Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Solving the Decay Equation using a Physics Informed Neural Network (PINN)

Semester Project

Department of Physics

ETH Zurich

Guglielmo Pacifico

supervised by

Dr. A. Adelmann

in collaboration with

Arnau Albà
Dr. R. Boiger

March 9, 2023

Abstract

In this thesis, we are going to consider the decay equation, a 1^{st} order linear ordinary differential equation (ODE) which in the general case does not have an analytical solution. In particular, we will focus on a linear nuclear chain reaction with three isotopes. To solve this equation we will use a Physics Informed Neural Network (PINN) which is a feed-forward neural network that embeds the physical law in its learning process.

The results of this thesis are a Python class that implements this PINN and the analysis of its performances, comparing different activation functions, optimizers, number of neurons and layers, number of time steps and weights, and trying to solve the decay equation in stiff cases.

The conclusion from this thesis is that by using Tanh as the activation function, LBFGS as the optimizer, using a single hidden layer and deciding the number of time steps, weights of the loss function and neurons accordingly to the problem considered, we can solve a linear nuclear chain of three isotopes with stiffness up to $||A|| \approx 500$.

Contents

1 Introduction

1.1 Motivations

The decay equation (1) is the mathematical model describing the populations and the activities of isotopes as a function of time:

$$\frac{d\mathbf{N}(t)}{dt} = A\mathbf{N}(t)\,, \qquad \text{with} \quad \mathbf{N}(t=0) = \mathbf{N}_0\,, \tag{1}$$

where $\mathbf{N}(t) \in \mathbb{R}^n$ is a vector whose components are the populations of the n isotopes at the time t and $A \in \mathbb{R}^{n\times n}$ is a matrix whose elements are the decay and transmutation constants [1].
It is in our interest to solve this equation to predict what the populations of each isotope will be at a given time. For instance, this is something we are interested in when we want to store spent fuel [2]. To do it safely we need to know the precise abundance of each element so that we can calculate their activities and avoid overheating and unexpected new chain reactions.

The problem with this equation is that in the general case, there is no analytical solution and we need to use some approximation method. Common methods to solve equation (1) numerically are:

- CRAM [3],
- Padé [3],
- Runge-Kutta [4].

However, they all have some limitations.
CRAM is the current state-of-the-art method and can solve any problem with almost no error (in the order of 10^{-8} [3]) as long as the eigenvalues of the A matrix are all on the real negative axis. But, the method becomes less reliable the more the eigenvalues have also a complex part.
Padé can solve problems, with both real and complex eigenvalues, however, starts to fail when the stiffness of the A matrix becomes significant.
Lastly, *Runge-Kutta* can make correct predictions both in the case of real or complex eigenvalues and also for stiff matrices as long as the time steps are reduced enough. However, increasing the number of time steps makes the method extremely slow and inefficient.

To avoid these limitations our approach will be to solve the decay equation using a *Physics Informed Neural Network* (from now on PINN) [5]. The idea is that the network will learn how to solve the problem directly from the equation, so having real or imaginary numbers will not be a problem.

1.2 The Decay Equation

The decay equation (1) gives us as a function of time the populations and activities of different isotopes that decay over time. This is a 1^{st} order differential equation (ODE), where we can read the population of each isotope at every time t from the vector $\mathbf{N}(t)$. This vector has n components, where each one is the population at the time t of one of the isotopes. The activity of each element is given by the $n \times n$ matrix A. Its elements are real but the eigenvalues can be both real or imaginary. When the eigenvalues are real they indicate the rate of spontaneous nuclear decay.

In the general case, there is no analytical solution for this equation. However, to be able to judge the quality of our PINN, we will consider a special case where the solution is known. We will from now on consider the case where we have a linear nuclear chain and the eigenvalues are

all real, that is a chain where each element can decay into only one other element and there are no feedback loops (neutron captures):

$$\mathcal{A} \xrightarrow{\lambda_1} \mathcal{B} \xrightarrow{\lambda_2} \mathcal{C}\,, \tag{2}$$

where $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ are three isotopes and λ_1, λ_2 are the decay rates of $\mathcal{A}$ and $\mathcal{B}$ respectively.
With this assumption, our A matrix will be diagonalizable, lower triangular and have real eigenvalues.
We can formulate problem (1) in mathematical terms as follows:

$$\begin{aligned}
&\frac{dN_1}{dt} = -\lambda_1 N_1\,, \\
&\frac{dN_i}{dt} = \lambda_{i-1} N_{i-1} - \lambda_i N_i\,, \qquad \text{with} \quad N_i = N_{i_0} \text{ for } i = 2, 3, ..., n\,.
\end{aligned}$$

Assuming zero concentrations of all daughter isotopes at time zero:

$$N_1(0) \neq 0 \quad \text{and} \quad N_i(0) = 0\,, \quad \forall i > 1\,,$$

we have the following solution found by Bateman in 1910 [1]:

$$N_n(t) = \frac{N_1(0)}{\lambda_n} \sum_{i=1}^{n} \lambda_i \alpha_i e^{-\lambda_i t}\,,$$

where

$$\alpha_i = \prod_{\substack{j=i \\ j \neq i}}^{n} \frac{\lambda_j}{(\lambda_j - \lambda_i)}\,.$$

To simplify our analysis even more we will consider the decay equation in the case of three isotopes, so our decay chain will be of the type shown in equation (2), implying the following A matrix:

$$A = \begin{pmatrix} -\lambda_1 & 0 & 0 \\ \lambda_1 & -\lambda_2 & 0 \\ 0 & \lambda_2 & 0 \end{pmatrix}\,. \tag{3}$$

In this special case, equation (1) can be formulated as:

$$\begin{cases} \frac{dN_1(t)}{dt} = -\lambda_1 N_1(t) \\ \frac{dN_2(t)}{dt} = \lambda_1 N_1(t) - \lambda_2 N_2(t) \\ \frac{dN_3(t)}{dt} = \lambda_2 N_2(t)\,, \end{cases}$$

having the following solution:

$$\begin{cases} N_1(t) = N_{1_0} e^{-\lambda_1 t} \\ N_2(t) = N_{1_0} \frac{\lambda_1}{\lambda_2 - \lambda_1} (e^{-\lambda_1 t} - e^{-\lambda_2 t}) \\ N_3(t) = N_{1_0} (\frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} - \frac{\lambda_2}{\lambda_2 - \lambda_1} e^{-\lambda_1 t} + 1)\,. \end{cases} \tag{4}$$

A graphical representation of the solution can be seen in figure 1.

