 | 2.0 |
| PDE Norm ($W_r$) | L1,L2 | L1,L2 | L1 | L1 |
| Activation in HL | Sigmoid,TanH | TanH | TanH | TanH |
| Optimizer | ADAM,L-BFGS | ADAM,L-BFGS | L-BFGS | L-BFGS |
| Learning Rate | 1e-2,1e-5 | 1e-3,1e-4 | 1e-3 | 1e-3 |

Table 1: Search Space for PINN

# 4 Results

Here follows plots which within each subsection are sorted in the sequence of optimizer and then $W_r$. The number of epochs was varied as needed to convey relevant plots.

## 4.1 Costs

Here follows the cost (loss) plots. First is the plot that shows the Total, Data and PDE (TDP) cost terms.

Then follows the Test and Train (TT) cost, where it should be noted that the test cost only considers the data cost and not the PDE cost.

### 4.1.1 Cost Plot with ADAM

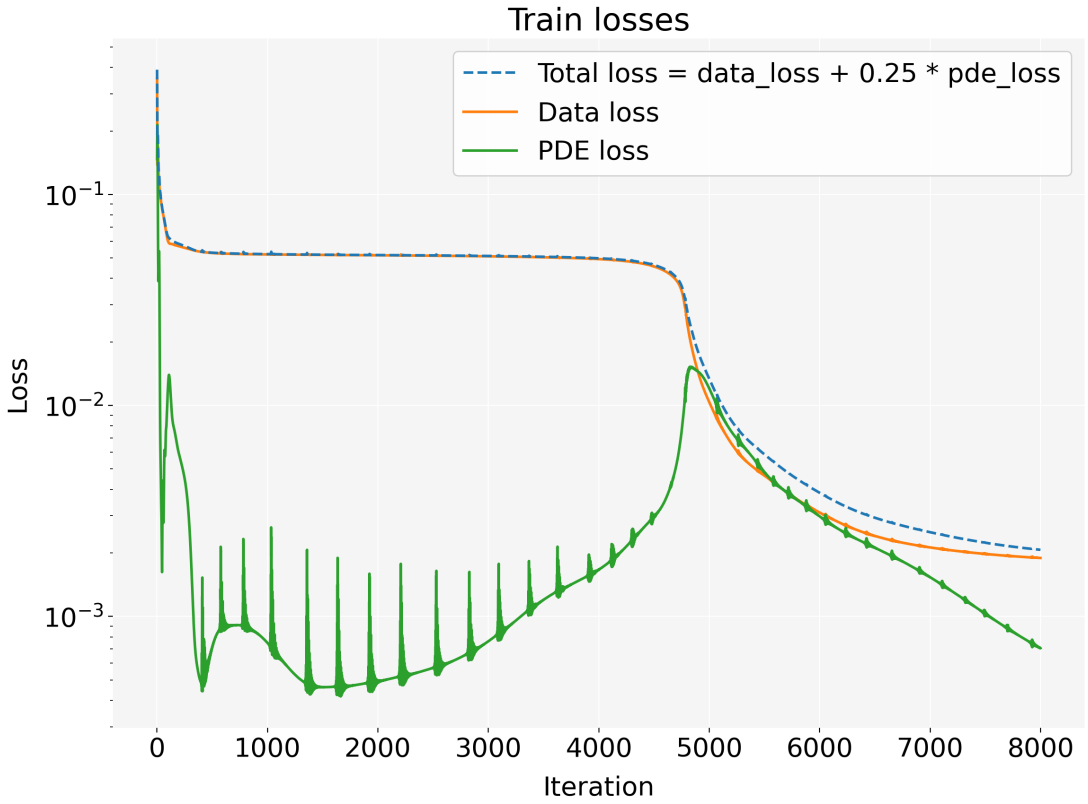


Figure 11: ADAM Total, Data and PDE Cost with $W_r = 0.25$ L2 norm.Epochs: 8000.


Figure 12: ADAM Test and Train cost with $W_r = 0.25$ L2 norm.Epochs: 8000.


Figure 13: ADAM Total, Data and PDE Cost with $W_r = 0.50$ L2 norm.Epochs: $10^4$.

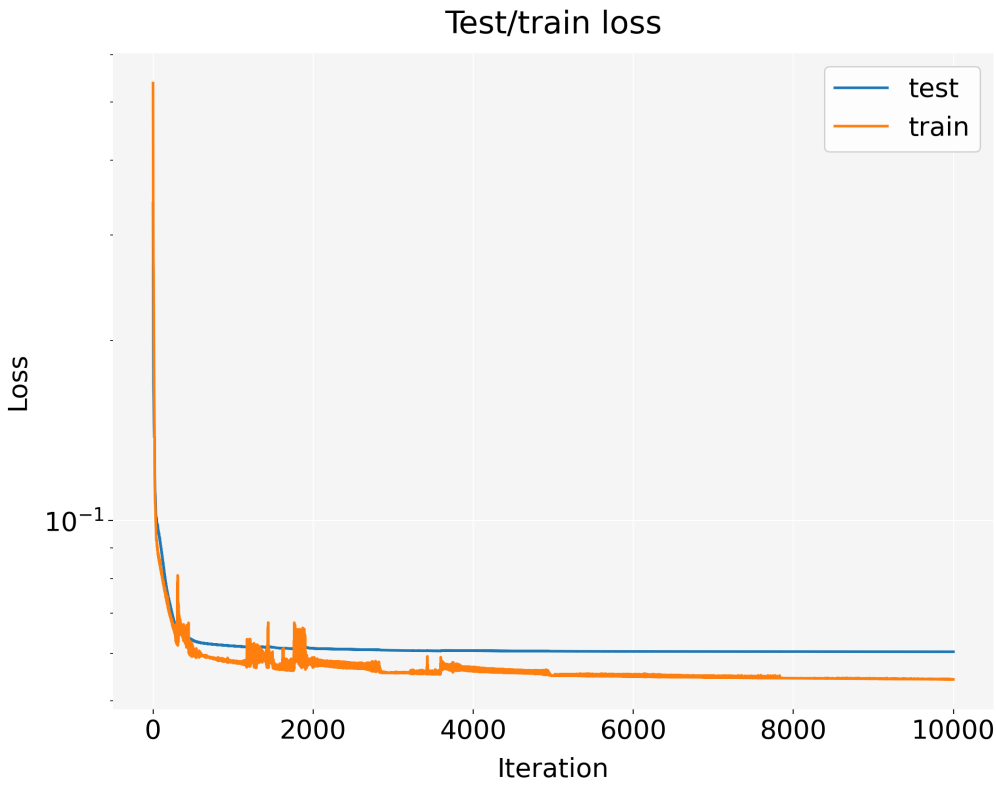


Figure 14: ADAM Test and Train cost with $W_r = 0.50$ L2 norm.Epochs: $10^4$.


