FACULTY OF SCIENCE AND TECHNOLOGY

MASTER THESIS

Study programme / specialisation:

Applied Data Science

The spring semester, 2022

Open / Confidential

Author:
Lars Belbo Lukerstuen

Lars B. Lukerstuen
....
(signature author)

Course coordinator:

Supervisor(s):
Aksel Hiorth

Thesis title:
A Study on Physics Informed Neural Networks,
with Applications for Compartment Models

Credits (ECTS):
30

Keywords:

PiNN, Machine Learning,
Numerical Computation,
Physics Informed Learning

Pages: 77

+ appendix: 104

Stavanger, 13/06/2022
date/year

Abstract

The main goal of this thesis was to investigate the methodology of Physics Informed Neural Networks (PiNN), as a computational tool leveraging differential equations as a regularization for a learning task. PiNN is a new field of research and therefore particular concern was given to the task of obtaining an understanding of the method, gauging benefits, performance, and appropriateness in relation to established methods. In order to develop this knowledge, the methodology was implemented and applied through four case studies, three of which demonstrates achievements already supported by the literature. In addition case three incorporates a thorough testing scheme, scoping out PiNNs' capabilities of parameter discovery and regularization. From this a larger framework is developed. In case four, the framework is utilized applying of the method of PiNN in a real world biomedical context, realized as a model of the circulatory system. The implementations were realized in a bottom up approach utilizing the neural network capabilities of PyTorch. Overall, the findings of the thesis support the established findings of previous literature in regards to performance and capabilities. Additionally, important details in regards to implementation and solution validity is highlighted, addressing the conditions necessary for the optimal use of PiNN as a methodology.

Acknowledgements

My sincerest gratitude will be directed towards my thesis advisor Aksel Hiorth, for his support and guidance throughout the entire process of shaping this thesis. Not only did he provide ample feedback, advice, and instruction when needed, but also maintained and shared an enthusiasm which was found to be of great help in maintaining motivation, and shaping ideas for the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the information age develops, the field of deep learning has seen large success in it's endeavour of developing large scale flexible models, leveraging models consisting of millions of parameters, finding and capturing patterns in large sets of data. The last decade saw a large growth in interest and developments of the techniques and technology of machine learning, in part due to the computational potential which has become available. As the technology develops, it sees more widespread adaptions and potential use cases deeper within several fields of science and industry. However, there are still areas where the standard techniques of deep learning and machine learning falls short. A not-so unrelated field is the field of mathematics and scientific computing. Within this field there is the general problem of solving differential equations. As one would know, differential equations are equations which models the relationships between rates of change in properties of a system. Since the properties of physical systems governed by physical laws often incorporate properties subject to variation and change, differential equations has proven valuable in the field of physics and science in general. Despite the various technological and educational advances in recent years, there are still differential equations that remains unsolved. The canonical example of this is the Navier Stokes equations, a set of differential equations which describes the dynamical properties of a fluid. Not only is there no general solution to this set of equations, but if one was to prove that three always exists solution one can stand to gain a fair amount of money. The point to drive home is the following: There are various differential equations describing dynamics, which may even be vital to some real world operations, which are *difficult to solve*. There exists various mature and effective numeric techniques for solving, or approximating solutions to these problems. However, there are still challenges to overcome. Considering the recent successes of technologies developed in deep learning one may ask the question: Can a neural networks solve differential equations? Maziar Raissi, Paris Perdikaris, and G. E. Karniadakis, 2017 (M. Raissi, P. Perdikaris, and G. Karniadakis, 2019), answers yes to this question, developing and setting a foundation for the methodology known as PiNN, *Physics Informed Neural Networks*.

1.1 Problem formulation

PiNN, refers to the technique of combining Machine Learning with prior information in the from of differential equations. The underlining task of this thesis means to highlight this technique as a both a machine learning and numerical methodology, scoping out strengths and weaknesses, while attempting to build an understanding of the method and cases where it may excel.