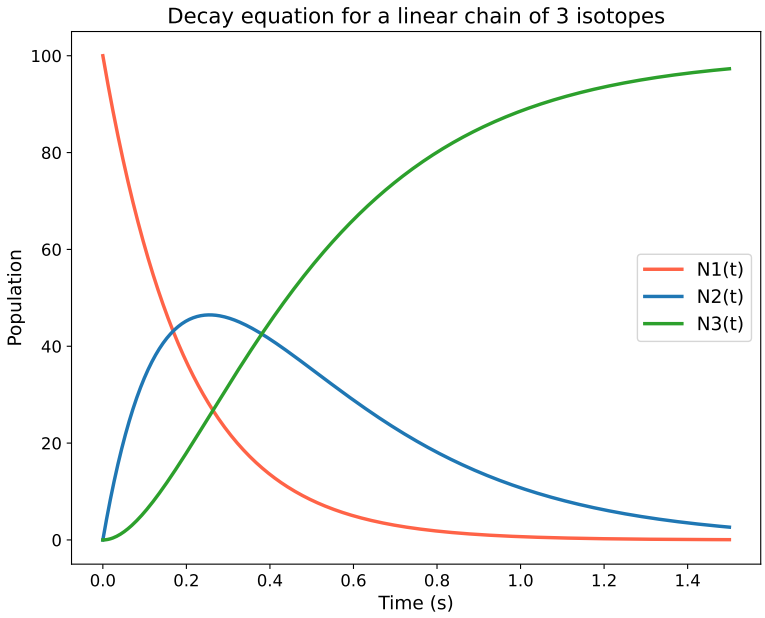


Figure 1: Populations over time of three isotopes governed by equation (4).

1.3 Neural Network

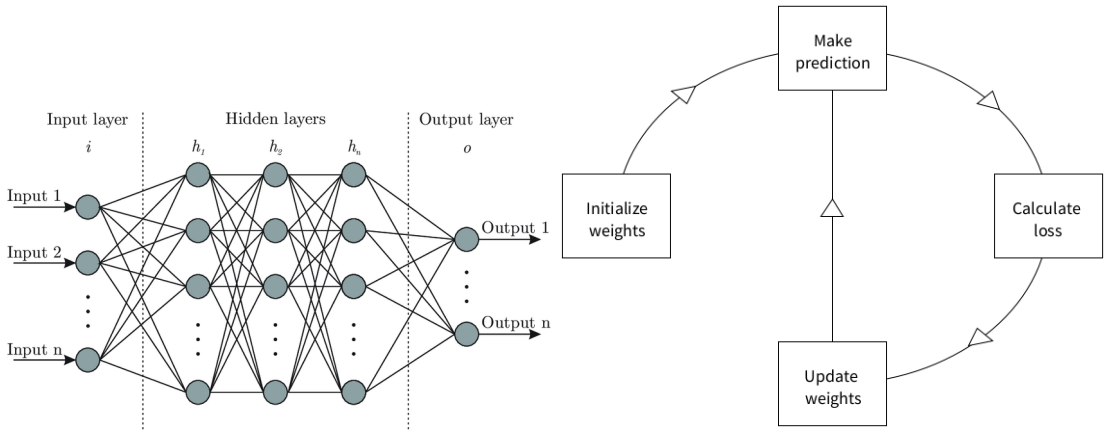
A neural network (NN) often referred to as an artificial neural network (ANN) is the core part of deep learning algorithms [6]. ANNs are comprised of nodes (*neurons*) and *layers*, see figure 2a. There are three types of layers, the *input layer*, the *output* and one or more *hidden layers*. The neurons are what form each layer. They are interconnected and each connection has an associated weight. Each neuron is a linear function composed with a fix nonlinearities.

There exist many different types of NN e.g. Convolutional Neural Networks (CNNs), Recurrent Neural Networks (ResNets), etc. In the context of this thesis, we will consider a specific subset of NNs, the *feed-forward* neural network. Their characteristic is that the connections between the nodes do not form a cycle [6], so the information moves in one direction from the input layer through the hidden layer(s) to the output layer.
Mathematically speaking we can define a feed-forward network as done in [7], a function $\Phi : \mathbb{R}^n \to \mathbb{R}^m$. If $\mathbf{z}_0 \in \mathbb{R}^n$ is the input to the network then we define recursively

$$\mathbf{z}_{i+1} = \sigma(W_i \mathbf{z}_i + \mathbf{b}_i)\,, \quad \text{where } 0 \leq i < L\,, \tag{5}$$

and the output $\mathbf{z}_{L+1} = W_L \mathbf{z}_L + \mathbf{b}_L \in \mathbb{R}^m$, being L the number of layers in the NN. Importantly, W_i are called the *weights* and are matrices, $\mathbf{b}_i$ are called the *biases* and are vectors, and σ is the *activation function* acting on each component individually. Choosing σ as a non-linear function allows us to have non-linearity. As σ once is decided is fixed the only free parameters are the weights W_i and the biases b_i, so we will have to choose these parameters correctly s.t. Φ is the closest possible to the function we want to fit.

The process to find the best parameter is known as *training* and can be schematised as shown in figure 2b. What this figure shows is an iterative process where each time we perform a loop we



(a) Structure of a feed-forward NN (figure from [8]). (b) Training of one epoch (figure form [7]).

Figure 2

say we have done an *epoch*. The training process can be easily understood following the scheme 2b. After having initialised the weights, this is done to avoid uncontrolled randomness in the code and be able to replicate the results, we proceed with making predictions. In this step, we pass the input value through the NN with the weights just assigned and store the output. With the output just computed we calculate the loss. This is a non-negative real function that (in the case of supervised learning) tells how much our prediction is distant from the real value. Our goal is to make this loss the smallest as possible so we update the weights s.t. the new prediction will reduce the loss function. To update the weights there are different methods, one is the gradient descent. What is done in this case is to compute the gradient of the loss function w.r.t the weights and update the weights moving in the opposite direction of the gradient (we are moving along $-\nabla$ to minimise the loss).
We repeat many epochs, so update the weights until one of the conditions is met. Either the loss is zero and so Φ is perfectly fitting our problem (this is rarely the case as often data have noise), or the loss is smaller than a certain threshold.

Thanks to the Universal Approximation Theorem [9] we can claim the given enough parameter to the network, this will always converge to the solution.

1.4 PINNs

Physics-informed neural networks (PINNs) are a type of universal function approximators that can embed the knowledge of any physical laws that govern a given data set in the learning process. The advantage of this method is to overcome the low data availability (or even the absence of data) on the system.

In this thesis, we will use a PINN to solve an ODE without providing any training data. The capability of PINNs in solving this task has been shown in this paper [7].
The goal of our network will be to solve equation (1), which means to predict the populations $\mathbf{N}(t)$ $\forall t \in [0, T]$. To fully understand how our PINN works, the differences and the advantage compared to classical NNs, let us first consider the case where we assume to know the analytical solution to equation (1). In this case, we can do supervised NNs where we compare the predictions of the NNs with the analytical solution. Naming Φ our network, this will be a function $\Phi : \mathbb{R} \to \mathbb{R}^n$ where the input will be the time $t \in \mathbb{R}$ and the output $\Phi(t) \in \mathbb{R}^n$ will be the populations

of the n isotopes we are considering at that time t. Assuming $\mathbf{M}$ to be the exact solution to equation (1): $\mathbf{N}(t_i) = \mathbf{M_i}$ and dividing t into *t_steps* time steps, we would have the following loss function:

$$\mathcal{L} = \| \sum_{i=1}^{t\_steps} \Big(\Phi(t_i) - \mathbf{M}_i \Big) \|_2^2 \ . \tag{6}$$

However, as explained in chapter 1.2, in the general case we do not have the analytical solution. With this in mind, we now move to the actual problem. The core idea of the method used is to make the network solve itself equation (1). To do this let's first rewrite the equation as:

$$\frac{d\mathbf{N}(t)}{dt} - A\mathbf{N}(t) = 0 \, . \tag{7}$$

Since we can compute the derivative of the network's output $\Phi(t)$, namely $\dot{\Phi}$, we now substitute $\mathbf{N}(t)$ in equation (7) with $\Phi(t)$. In this way, we are requiring that what the PINN is computing is the solution to our equation. As we want this to be equal to zero we can use this as our loss function:

$$\mathcal{L} = \| \sum_{i=1} \Big(\dot{\Phi}(t_i) - A\Phi(t_i) \Big) \|_2^2 \ . \tag{8}$$

In a nutshell, what we are doing is evaluating the network at a given time t and asking that the outcome fulfils the decay equation.