Figure 15: ADAM Total, Data and PDE Cost with $W_r = 2.00$ L2 norm.Epochs: $2 \times 10^4$.

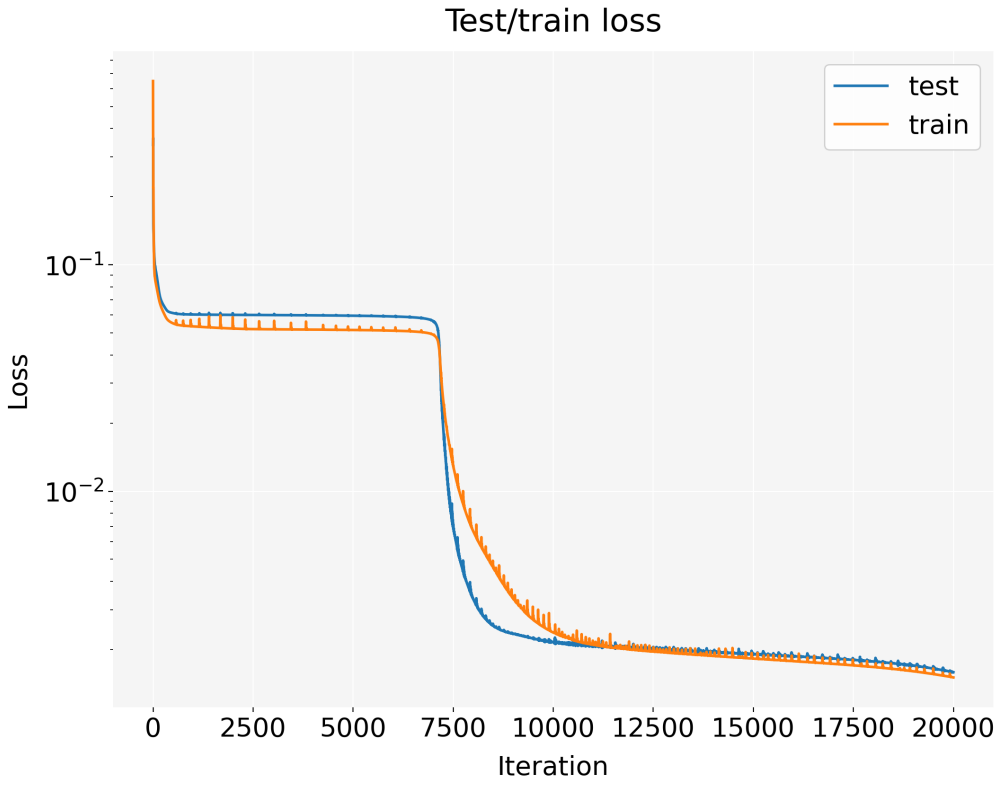


Figure 16: ADAM Test and Train cost with $W_r = 2.00$ L2 norm.Epochs: $2 \times 10^4$.

### 4.1.2 Cost Plot with L-BFGS

The L-BFGS algorithm was set to a high max epochs and stopped when convergence criteria was met, or a maximum of 8000 epochs, unless otherwise specified.

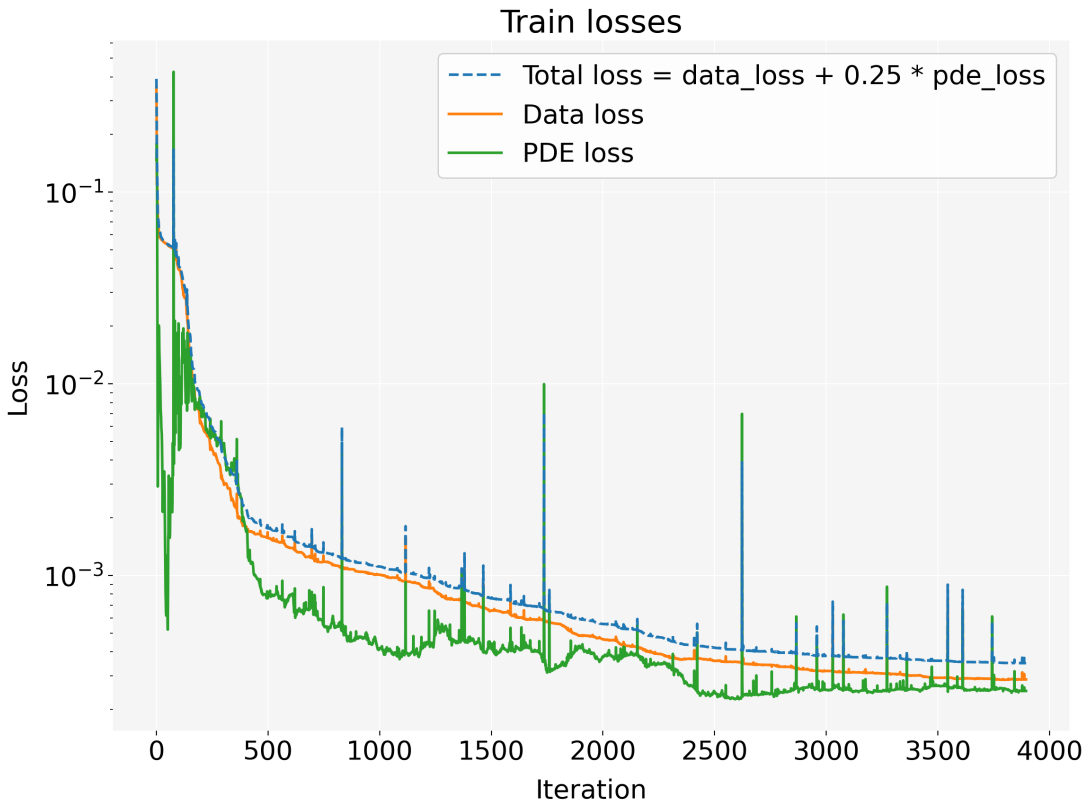


Figure 17: LFBGS Total, Data and PDE Cost with $W_r = 0.25$ L2 norm.