Chapter 2

Background

One of the goals of the thesis is to explore the methodology of PiNN. Hence consideration will be given to the background and current state of relevant fields. The PiNN technique can be said to stem from Machine Learning, but also reaches into the realms of what traditionally has been considered numerical methods. This chapter is meant as a general overview, scoping out the current landscape, some of the relevant history of the method, and its alternatives.

2.1 State of the art

PiNN is the technique of applying neural networks to problems seeking solutions to partial and ordinary differential equations (PDE's/ODE's). Although the main idea behind the technique, utilizing neural network models to solve PDE's is not necessarily recent (Lagaris, Likas, and Fotiadis, 1998), interest in the method has seen increased interest in recent years. In particular, Maziar Raissi, Paris Perdikaris, and G. E. Karniadakis, 2017 introduces and underlines the methodology, coining the term Physics Informed Neural Networks which has since been followed by many (Cuomo et al., 2022). Partly the reason for the sudden popularity of the subject is the development of the technology in the field of machine learning. Deep learning tool-kits developed in this field, now allows for readily available methods for computing auto-differentiation and neural network models. This in turn has allowed for better ease of development and progress in the exploring of the method and its potential applications. As it stands today, interest around PiNN as a technique is high, and the methodology has been adapted and tested one a widespread set of problems. Notably Pang, Lu, and G. E. Karniadakis, 2019 adapts the methodology into solving differential equations of non integer orders, Stiasny, Chevalier, and Chatzivasileiadis, 2021 adapts the methodology into a method they coin as RK-PINN (Runge Kutta-PINN), which was demonstrated to significantly speed up the computation of solutions to power system ODE's. Other examples of included applications within quantum chemistry, thermal fluid dynamics, plasma physics, material sciences, geophysics, molecular simulations, differential power systems, and more (Karniadakis et al., 2021). Even with the recently large amounts of developments and adaptions for the methodology, Cuomo et al., 2022 expresses an opinion that there is much more facets of the method which is still to be developed. Notably, various applications to real-world situations and equations is still an open challenge.

2.2 Machine learning

Machine learning is a field of computer science which originates in the application of techniques and theories from the field of statistics and regression analysis. As the names suggest the main concern of machine learning is the development and investigation of algorithms which are able to process and *learn from data*. More precisely, of the task within machine learning one may single out the methodologies related to supervised learning. Supervised learning algorithms often known as ML-models, are constructions which essentially have two main functionalities: Reading input from data, yielding an output, and changing their internal state dependent on how close this output is to the actual sought value. By repeatedly predicting and updating the internal state so that the model predicts closer and closer to the actual answer, the model can be said to be *learning*.

In the recent decade the field of machine learning has seen a large increase in interest and technological relevance. This is largely due to innovations in the sub-field known as deep-learning. The name stems from the aspect of the size, or amount of layers, in the models being deep. The success of these method can partly be attributed to the application of machine learning in cases where the amount of data processed in considered large (big data). Traditionally, machine learning has been largely dependent upon the quality of data, where careful considerations for validity had to be upheld. The ML approach was usually most effective when large focus was given to the appropriateness of the data, and other pathological effects such as overfitting, was accounted for. Deep learning solves these problems from the other perspective. The field has been successful leveraging sheer amount of data, fitting models with trainable parameters ranging in millions. In a sense, useful information can be extracted from these large sets of data from the virtue of the sets themselves being large. This property of deep learning has developed in tandem with the means of recording and producing relevant data. The ever increasing technological connectives of the world has allowed for the creation and collection of large pools of data. With these large sets of data, the deep learning methods are adapted to take advantage of scale.

When it comes to the task of finding the underlying structures of data, deep-learning and machine learning operate on the same principle. Mainly, having a *black box* model, trained over either large amounts of data in the deep learning case, or more selected and carefully considered data in the traditional machine learning sense. The black box nature of machine learning models introduces a problem in interpretability, as the potential of a qualitative inference of the relationships the models may capture is obscured.