2 Implementation

2.1 Solving the Problem Analytically

As explained at the end of chapter 1.2 we are considering the decay of three isotopes, where the A matrix is of the form (3). In this case, we have an analytical solution for our problem and this solution is given by the set of equations (4).

Following the idea introduced in chapter 1.4, we start to solve our problem by writing a supervised NN. We do this to better understand the problem that we want to solve, but also because training a supervised NN will be a benchmark for us to understand if it is possible to solve this problem using a neural network and if so if the architecture that we are using could be sufficient to learn the problem.

The core part of this NN is the loss function. To train the network we decide on a final time t and divide it into *t_steps* steps. These will be inputs of the network and we will evaluate the NN at each of them. To optimize the parameters of the network we will use L-BFGS, a quasi-Newton method, minimizing the loss at all the time steps with the respective analytical solution computed using equation (4) at all the time steps.

In figure 3 is reported a snipped of the loss function. Relevant to note is that ODE_{loss} is obtain as:

$$ODE_{loss} = N_{analytical} - N_t$$

where N_t is the output on the NN given t as input and $N_{analytical}$ is the analytical solution evaluated at time t.

2.2 Solving the Problem with the PINN

We now move on to explain how the PINN has been implemented.
Our goal is to have a network Φ that takes as input a real value, the time t, and returns as the

```python
def lossFunction_Supervised(self, t, weights=[1,1], batch_size=None):
    """
    This method is used if we want to do supervised learning.
    This is only possible if the problem can be solved analytically.
    """
    #initial time in correct shape
    t0 = torch.tensor([0], dtype=torch.float, device=DEVICE).reshape(-1,1)
    #if bath_size is give a random sample of training point is used for each epoch
    if batch_size != None:
        n = t.shape[0]
        t = t[np.random.choice(n,int(batch_size*n),replace=False)]
    #exact and predicted initial values
    N_exact_0 = self.initial
    N_0 = self(t0)
    #predict values of the network at training times
    N_t = self(t)
    #compute the analytical solution
    N_analytical = get_AnalyticalSolution(self.lamda.cpu().detach().numpy(), self.initial.cpu().detach().numpy(), t.cpu().detach().numpy())
    N_analytical = torch.tensor(np.array(N_analytical), dtype=torch.float, device=DEVICE).reshape(self.initial.shape[0], -1)    #make N_alaytical a tenosor with shape [2, t_steps]
    N_analytical = torch.transpose(N_analytical,0,1)    #to correctly do operation with N_pred, traspose N_alaytical to [t_steps, 2]
    #Initial conditions loss
    IC_loss = ( N_exact_0 - N_0 ).square().mean()
    #ODE loss
    ODE_loss = ( N_analytical - N_t).square().mean()

    return (weights[0]*IC_loss + weights[1]*ODE_loss)/sum(weights)
```

Figure 3: Snippet of the function *lossFunction_Supervised* of the class *BatemanPINN*.

outputs n real values, the populations of the n isotopes at that time t,

$$\Phi : t \to \Phi(t) = \mathbf{N}(t)\,.$$

Furthermore, we want that the output $\Phi(t)$ satisfies the decay equation (1).

To understand how such a network works and is trained we need to focus our attention on the loss function $\mathcal{L}$. The loss function we have decided to use consists of two terms, each of which is a mean squared error. The first term, MSE_{ODE}, regards the decay equation and requires that the output is a solution of the ODE. The second term, MSE_{IC}, makes sure the initial conditions are satisfied to make the solution unique.

$$MSE_{ODE} = \frac{1}{t\_steps} \sum_{i=1}^{t\_steps} \parallel \dot{\Phi}(t_i) - A\Phi(t_i) \parallel_2^2 ,$$
$$MSE_{IC} = \parallel \Phi(0) - \mathbf{N_0} \parallel_2^2 ,$$

where t_steps is the number of time steps we divide t into and $\mathbf{N_0}$ are the populations of the n isotopes at time $t = 0$.
The loss is then a weighted sum of the four individual losses:

$$\mathcal{L} = \frac{w_1 MSE_1 + w_2 MSE_2}{w_1 + w_2}\,. \tag{9}$$

Notice that the parameters that maximise and minimise the loss are invariant under scaling of the loss and so for the transformation: $(w_1, w_2) \to (\lambda w_1, \lambda w_2)$. This allows us to set one of the w_i to 1 without loss of generality.
We have decided to not add other terms to the loss as adding more terms would have implied more weights to tune.

Let's now focus on MSE_{ODE} and let's analyse how the PINN can solve the ODE. The first step is to divide the time t in a vector of t_steps component: $\mathbf{t} = (t_0, t_1, ..., t_{t\_steps})$. The result will be $\Phi(\mathbf{t})$ which will be an $n \times t\_steps$ matrix where each column will be the population of one isotope evaluated at all the $\mathbf{t}$ times. We now divide this matrix in n vectors $\mathbf{N}_i$ where each one has the population at all the times of one isotope. We proceed now to differentiate these vectors w.r.t. the input $\mathbf{t}$ using PyTorch's built-in automatic differentiation method **torch.autograd.grad**. This will give us the $\dot{\Phi}(\mathbf{t})$ obtaining the RHS of the equation (1). To get the LHS we just have

```python
def lossFunction(self, t, weights=[1,1], batch_size=None):
    """
    This method is used to train the PINN.
    This is can be used for any problem as the PINN will learn the solution to the equation its self
    """
    #initial time in correct shape
    t0 = torch.tensor([0], dtype=torch.float, device=DEVICE).reshape(-1,1)
    #if bath_size is give a random sample of training point is used for each epoch
    if batch_size != None:
        n = t.shape[0]
        # t_np = np.random.rand(int(batch_size*n))*t.cpu().detach().numpy()[-1]
        # t = torch.tensor(t_np, requires_grad=True, dtype=torch.float, device=DEVICE).reshape(-1,1)
        t = t[np.random.choice(n,int(batch_size*n),replace=False)]
    #exact and predicted initial values
    N_exact_0 = self.initial
    N_0 = self(t0)
    #predict values of the network at training times
    N_t = self(t)

    #Initial conditions loss
    IC_loss = ( N_exact_0 - N_0 ).square().mean()

    #ODE loss
    #here we solve the 'right hand side' of the dacay equation, doing first the mat. multp. then deriving the 'left hand side' and finally equaling them
    ODE_loss = 0    #initialize the counter
    rhs = torch.matmul(self.lamda_matrix, torch.transpose(N_t, 0, 1))
    for i in range(len(self.initial)):
        Ni_t = N_t[:,i]
        Ni_t_dot = torch.autograd.grad(Ni_t, t, retain_graph=True, create_graph=True, grad_outputs=torch.ones_like(Ni_t))[0].transpose(0,1)
        ODE_loss += ( Ni_t_dot - rhs[i,:] ).square().mean()
    # del t
    return (weights[0]*IC_loss + weights[1]*ODE_loss)/sum(weights)
```

Figure 4: Snippet of the function *lossFunction* of the class *BatemanPINN*.

to do the matrix multiplication between A, that is an $n \times n$ matrix and $\Phi(\mathbf{t})$, which will result in an $n \times t\_steps$ matrix. Finally, subtract the LHS from the RHS and to minimize it we take the mean square of each component of the resulting matrix.