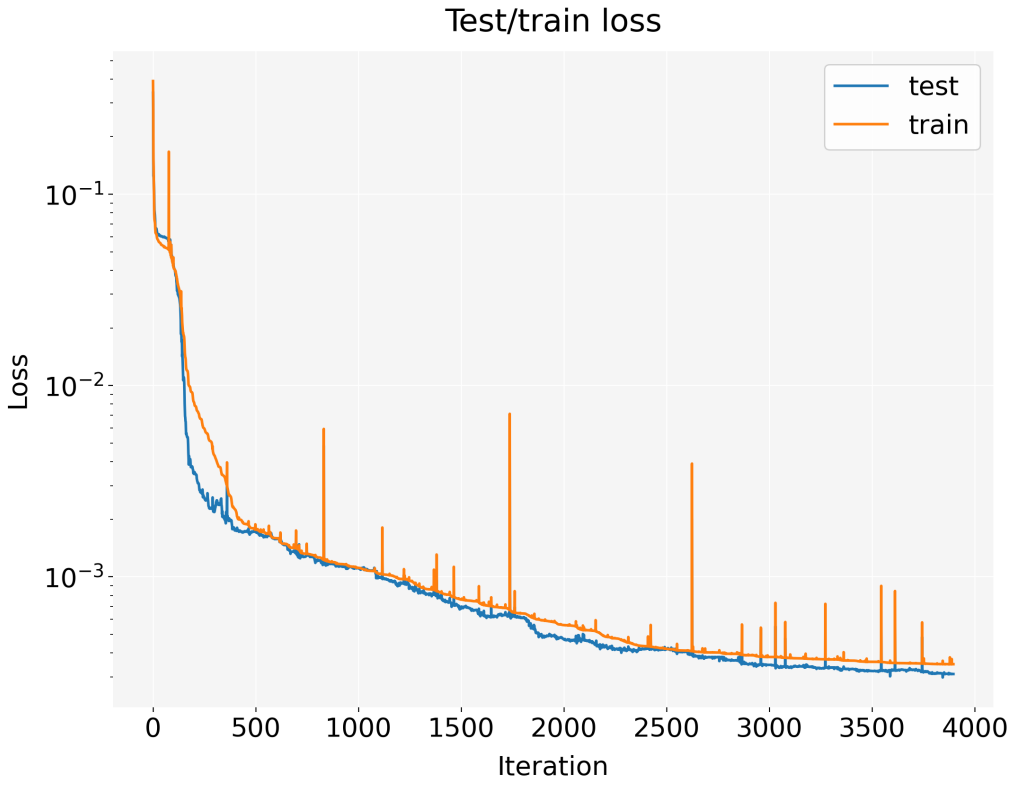


Figure 18: LFBGS Test and Train cost with $W_r = 0.25$ L2 norm.

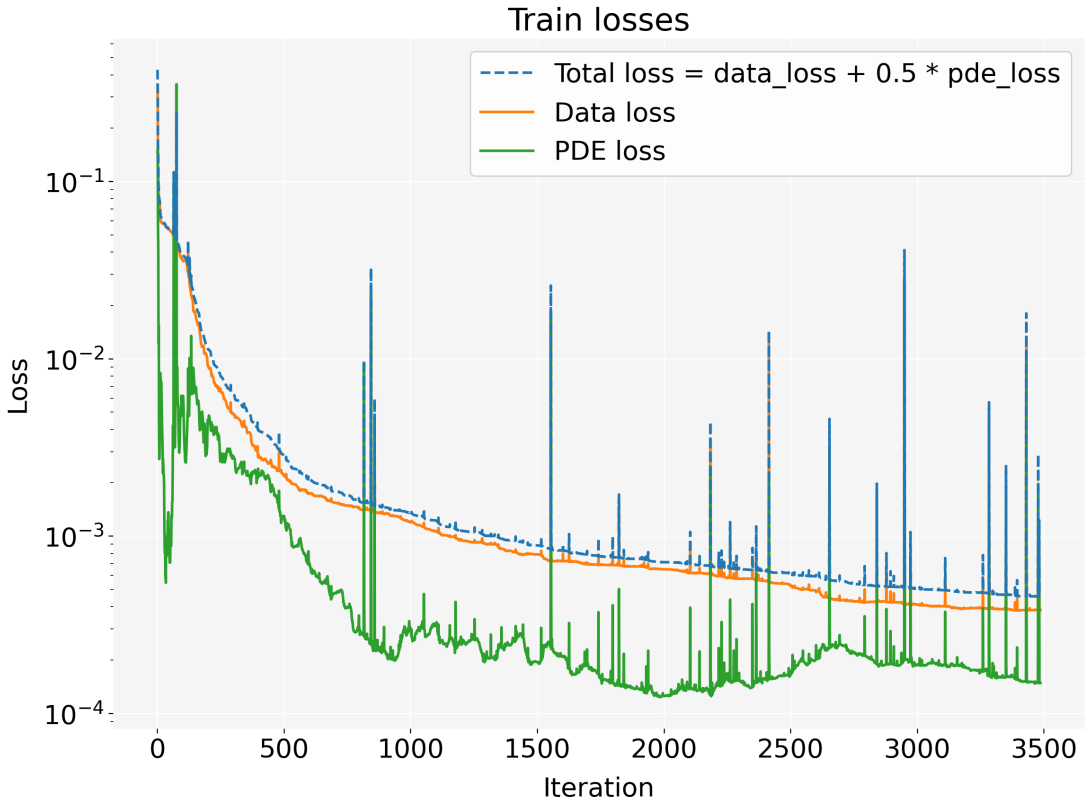


Figure 19: LFBGS Total, Data and PDE Cost with $W_r = 0.50$ L2 norm.