2.3 Numerical methods and differential equations

Numerical methods is a set of techniques developed in order to solve scientific and mathematical problems by means of *numerical analysis*. That is, the technique of solving a problem by means of *approximation*. The problems these methods are typically applied to varies in a multitude of subjects and aspects, but one can characterize numeric methods overall as a means to solve equations. As a field, the subject is mature, having developed various techniques over long range of time, both as general methods useful in various contexts, and special methods for particularly difficult problems. Although the subject may be considered as old as mathematics itself (Fowler and Robson, 1998), the field itself has found paramount importance in computational science. Naturally, a large suite of numerical methods has been developed. Techniques such as Finite Element Methods, Runge-Kutta, and Iterative Methods are mature, optimized methods with a long history of applications within scientific computing.

Although numerical methods may be applied to any sort of equation problem, particular concern may be given toward the task of solving differential equations. When attempting to model natural phenomena, physical laws are often invoked and these laws tend to involve *rates of change* (Agarwal and O'Regan, 2009). Rates of change as known corresponds to derivatives of functions, and by constructing equations relating unknowns of functions and derivatives, one obtains a *differential equation*. Solving a differential equation (or a system of differential equations), generally means finding a function which by taking and combining derivatives satisfies a particular equation. Finding such solutions however, largely depend on the type of equation, and many problems may be considered difficult to solve by traditional means.

In the field of scientific computing, particular weight is given to the problem of solving *non-linear* problems. The reason being, that often, there is no feasible way of calculating solutions to these equations analytically. The techniques of numerical methods are then often the remedy in these situations, were solutions and approximations to solutions are computed using these techniques. Still, there exists particularly pathological problems, which are considered to be hard to solve even with standard numerical methods. In particular, highly non-linear PDE's, including effects of shocks in fluid equations, and convection dominance in equations of heat transfer, are examples of problems which may be difficult to solve with traditional numeric methods (Cuomo et al., 2022).

2.4 Applications, why use a PiNN?

From a mathematical modelling point of view, PiNN may be considered as a bridge between the techniques and methods found in the field of machine learning and numerical methods. From the standpoint of numerical computing, PiNN as a methodology is proving to become a flexible tool capable of solving non-linear differential equations, with added benefits from the flexibility of the ML-basis. In addition as pointed out by Karniadakis et al., 2021, PiNN is collectively demonstrated to be particularly effective in solving ill-posed and inverse problems (see section 3.7). In addition, as stated in Cuomo et al., 2022, PiNNs have been shown to perform better than conventional numerical methods in several cases. A benefit of PiNN as opposed to numeric methods such as finite and spectral methods is that PiNNs' computational costs are independent of number of grid points. In addition, PiNN methods trained on a grid of a particular resolution may easily utilized to predict over a finer or coarser grid. However, As stated in M. Raissi, P. Perdikaris, and G. Karniadakis, 2019, the originator of the term PiNN, the methodology is by no means meant to replace or surpass the already established numerical methods. In addition, for most ordinary problems as stated by Krishnapriyan et al., 2021, traditional numeric methods will perform better in general.

PiNNs may also be useful from a machine learning perspective. In machine learning models are created to extract relational structures hidden within large sets of data. ML models can largely be treated as black boxes, where interpretable information of the underlying structures remain hidden. As Karniadakis et al., 2021 writes, purely data driven models may fit the data well, but predictions may be physically inconsistent or implausible, which may lead to poor generalization performance. From the machine learning standpoint PiNN can be a way to manage the black box nature of machine learning models incorporating prior information in the form of differential equation. This in turn allows for models which may to a greater extent be interpretable, and find solutions to systems with ML models constrained to satisfy physical laws.