In figure 4 it can be seen how the loss function has been implemented. Relevant to note is the difference from the Analytical approach. Here to compute ODE_{loss} we do not use the analytical solution but invert the equation (1).

3 Analysis

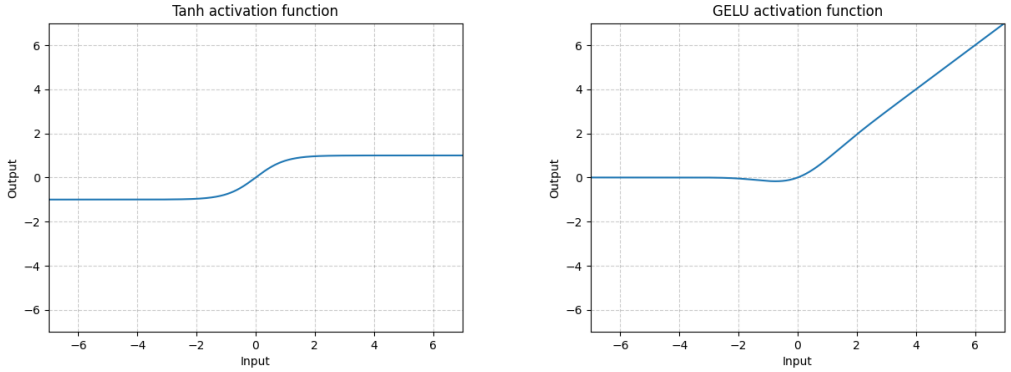
The goal of this thesis is to prove that it is possible to solve a low dimensional problem of nuclear decay using a PINN.
To achieve this, we proceed step by step in finding the best architecture for the network.

In our analysis, we will first investigate which activation function and optimizer are to be considered the optimal choice. Secondly, we will focus on the problem linked with stiff matrices.

In all the analysis that we will do, we will repeat each simulation 10 times, each one with a different seed randomly generated. This will allow us to repeat each simulation starting with a different random network to avoid the chance of getting stuck in a local minimum due to unlucky initial conditions of the network and will also allow us to do statistics on the results obtained.

3.1 The Activation Function

The activation function of a node defines the output of that node's given input, this is what allows us to have non-linearity in our model. In this thesis, we need to have an activation function that is differentiable everywhere as we need to compute $\dot{\Phi}$ for our loss function. For this reason, our analysis will be focused on two activation functions, *Tanh* and *GeLu*.



(a) Tanh activation function, given by eq. (10) (b) GeLu activation function, given by eq. (11)

Figure 5

Tanh:

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{10}$$

As proven in this paper [5], this activation function gives good results when used in PINNs to solve ODE. However, this function can become problematic. As we can see from figure 5a, when the input of the node is far from zero, the derivative of the activation function goes to zero, this is known as the vanishing gradient and causes the network to get suck in the learning process. To avoid this problem we consider also another function.

GeLu:

$$\text{GeLu}(x) = x\phi(x) \tag{11}$$

where $\phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution [10]. This function solves the problem of the vanishing gradient (see figure 5b).

To compare the two activation function we analyse the loss, defined as the absolute value of the difference between the analytical solution $\mathbf{M}(t)$ and the output of the PINN $\Phi(t)$,

$$loss = \left| \sum_{i=1}^{t\_steps} \Big(\mathbf{M}(t_i) - \Phi(t_i) \Big) \right| .$$

The case that we consider is a simple one, a two isotope decay with initial condition $\mathbf{N_0} = (100, 0)$ and the A matrix being $A = \left(\begin{smallmatrix} -5 & 0 \\ 5 & -10 \end{smallmatrix} \right)$.
To solve this case we can use PINNs with a simple architecture. The weights of the loss function, equation (9), are $\mathbf{w} = (w_1, w_2) = (1, 1)$. Each layer of the NNs has 50 neurons and we compared six PINNs where each one has a different number of hidden layers, from 1 up to 6. We then considered the population of the two isotopes after a time $t = 5\,s$ that we divide in $t\_steps = 250$.

The results obtained are shown in figure 6.
As we can see over the six configurations the results are similar, showing that there are no real benefits of using more hidden layers. Moreover, we can conclude that both activation functions reach almost zero loss, but Tanh is faster. For this reason, we will continue our analysis using Tanh.

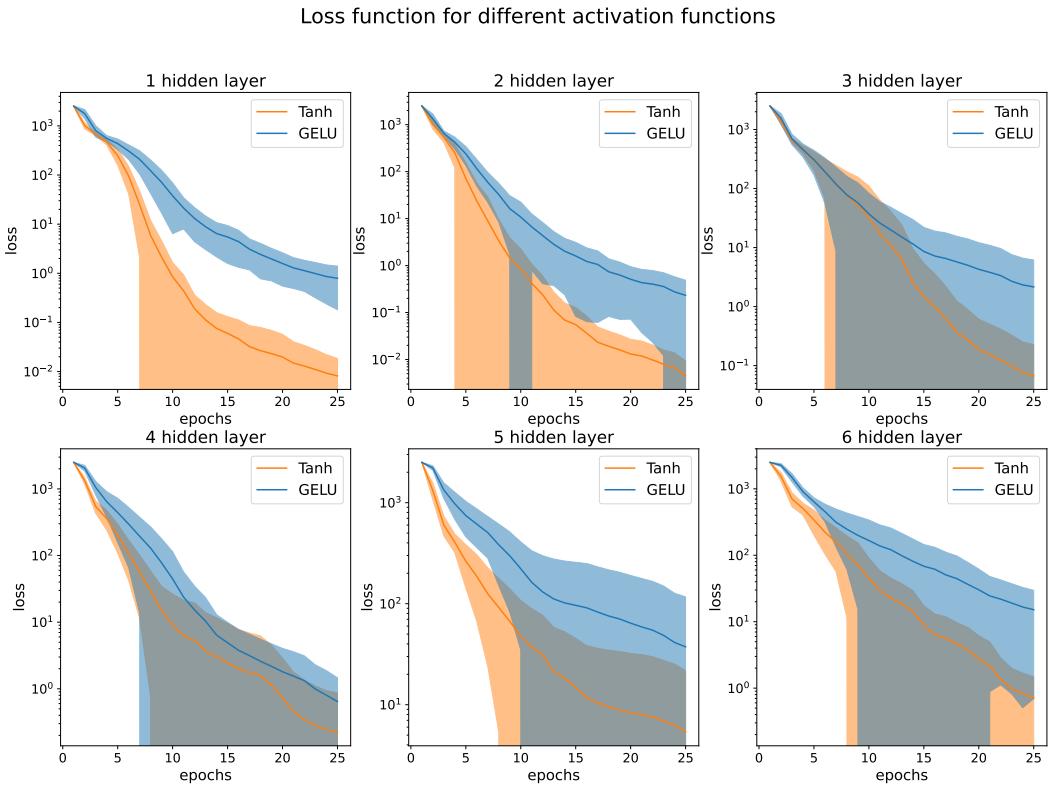


Figure 6: Six plots of the six architectures of the PINN, each one with a different number of hidden layers used. What is shown is the average of 10 runs with the shade being the std. On the $x-$axis we have the epochs of the training and on the $y-$axis the respective loss.