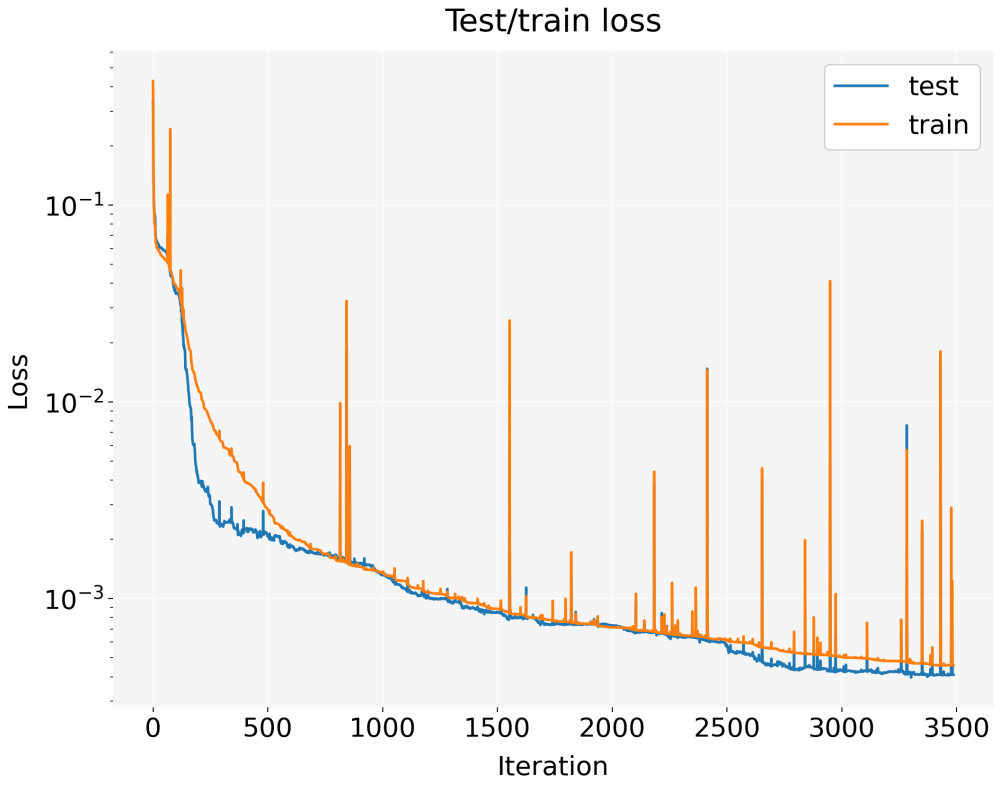


Figure 20: LFBGS Test and Train cost with $W_r = 0.50$ L2 norm.

## 4.2 The Concentration and MRI Image Replication

TBD skrive i Metode alle settings.

Here follows the Concentration and MRI Image Replication at two time steps, 16.8 hours and 33.6 hours, with the ANN prediction on the left side and the actual synthetic MRI data on the right hand side.

The plots are sorted in the sequence of optimizer, $W_r$, norm and number of epochs.

### 4.2.1 Concentration and MRI Image Replication with ADAM

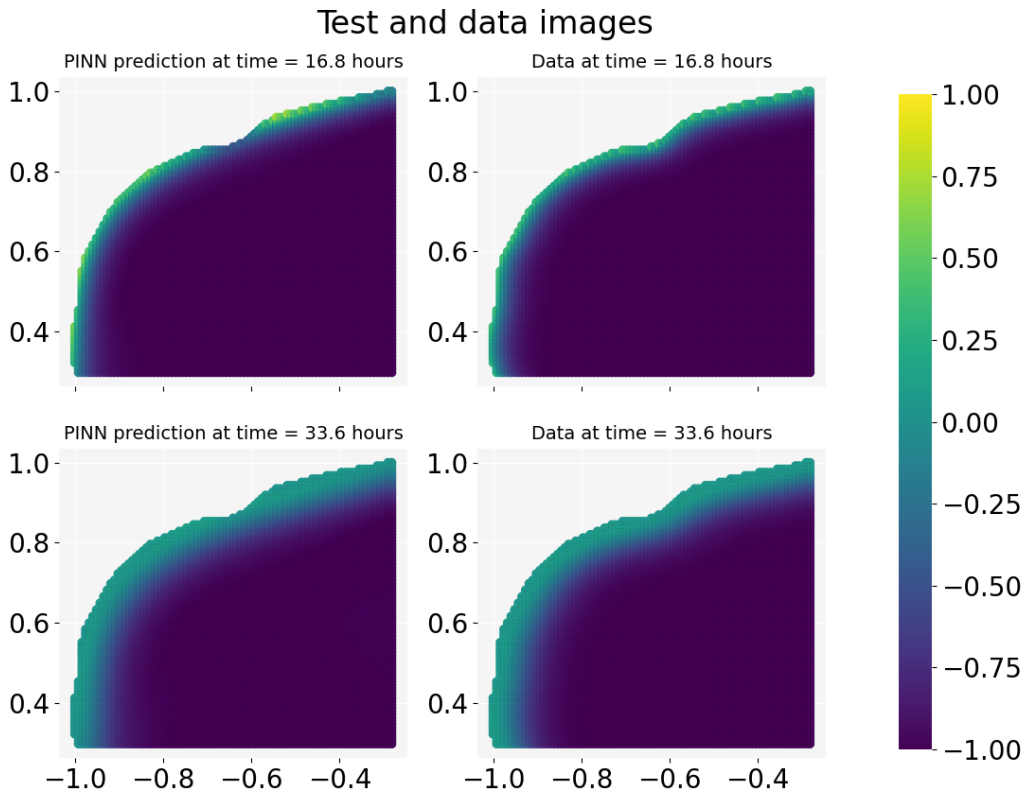


Figure 21: ADAM MRI replication with $W_r = 0.25$ L2 norm.Epochs: 8000. We note that at 16.8hours the PINN prediction misses some details in the dented edge zone. At 33.6 hours it seems to be a close resemblance to the actual data.

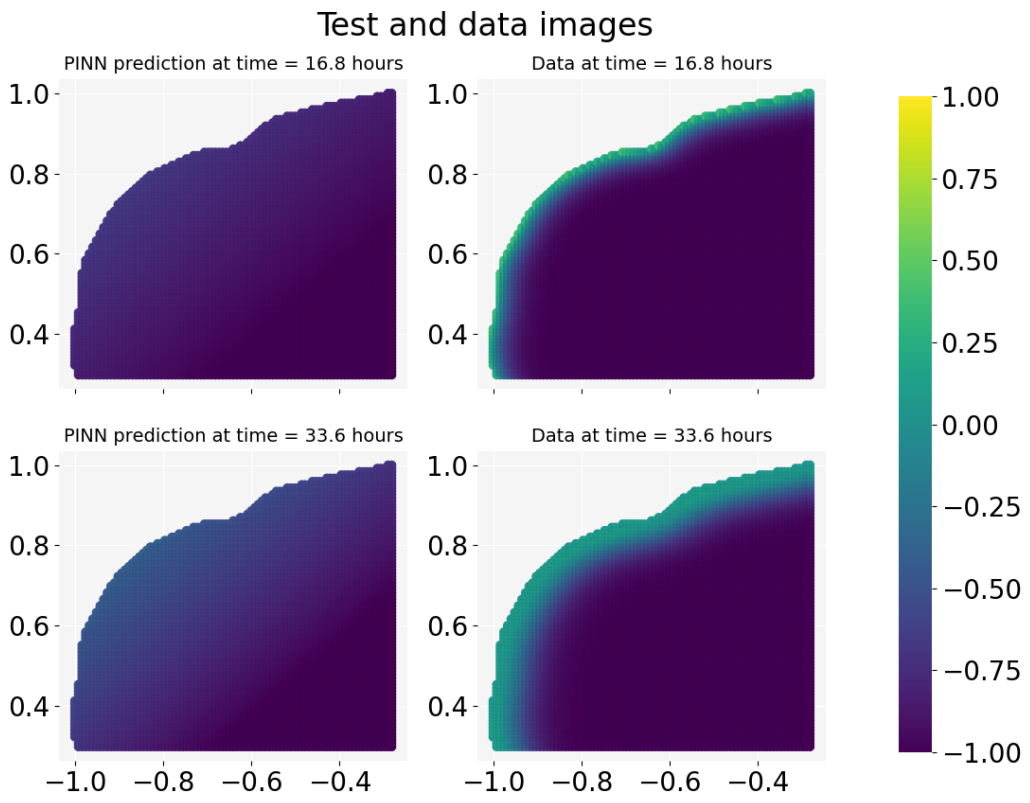


Figure 22: ADAM MRI replication with $W_r = 0.50$ L2 norm.Epochs: $10^4$. We note that at 16.8hours the PINN prediction completely miss the details from the actual data.