Chapter 3

Theory

3.1 Regression analysis

The Method of PiNN as it is realized in this thesis, builds upon the theory of Artificial Neural Networks (ANN) in the context of function regression. Regression analysis, is a form of statistical modeling, where the main concern is finding a function which describes the relationship between a dependent variable and an independent variable. Note however as regression analysis may also entice qualitative analysis of the describing functions. Terms dictating the effect of any independent variable in the optimal function may indicate the degree of causal relationships. As described, a neural network is often seen as a *black box* meaning that this type of analysis is difficult to perform when utilizing a neural network. However, the techniques of the field of regression analysis with regard to finding the optimal function which describes a set of data (dependent and independent variables) is to a large degree applicable to neural networks.

In order to fully understand how a neural network can be used as a *regression model*, first consider the general regression task. The main concern of regression analysis is to find some model which describes a set of data. Suppose a process which can be accurately described by a function:

$$f(\boldsymbol{x};\boldsymbol{\beta}), \tag{3.1}$$

Where $\boldsymbol{x} = [x_1, x_2, x..., x_q]$ are the function inputs (independent variables) and $\boldsymbol{\beta} = [\beta_1, \beta_2, ..., \beta_\omega]^T$ is a set of function parameters. Now, suppose there is available sample data $\boldsymbol{X}, \boldsymbol{y}$. Where

$$\boldsymbol{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{pmatrix}_1, \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{pmatrix}_2, ..., \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{pmatrix}_n, \tag{3.2}$$

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \tag{3.3}$$

Meaning there are n samples of independent variables (x_i^n) and dependent variable (y^n). The goal is now to find a function $\hat{f}(\boldsymbol{x}; \boldsymbol{\hat{\beta}})$ which *estimates* function 3.1. Suppose one such function is proposed, and let $\hat{y}_i$ denote the output of $\hat{f}$ for sample datapoints i. In order to judge how close this approximation is to the actual function f, a *loss function* (error/cost -function) $C(\hat{y}, y)$, is applied. Often, this loss function is the mean squared error (MSE). The MSE can be calculated as:

$$MSE = \frac{1}{n} \sum_{i}^{n} (y_i - \hat{y}_i)^2, \tag{3.4}$$

The goal of the regression is to find the parameters $\hat{\beta}$ of the function $\hat{f}$ so that equation 3.4 is minimized.

3.2 The multilayer perceptron model

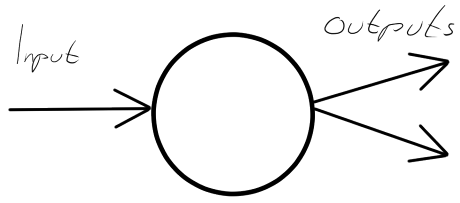


FIGURE 3.1: Single Neuron

The multilayer perception (MLP) model is the theoretic model behind artificial neural networks, serving as the foundation of neural networks a computational models. The MLP model dates to Rosenblatt, 1958, which proposed the perception as model capable of describing the way neurons in the brains of animal and humans processes and store information. At the time, the computational power needed in order develop a realization of the model in a computational form was yet to be readily available. As a result the model as a form of viable (and powerful) regression model, was not realized, and the model remained relatively obscure until the late 1980, notably with the work of Hornik, Stinchcombe, and White, 1989, which proved the universal approximation property of feed forward neural networks. This innovation marked a turning point of the method, and in addition the more readily available, ans still increasing computing power resulted in the model seeing wider adaptions. Today the multi-layer perceptron and feed forward neural network is regarded as the foundation for the technologies and techniques which comprise machine learning and deep learning; technologies which to this day have an ever-increasing importance in the modern world.

The model builds upon the concept of neurons (nodes) see figure 3.1. These neurons are modeled as capable of of taking multiple inputs including a bias term, an activation function, and yielding an output. The type of input and output can be dependent upon use cases, usually real or binary numbers. Further these neurons are arranged in layers such that a layer of neurons can take some input from a preceding layer and yield an output for the next 3.2.