3.2 The Optimizer

Optimizers are algorithms or methods used to train parameters of a neural network such as weights and biases to reduce the losses. In this thesis, we are going to consider two optimizers, *Adam* and *LBFGS*.

Adam (Adaptive Moment Estimation) [11] is a gradient descent with momentum. The reason we consider this optimizer is that is known to be extremely good if one wants to train the neural network in less time and more efficiently.
Important, when using this method is the learning rate, often referred to as η. In our analysis, we are going to consider $\eta = 0.001$

LBFGS is a method based on the idea of Newton's root-finding method. To find a minimum of $\mathcal{L}$ we seek a zero of $\nabla\mathcal{L}$. With the Hessian matrix $H_{\mathcal{L}}$, or an approximation thereof, we define iteratively [12]

$$\theta_{n+1} = \theta_n - \frac{\nabla\mathcal{L}(\theta_n)}{H_{\mathcal{L}}(\theta_n)}\,.$$

This optimizer has been used in this paper [7] and has been proven to give good results when used for PINNs [5].
Furthermore, this method requires the specification of a learning rate as for Adam, however, after

several attempts we figured out that it is better to use an adaptive learning method, adjusting η at each step. This can be done using the built-in function in PyTorch strong_wolfe [13] as shown here:

```
LBFGS = optim.LBFGS(PINN_LBFGS.parameters(),
                    lr=0.1,
                    line_search_fn='strong_wolfe'
                    )
```

As done for the comparison of the two activation functions, we will analyse the loss when using the two optimizers. The setting of the PINN is the same as used in the previous analysis, 50 neurons per layer and we will compare the six cases as before, starting with 1 hidden layer and going to the case with 6 hidden layers. Initial condition $\mathbf{N_0} = (100, 0)$, weights of the loss $\mathbf{w} = (w_1, w_2) = (1, 1)$ and $A = \begin{pmatrix} -5 & 0 \\ 5 & -10 \end{pmatrix}$. Finally, as concluded at the end of chapter 3.1 the activation function that we are using is Tanh.

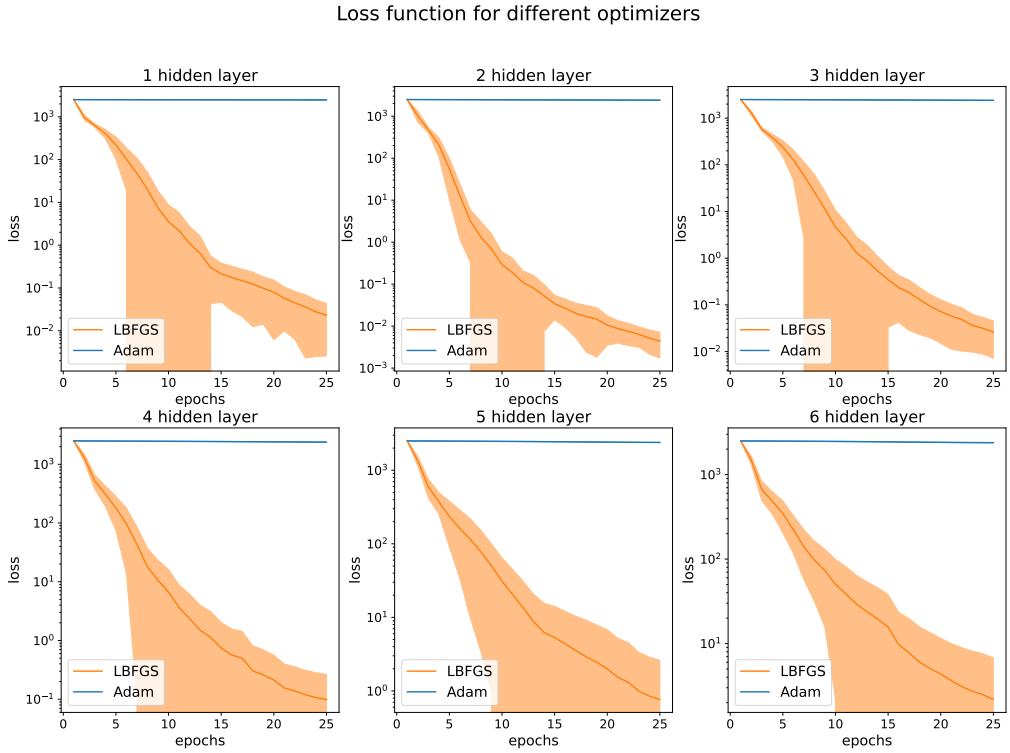


Figure 7: Six plots of the six architectures of the PINN, each one with a different number of hidden layers used. What is shown is the average of 10 runs with the shade being the std. On the $x-$axis we have the epochs of the training and on the $y-$axis the respective loss.

The results obtained are shown in figure 7.
For this analysis, we can conclude that the number of layers is irrelevant to the performance but we can clearly see how LBFGS outperforms Adam. For this reason from now on in the next analysis, we will consider PINNs with 1 single hidden layer that uses Tanh as the activation function and LBFGS as the optimizer.

3.3 Stiffness

Now we are going to analyse the performance of our PINN when dealing with a stiffer problem. First of all, we need to define what stiffness is. The *stiffness* $||A||$, of a matrix A, is the absolute value of the ratio between the real part of the maximum and the minimum eigenvalue:

$$||A|| = \frac{|\operatorname{Re}(\lambda_{max})|}{|\operatorname{Re}(\lambda_{min})|} \quad \text{where} \quad |\operatorname{Re}(\lambda_{max})| \geq |\operatorname{Re}(\lambda_i)| \geq |\operatorname{Re}(\lambda_{min})|\,,$$

with λ_i being the eigenvalues of A.
Then it is important to keep in mind why we want to be able to solve stiff problems. The reason is that in real problems the A matrix will be extremely stiff, approximately $||A|| \approx 10^{20}$ [2]). The problem that derives from stiff problems is that the populations of the isotopes have very different lifetime scales so it is very complicated for the network to sample all of them correctly.