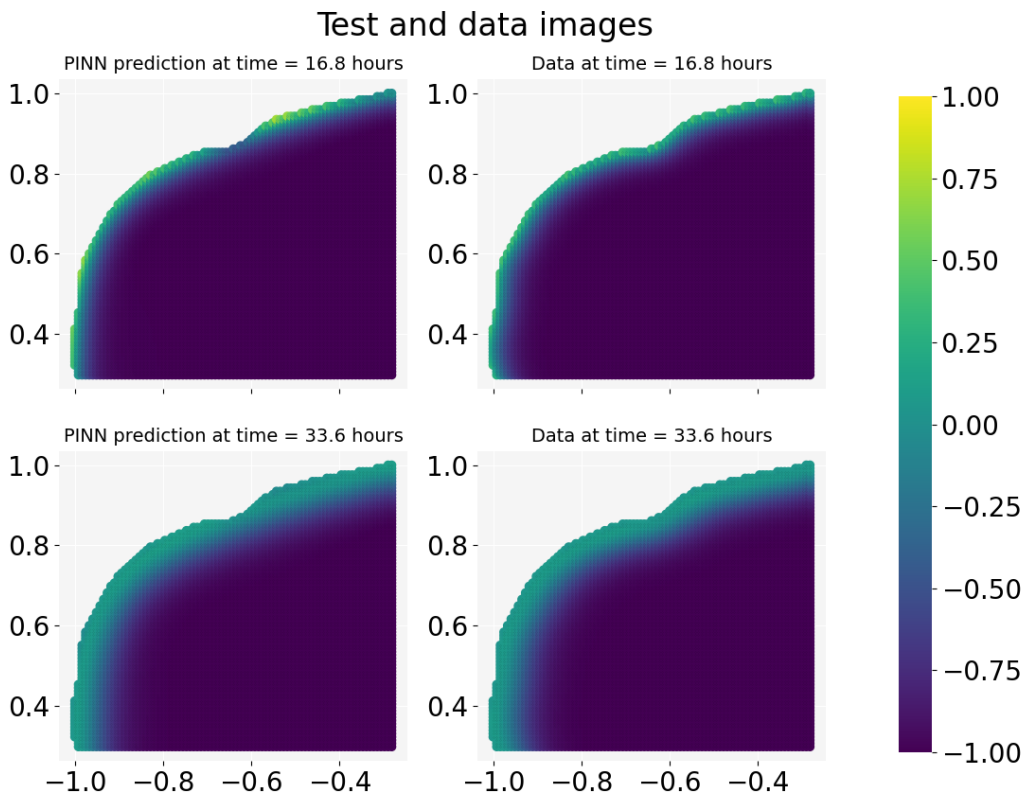


Figure 23: ADAM MRI replication with $W_r = 2.00$ L2 norm. Epochs: $2 \times 10^4$. We have increased to $2 \times 10^4$ epochs and note that at 16.8hours the PINN prediction misses some details in the dented edge zone. At 33.6 hours it seems to be a close resemblance to the actual data. This is similar to figure 21.

### 4.2.2 Concentration and MRI Image Replication with LFBGS

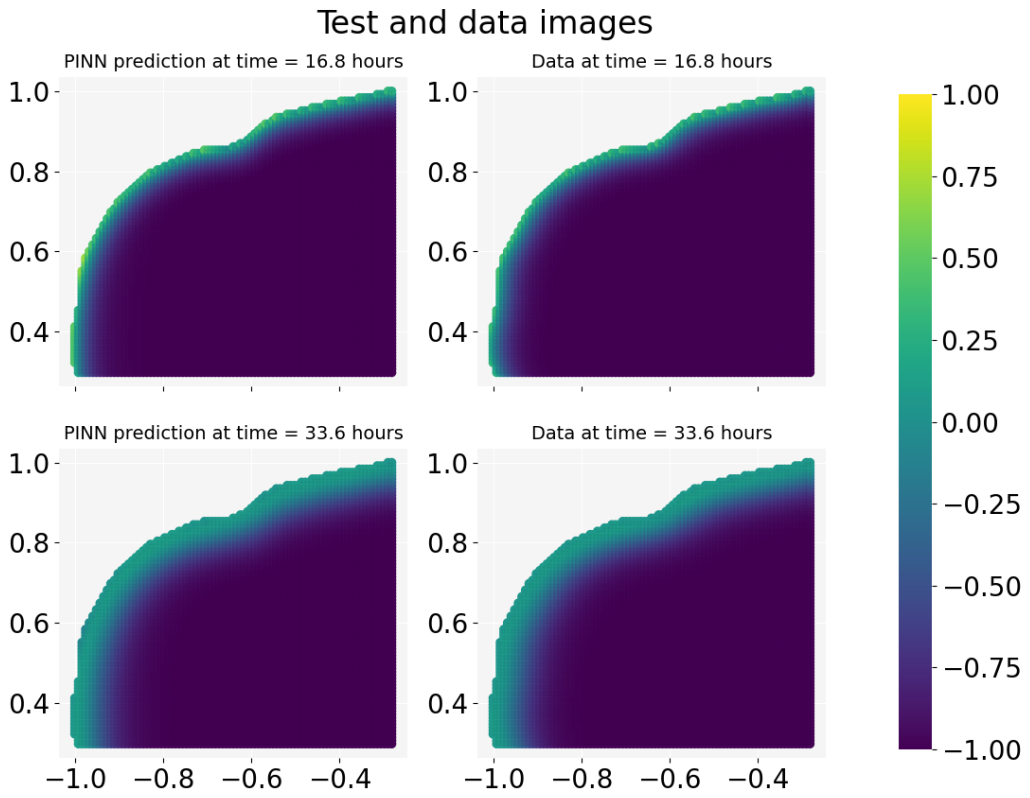


Figure 24: LFBGS MRI replication with $W_r = 0.25$ L2 norm. We note that at 16.8hours the PINN prediction has good detail even in the dented edge zone. At 33.6 hours it also seems to be a close resemblance to the actual data.