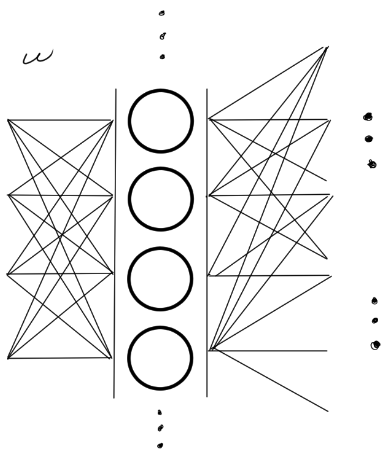


FIGURE 3.2: Layer of neurons

Between each layer of neurons there is an associated matrix of weights. As the name suggests, these values dictate the degree to which the output of the preceding layer is carried on into each neuron in the next layer. In addition, the output of the preceding layer, a bias term may act as part of the input to any neuron. With its own weighting, the bias term can be realized as a constant neuron added to each layer. An important element of the model is known as the activation function. Principally, this function introduces *non-linearities* between the layers, which in essence grants the neural network its general ability to approximate functions. Traditionally, this function has been the sigmoid shaped logistic function which maps any of the real numbers to the range $(0, 1)$. In order to compute the value of a particular node in a layer, first start by multiplying the values of the previous layer with the weights associated with the neuron. This is simply some linear combination of the values of nodes in the preceding layer. Then the activation function is applied to this linear combination, and this results in what is considered the node-value.

Neural network models are fairly flexible and has been applied in numerous contexts. The principal structure has been carried over to more specialized models such as image recognizing models which often use convolutional neural networks, or large scale models such as BERT (Devlin et al., 2018) which is used in natural language processing (NLP).

3.3 Simple feed forward NN example

To illustrate how a neural network computes a prediction figure 3.3 depicts a simple two-layer network. In this figure, there is a two-dimensional input layer consisting of input x_1 and x_2 with three layers of weights w_1, w_2 and w_3. Further, a sigmoid logistic function is chosen as the activation function for each node. A prediction also known as a forward pass is calculated by passing along the input over these layers.

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{3.5}$$

$$NN(x) = \sigma(w_3 \cdot \, \sigma\,(w_2 \cdot \sigma\,(w_1 \cdot x))), \tag{3.6}$$

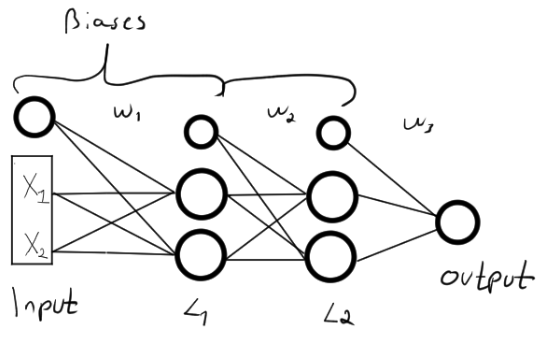


FIGURE 3.3: Simple two-layer NN structure

The universal approximation theorem as stated by Hornik, Stinchcombe, and White, 1989 implies that any continuous function $f : \mathbf{R^n} \to \mathbf{R}^m$ can be approximated with arbitrary accuracy by a *one-layer feed forward neural network.*

To further illustrate how these networks can be setup up to approximate a function, one can consider the case of approximating the XOR (exclusive or function), a non-elementary Boolean function. Table 3.1 summarizes the inputs and corresponding output of this function:

Input		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 3.1: XOR output

First, consider figure 3.4, illustrating a one neuron net which models an AND function.