In the following analysis, we are going to consider a linear decay chain with three isotopes, the same as in equation (2), where the initial conditions are $\mathbf{N_0} = (100, 0, 0)$ and the A matrix will be of the type $A = \begin{pmatrix} -\lambda_1 & 0 & 0 \\ \lambda_1 & -\lambda_2 & 0 \\ 0 & \lambda_2 & 0 \end{pmatrix}$. Using this matrix we know that after enough time the final populations will be $\mathbf{N}(T) \approx (0, 0, 100)$. In this way, to evaluate the performance of the network we will compute the relative error on N_3, the population of the third isotope:

$$rel\_error = \frac{|N_3^{Analy}(T) - N_3^{PINN}(T)|}{N_3^{Analy}(T)}$$

The reason we are only considering the relative error on N_3 is that if we would consider also the relative error on N_1 and N_2 we would get errors, as dividing by $N_{2/3}^{Analy}(T)$ would be like dividing by zero.
In figure 8 it is shown how when we consider stiff problems the lifetimes of the isotopes are very different (compare this plot with figure 1). Moreover, this plot shows how already for $||A|| = 200$ we can not consider the relative error of all the isotopes as $N_1(t)$ goes to zero immediately.

We are going to investigate the performances of different stiffness in the range $||A|| \in [1, 1000]$. We will let the network train until the loss will be smaller than 10^{-5} for five epochs in a row. This is what is called *auto-stop*. As the number of epochs will change for every training we will also consider this to evaluate how long the training will be.

The analysis that we are going to do will be divided into three steps:

1. we will increase the number of t_steps,
2. we will change the weights in the loss function,
3. we will increase the number of neurons.

Increasing the number of time steps will make the training more computationally heavy but in return will allow us to divide the problem into more points to evaluate allowing the network to have more information on the populations.
Changing weights is necessary as soon as we go to non-trivial stiffness as the populations of the first two isotopes go rapidly to zero, so if we do not ask to give a big weight to the initial condition, the network will easily satisfy the loss function prediction zero everywhere since only in the first time_steps the values of N_1andN_2 are $\neq 0$.
Finally using more neurons will increase the model capacity. The number of neurons in a neural network determines its capacity to learn and represent more complicated functions. As going to the more stiff problem will imply that the populations' curves will be less smooth, we expect we

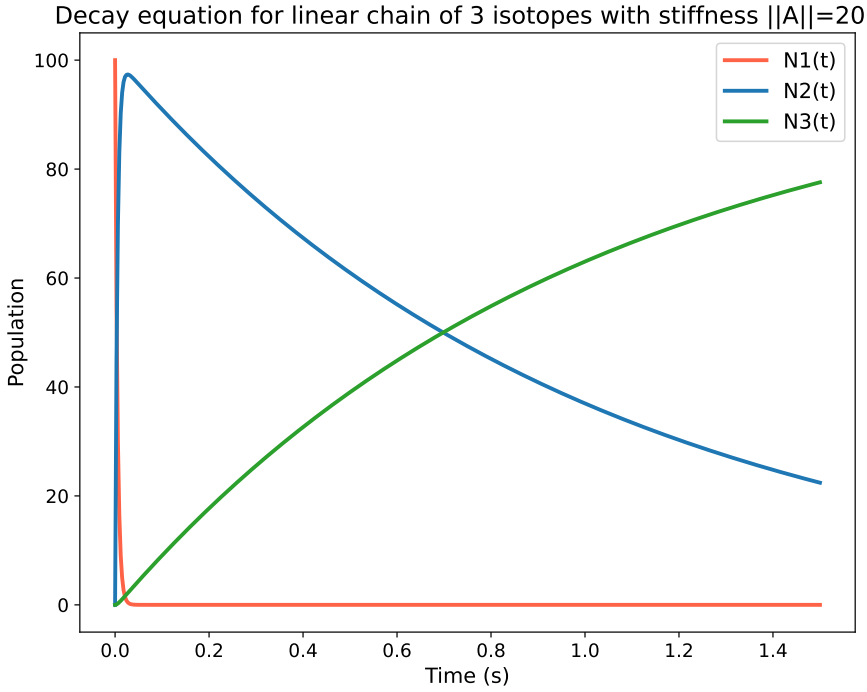


Figure 8: Population over time of three isotopes in the case of stiff A matrix.

will need more neurons to fit them.
With this in mind, we will analyse different configurations of these three parameters. The values that we are going to analyse will be:

- t_steps $\in \{250, 500, 1000, 10000, 100000\}$
- $\mathbf{w} = (w_1, w_2) \in \{[1, 1], [25, 1], [50, 1], [75, 1], [100, 1], [215, 1], [464, 1], [1000, 1]\}$
- neurons $\in \{50, 100, 200, 500\}$

In the following plots we are going to focus only on the most important result and we will comment on those.

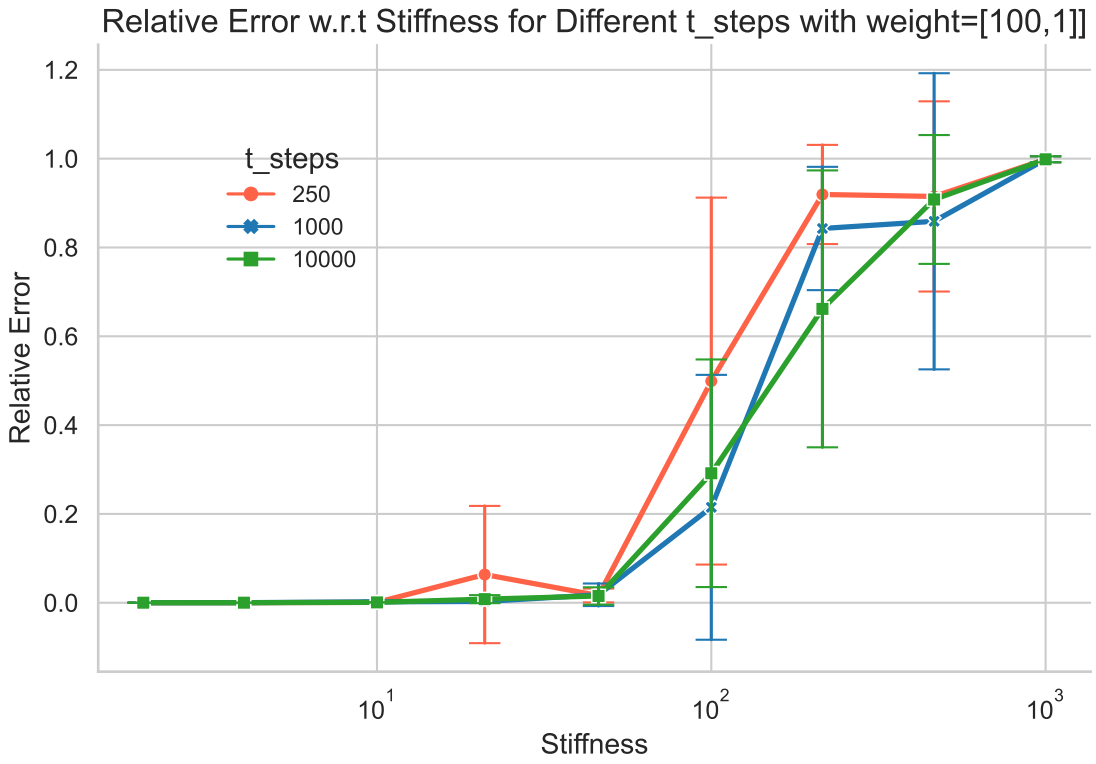


Figure 9: Relative error on N_3 for increasing values of stiffness. With three different colours are reported the plots for PINNs using an increasing number of *t_steps*.
Each of the points drawn is the average over 10 different runs and the error bars represent the std.