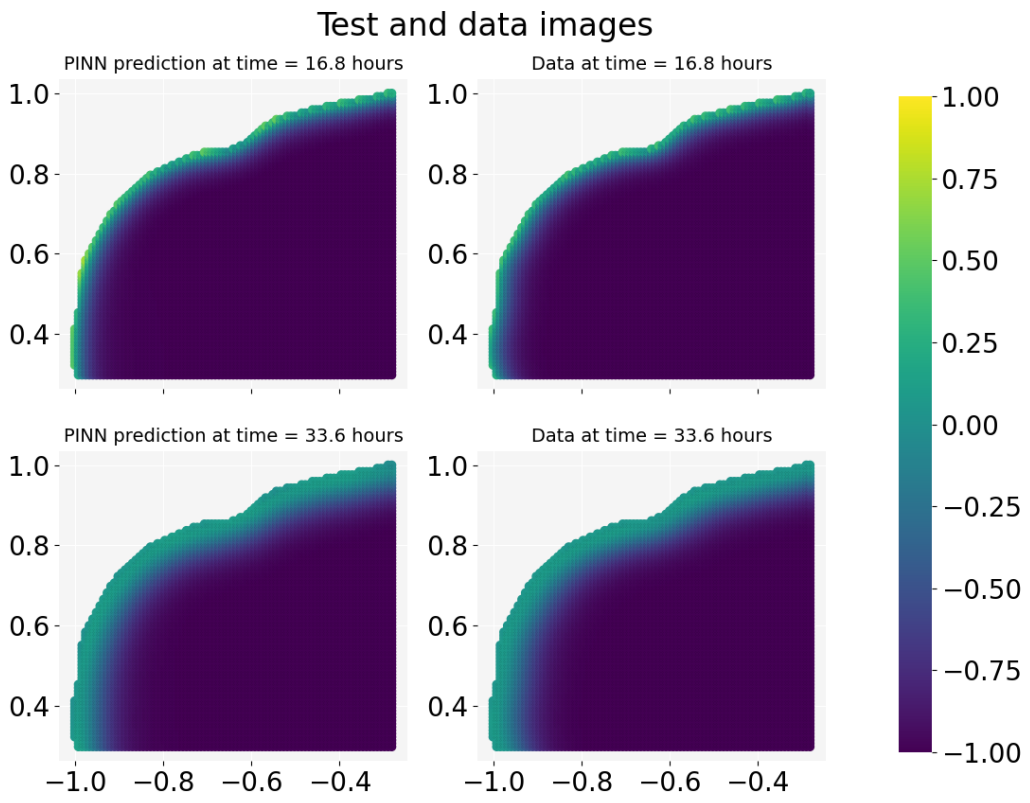


Figure 25: LFBGS MRI replication with $W_r = 0.50$ L2 norm. We note that at 16.8hours the PINN prediction has good detail even in the dented edge zone. At 33.6 hours it also seems to be a close resemblance to the actual data.

## 4.3 The Diffusion Coefficient

### 4.3.1 Diffusion Coefficient with ADAM

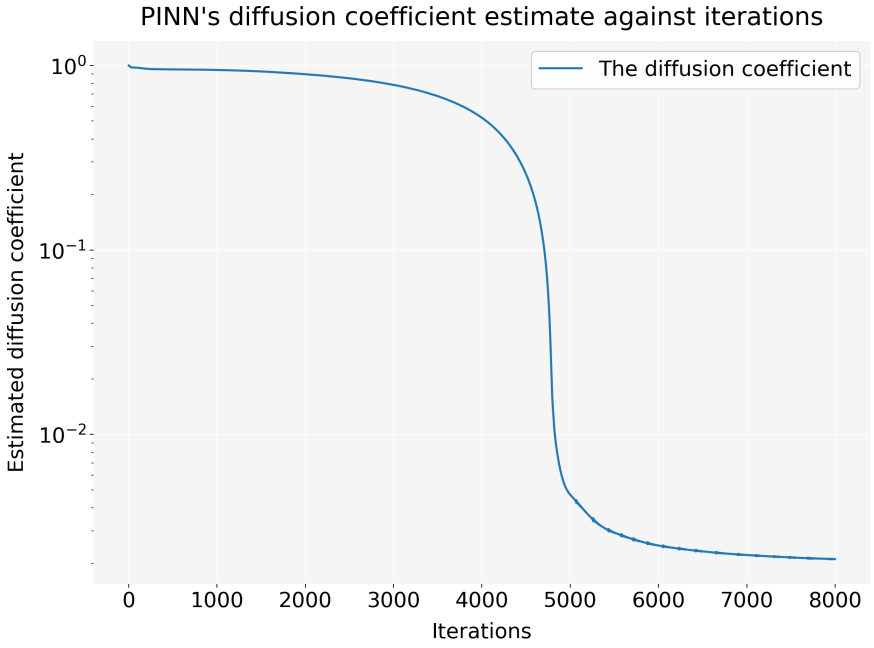


Figure 26: ADAM diffusion coefficient with $W_r = 0.25$ L2 norm.Epochs: 8000.Mean of last 50 iterations $D_{50\_mean} = 2.11 \times 10^{-3}$. Last iteration $D_{last} = 2.11 \times 10^{-3}$.

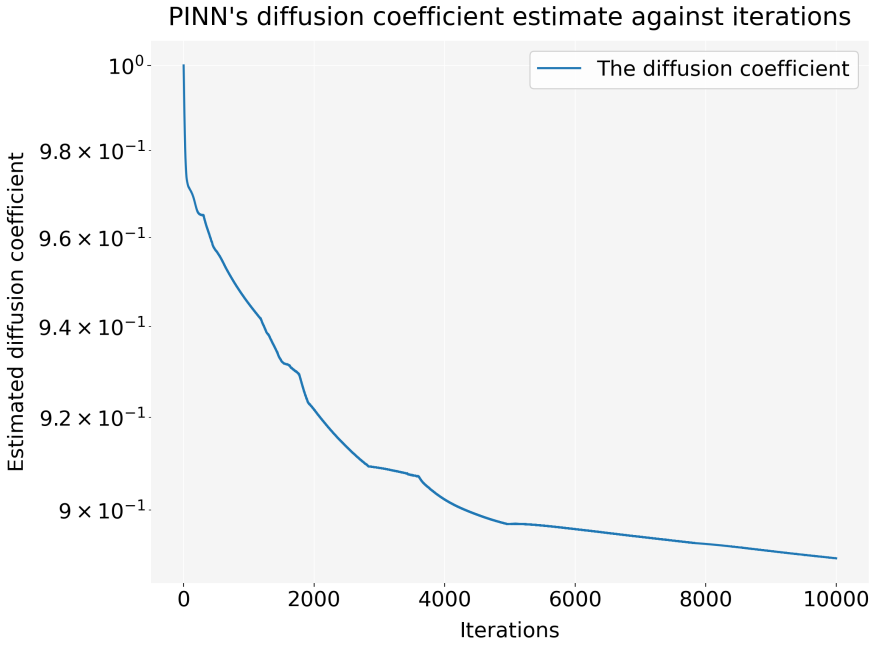


Figure 27: ADAM diffusion coefficient with $W_r = 0.50$ L2 norm.Epochs: $10^4$. Mean of last 50 iterations $D_{50\_mean} = 8.90 \times 10^{-1}$. We see that the diffusion coefficient did not converge.

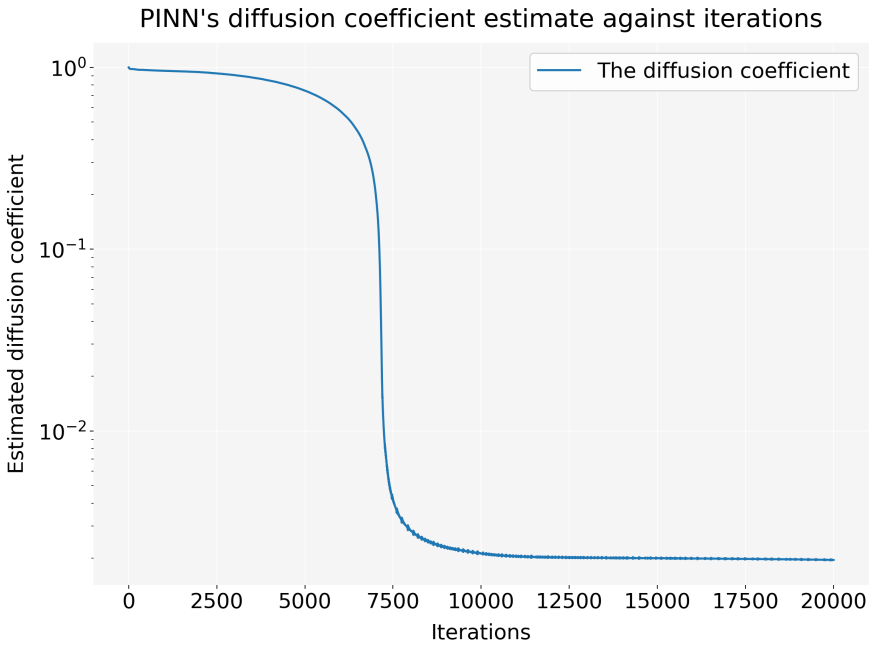


Figure 28: ADAM diffusion coefficient with $W_r = 2.00$ L2 norm.Epochs: $2 \times 10^4$. Mean of last 50 iterations $D_{50\_mean} = 1.76 \times 10^{-3}$. Last iteration $D_{last} = 1.76 \times 10^{-3}$.

### 4.3.2 Diffusion Coefficient with LFBGS

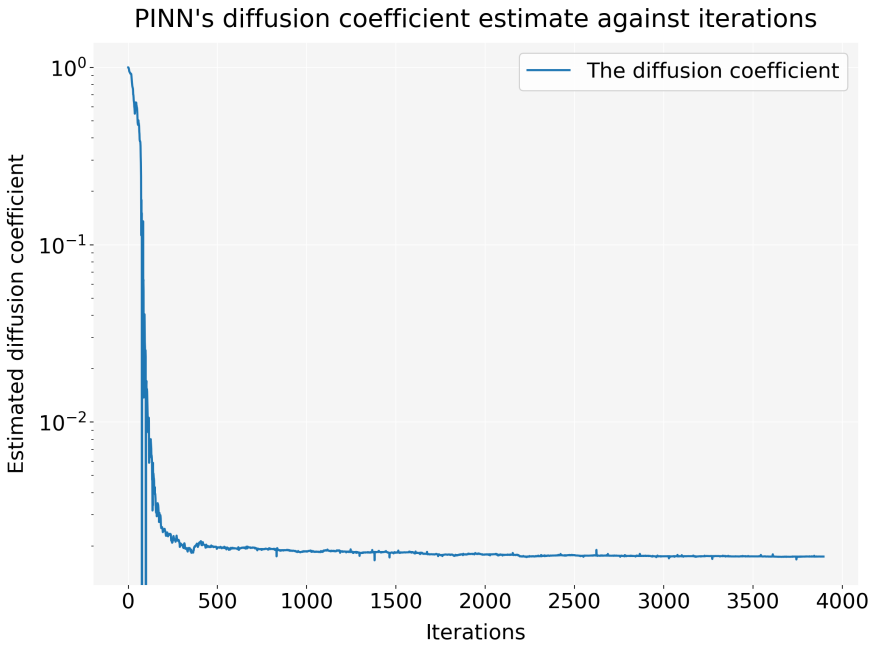


Figure 29: LFBGS diffusion coefficient with $W_r = 0.25$ L2 norm. Mean of last 50 iterations $D_{50\_mean} = 1.75 \times 10^{-3}$. Last iteration $D_{last} = 1.75 \times 10^{-3}$.