In figure 9 we can see that for small stiffness the number of t_steps is not important. The PINN is in any case able to correctly predict the final population of N_3. However, when we move to $||A|| \approx 100$, then the number of t_steps becomes relevant. In figure 9 is represented the analysis of a specific case, a PINN with loss weights $\mathbf{w} = (w_1, w_2) = (100, 1)$, with 50 neurons and the comparison of three t_steps is shown: $\{250, 1000, 10000\}$. In this scenario at $||A|| \approx 100$ when using 10000 t_steps allows us to have a relative error of 20%. Instead, when using 250 t_steps we would have a relative error of more than 50%. Finally, we can see that for problems with $||A|| \approx 1000$ we have almost 100% of relative error, implying that increasing only the time steps is not enough.

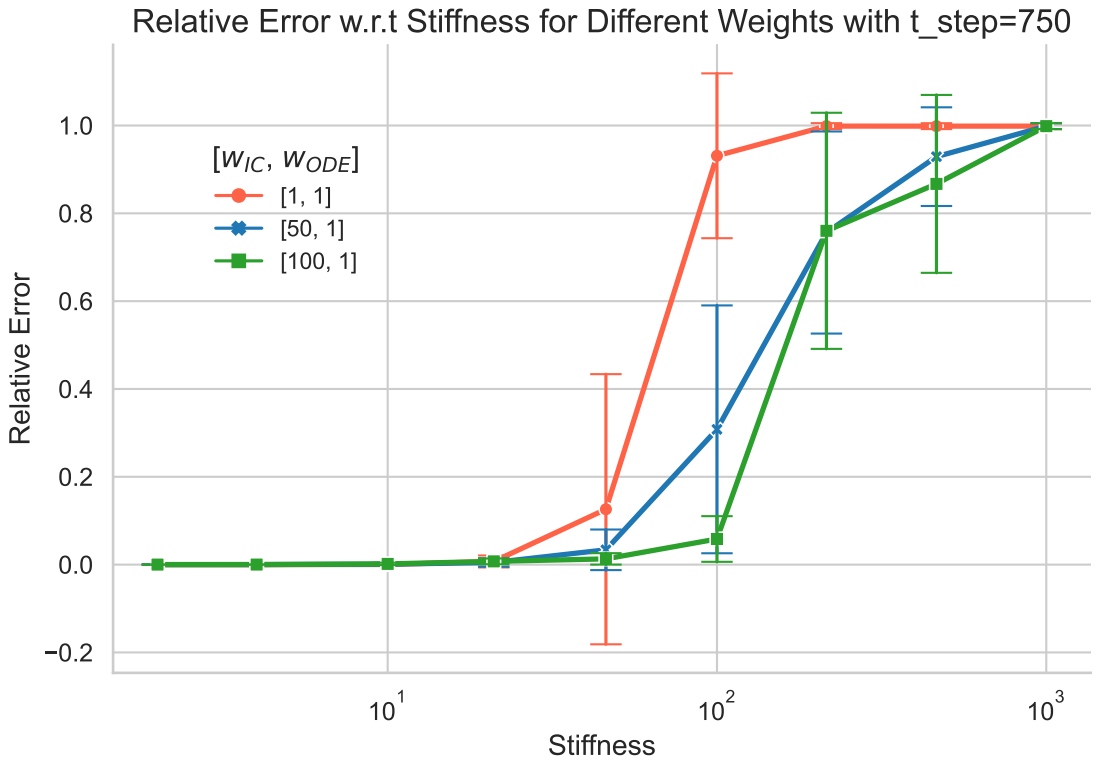


Figure 10: Relative error on N_3 for increasing values of stiffness. With three different colours are reported the plots for PINNs using different combinations of weights.
Each of the points drawn is the average over 10 different runs and the error bars represent the std.

In figure 10 a part of the results obtained when varying weights is reported. From this plot, we can immediately see how much the weights in the loss function are important. Already around $||A|| \approx 50$, increasing w_1 by $(1 \to 50)$ reduces the relative error from around 10% down to 3%. However, also for the weights as for the time steps, we can see that at $||A|| \approx 1000$, the error goes up to 100%

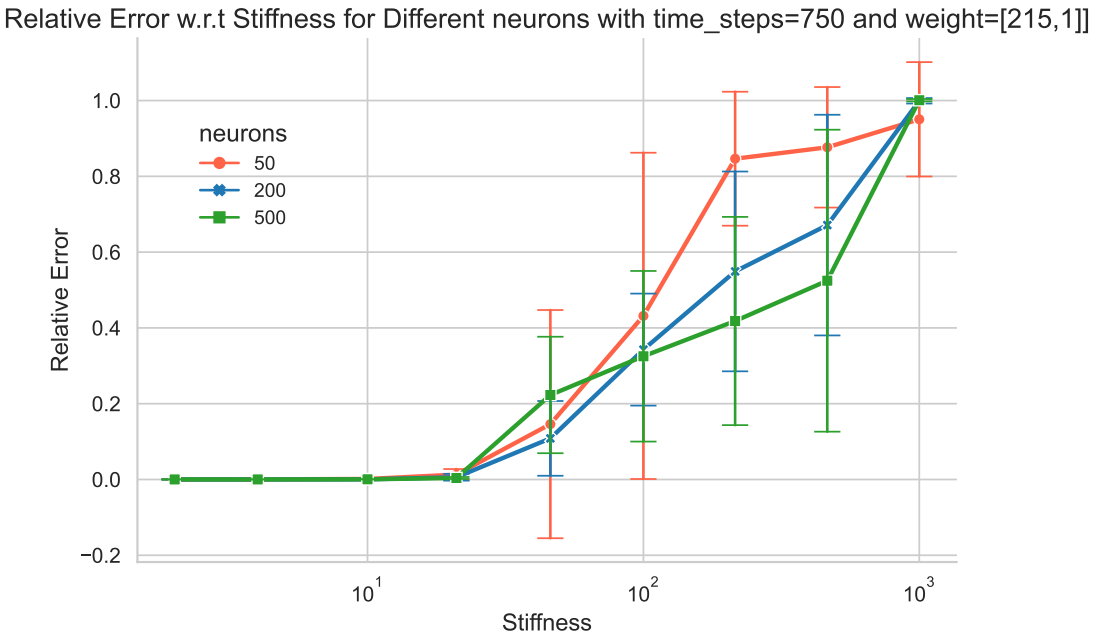


Figure 11: Relative error on N_3 for increasing values of stiffness. With three different colours are reported the plots for PINNs using an increasing number of neurons.
Each of the points drawn is the average over 10 different runs and the error bars represent the std.

Finally, in figure 11 the relative error is represented when using a different number of neurons. The plot is similar to the two described above. We have almost zero error for low stiffness, independent of the number of neurons used. A benefit when using more neurons for higher stiff problems but a total failure for $||A|| = 1000$ no matter the number of neurons.

From these analyses, we can conclude that all three parameters have an impact on the PINN's performance. So now we perform a hyperparameter search to find the optimal combination.