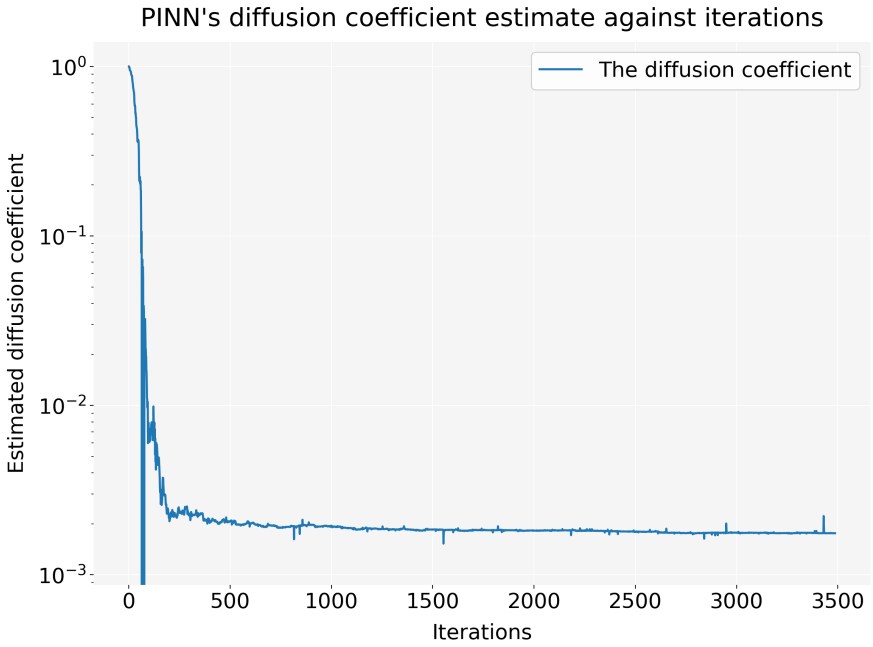


Figure 30: LFBGS diffusion coefficient with $W_r = 0.50$ L2 norm. Mean of last 50 iterations $D_{50\_mean} = 1.95 \times 10^{-3}$. Last iteration $D_{last} = 1.95 \times 10^{-3}$.

# 5 Discussion

## 5.1 The Effect of Hyperparameter Selection

We realized that it was hard to train a PINN, and there were many possible hyperparameters to tune. We saw comparable studies where hidden layers between 5-9 had been used, so we decided after some preliminary tests to limit our PINN model to 5 hidden layers.

## 5.2 Further Improvements

Our PINN model required a lot of computational power and we where therefore held back from exploring the hyperparameter space to the full extent. An improvement would be to further optimize the code for speed, as well as to get access to high performance computing.

For each run of the optimization we got slightly different results, and bootstrap could improve the confidence in the final result. However, for computational time issues, we did not use bootstrap on the hyperspace search runs, since each iteration run took several hours. For greater confidence in a final model, making B runs and taking the mean and variance of these B runs, could verify the confidence of the estimated diffusion coefficient D. (Efron and Tibshirani 1994)

# 6 Conclusion

In this project we studied how a physics informed neural network (PINN) can be used to determine the diffusion constant of the TBD system in the brain.

The PINN model provides a unique opportunity to regularize the network. In contrast to standard ANN where regularization is done by adding the norm of the weights in the cost function (Patey and Somer 2022) we now embed information on the physical process.

We learned that training a PINN is challenging, and requires a lot of tuning of hyperparameters. We first verified our PINN implementation successfully on a simpler problem, the parabolic trajectory, before we embarked on the much more complicated problem of finding the diffusion constant.

We noted that the choice of L1 or L2 regularization on the PDE term had significance, as well as the scaling of the PDE cost term.

We where able to train the PINN to achieve concentration plots that were comparable to the actual concentration plots, with a loss value of TBD. The associated diffusion constant D where found by the PINN, but here the results varied somewhat from the theoretically correct solution.

# Bibliography


Association, Alzheimer's et al. (2018). '2018 Alzheimer's disease facts and figures'. In: *Alzheimer's & Dementia* 14.3, pp. 367–429.

Bischof, Christian et al. (1992). 'ADIFOR: Automatic differentiation in a source translator environment'. In: *Papers from the international symposium on Symbolic and algebraic computation*, pp. 294–302.

Cserr, Helen F and LH Ostrach (1974). 'Bulk flow of interstitial fluid after intracranial injection of blue dextran 2000'. In: *Experimental neurology* 45.1, pp. 50–60.

Efron, Bradley and Robert J Tibshirani (1994). *An introduction to the bootstrap*. CRC press.

Franco Fernandez, Rosalia (June 2021). *A glymphatic system: controversial but conceivable*. URL: https://vusci.blog/2021/06/18/a-glymphatic-system-controversial-but-conceivable/.

Géron, Aurélien (2017). 'Hands-on machine learning with scikit-learn and tensorflow: Concepts'. In: *Tools, and Techniques to build intelligent systems*.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep learning*. MIT press.

Iliff, Jeffrey J et al. (2012). 'A paravascular pathway facilitates CSF flow through the brain parenchyma and the clearance of interstitial solutes, including amyloid $\beta$'. In: *Science translational medicine* 4.147, 147ra111–147ra111.

Kaur, Jasleen et al. (2021). 'Waste clearance in the Brain'. In: *Frontiers in Neuroanatomy* 15, p. 665803.

Ketkar, Nikhil and Eder Santana (2017). *Deep learning with Python*. Vol. 1. Springer.

Kingma, Diederik P and Jimmy Ba (2014). 'Adam: A method for stochastic optimization'. In: *arXiv preprint arXiv:1412.6980*.

Lu, Jie and George Em Karniadakis (2020). 'Physics-informed neural networks for inverse problems: A review'. In: *Annual Review of Fluid Mechanics* 52, pp. 173–199. DOI: 10.1146/annurev-fluid-011519-050623.

Magdoom, Kulam Najmudeen et al. (2019). 'MRI of whole rat brain perivascular network reveals role for ventricles in brain waste clearance'. In: *Scientific reports* 9.1, pp. 1–11.

Paszke, Adam et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. https://pytorch.org/.

Patey, Stig and Jouval Somer (Nov. 2022). *Classification and Regression, from linear and logistic regression to neural networks*. URL: https://github.com/JouvalSomer.

Raissi, Maziar, Paris Perdikaris and George E Karniadakis (2019). 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations'. In: *Journal of Computational physics* 378, pp. 686–707.

Shen, Mark D (2018). 'Cerebrospinal fluid and the early brain development of autism'. In: *Journal of neurodevelopmental disorders* 10.1, pp. 1–10.

Spires-Jones, Tara L and Bradley T Hyman (2014). 'The intersection of amyloid beta and tau at synapses in Alzheimer's disease'. In: *Neuron* 82.4, pp. 756–771.

Stein, Sherman C, Mark G Burnett and Seema S Sonnad (2006). 'Shunts in normal-pressure hydrocephalus: do we place too many or too few?' In: *Journal of neurosurgery* 105.6, pp. 815–822.

Sukumar, N and Ankit Srivastava (2022). 'Exact imposition of boundary conditions with distance functions in physics-informed deep neural networks'. In: *Computer Methods in Applied Mechanics and Engineering* 389, p. 114333.

Wright, Stephen, Jorge Nocedal et al. (1999). 'Numerical optimization'. In: *Springer Science* 35.67-68, p. 7.

Zapf, Bastian et al. (2022). 'Investigating molecular transport in the human brain from MRI with physics-informed neural networks'. In: *arXiv preprint arXiv:2205.02592*.

# Appendix

## A Github Repository

The code and associated test runs for this project is available at: The source files, test results and other files can be found at the GitHub: https://github.com/JouvalSomer/FYS-STK3155