3.4 Hyperparameter Search

We now perform a hyperparameter search to find the optimal combination of the three parameters analysed in chapter 3.3. To do this we will do a grid search over all the possible combinations. In figure 12 we can see one instance of the grid search. At first look, it could seem that in any combination we are not able to solve the problem for a problem with $||A|| > 200$. However, what we are looking at in figure 12 are the averages over 10 runs each one using a different seed. We are not interested in the PINN giving the correct output for any seed, but in reality, we care that works for at least one. For this in figure 13 is shown the relative error of a single run. Here we can observe that for certain combinations we are able to solve correctly the problem.

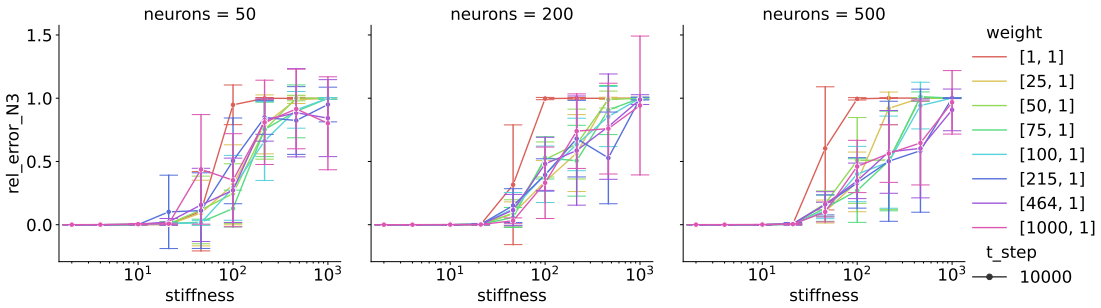


Figure 12: The plots report the relative error on N_3 for increasing values of stiffness. All the results shown are obtained using 10000 *t_steps*. The three different plots show the analysis for a different number of neurons used. The different colours indicate a different combination of weights used.
Each of the points drawn is the average over 10 different runs and the error bars represent the std.

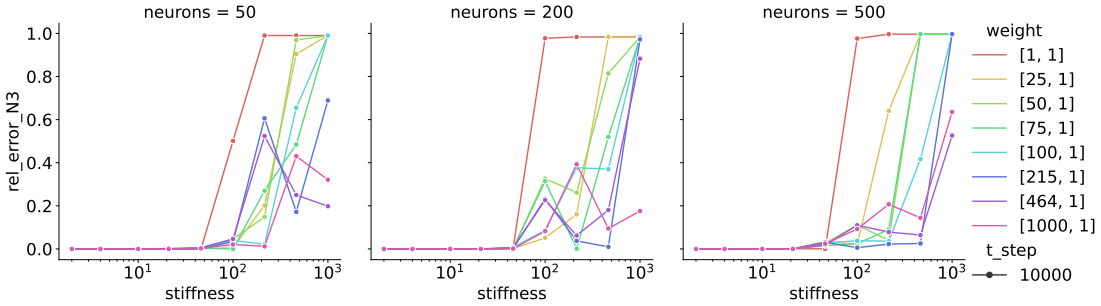


Figure 13: The plots report the relative error on N_3 for increasing values of stiffness. All the results shown are obtained using 10000 *t_steps*. The three different plots show the analysis for a different number of neurons used. The different colours indicate a different combination of weights used.
Each of the points drawn is the relative error of one of the 10 runs done, chosen as the best s.t. the relative error is the smallest for a specific combination of neurons and weights.

This proves that the class that we implemented is working and is able to solve the decay equation for a linear chain of three isotopes with stiffness up to $||A|| \approx 500$.

4 Conclusions and Future Work

In conclusion, we can claim that the PINN that we implemented is working and can solve the decay equation for a linear chain of three isotopes with stiffness $||A|| \approx 500$. The optimal configuration that we have found for it is to use LBFGS as the optimizer and Tanh as the activation function, using only one hidden layer with the number of neurons, time steps and weights of the loss function that we have to decide depending on the problem considered.

As future work that could be done is to try to increase the size of the A matrix and its stiffness, ideally arriving to solve a real problem where $A \in \mathbb{R}^{4000\times4000}$ and $||A|| \approx 10^{20}$.

Then one can work on the uncertainty quantification that is where using a PINN to solve the decay equation could bring some benefits in comparison to the other methods.
Moreover, one can compare the performances of this method with the other currently used in terms of computation time and accuracy.
Finally, one can work on transfer learning. This can be seen for two reasons. One is uncertainty quantification. Training a first network for A and then retraining it for perturbations ΔA of the A matrix. As ΔA, is a small perturbation, this new problem will have a loss function very similar to the first one and so the new training should be extremely fast. Second, we can use transfer learning in case solving a real problem is too complicated for the network. The idea is that we simplify the A matrix such that we have an analytical solution. With the solution, we train a supervised NN and then starting from the weights of this NN we train the PINN on the real problem. Doing so we would not start from a random point in the loss function risking getting stuck on a bad minimum, but we should start closer to the actual solution.

References

[1] Jerzy Cetnar *General solution of Bateman equations for nuclear transmutations.* Annals of Nuclear Energy (2013) https://www.sciencedirect.com/science/article/pii/S0306454906000284#:~:text=The%20Bateman%20equations%20for%20radioactive,decay%20constant%20of%20ith%20nuclide.

[2] ENSI *Geologische Tiefenlager.* ENSI DE, (2020) https://www.ensi.ch/de/dokumente/richtlinie-ensi-g03-deutsch/

[3] M. Pusa and J. Leppänen. *Computing the Matrix Exponential in Burnup Calculations.* Nuclear Science and Engineering: 164, 140–150, (2010) https://www.tandfonline.com/doi/abs/10.13182/NSE09-14

[4] J. M. Hykes and R. M. Ferrer. *Solving the Bateman equations in CASMO5 using implicit ode numerical methods for stiff systems.* Technical report, American Nuclear Society - ANS; La Grange Park (United States), (2013) https://www.osti.gov/biblio/22212817

[5] M. Raissi, P. Perdikaris, and G. E. Karniadakis. *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.* Journal of Computational Physics, 378:686–707, (2019) https://www.sciencedirect.com/science/article/pii/S0021999118307125

[6] Zell, Andreas. *Simulation Neuronaler Netze [Simulation of Neural Networks]* (1994) ISBN 3-89319-554-8

[7] Justin Mazenauer and Andreas Adelmann, *Kepler problems with physics-informed neural networks.* (2022)

[8] Website last visited 23 February 2023 https://www.kdnuggets.com/2019/11/designing-neural-networks.html

[9] Patrick Kidger and Terry Lyons, *Universal approximation with deep narrow networks.* (2019) https://arxiv.org/abs/1905.08539

[10] PyThorch documentation on GeLu https://pytorch.org/docs/stable/generated/torch.nn.GELU.html

[11] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization.* (2014) https://arxiv.org/abs/1412.6980

[12] Mykel J. Kochenderfer and Tim A. Wheeler, *Algorithms for Optimization. MIT Press, Came- bridge, Massachusetts.* (2019)

[13] PyThorch documentation on LBFGS https://pytorch.org/docs/stable/generated/torch.optim.LBFGS.html