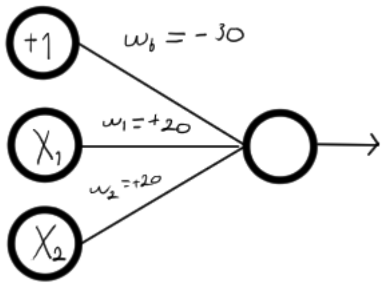


FIGURE 3.4: AND configuration of neurons

The output of this network is given as:

$$N(x) = \sigma(-30 + 20x_1 + 20x_2). \tag{3.7}$$

With the same activation function as before, this function approximates the AND function:

Input		Output	
A	B	A AND B	N(A,B)
0	0	0	$\sigma(-30) \approx 0$
0	1	0	$\sigma(-10) \approx 0$
1	0	0	$\sigma(-10) \approx 0$
1	1	1	$\sigma(10) \approx 1$

TABLE 3.2: AND function

Similarly, a NOT function is given as in figure 3.5

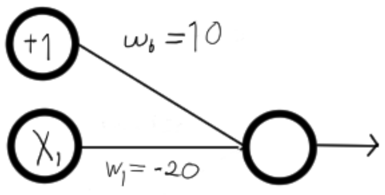


FIGURE 3.5: NOT function NN

With the output

$$N(x) = \sigma(10 - 20x_1), \tag{3.8}$$

Input	Output	
X	NOT X	N(X)
0	1	≈ 1
1	0	≈ 0

TABLE 3.3: NOT function

Consider the again the simple two-layer structure, but now with the weights given as in figure 3.6. The forward pass computed values for this network is summarized in table 3.4, and notice that this structure indeed approximates the XOR function.

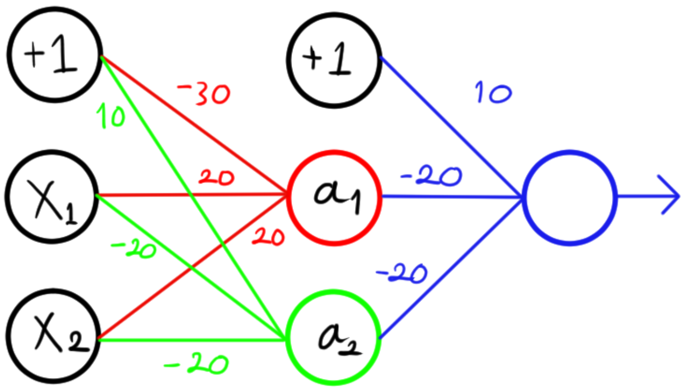


FIGURE 3.6: XOR function NN

Input		Hidden Layers		Output
x_1	x_2	a_1	a_2	N(X)
0	0	0	1	≈ 0
1	0	0	0	≈ 1
0	1	0	0	≈ 1
1	1	1	0	≈ 0

TABLE 3.4: XOR function

The approximation of the XOR function illustrates how such neural structures can be configured to approximate functions. In particular, one can see that a small set of these artificial neurons can approximate simple logical functions, which in conjunction amounts to a function of higher order.

3.4 Training a Neural Network

3.4.1 Gradient decent regression

Recall from section 3.1, how the goal of regression analysis is to find an approximate function $\hat{f}(\boldsymbol{x};\hat{\boldsymbol{\beta}})$ which minimizes an error function over data. An artificial neural network can be proposed as an approximation function, where the network weights $\boldsymbol{\omega}$ are the function parameters. The gradient decent algorithm (Theodoridis, 2020) is an algorithm which can be used to update these weights so that the error function minimizes.

Algorithm

Gradient decent is best described as an example: Suppose available data $\boldsymbol{X}, \boldsymbol{y}$, from a true distribution with random normal noise. That is the true, distribution of y given $\boldsymbol{x}$ is:

$$y = \beta_1 {x_1}^2 + \beta_2 x_2^2 + \beta_3 x_1 + \beta_4 e^{\beta_5 x_2} + \beta_0 + N(0, \sigma) \,. \tag{3.9}$$

Where N denotes a Gaussian distribution of the random noise and the beta parameters are unknown. Note that this choice of function is arbitrary, but non-linear in its parameters. The goal of the regression analysis is then to estimate these beta parameters from the given data. Fix a constant *learning rate* parameter l_r. The stochastic gradient decent algorithm in its simplest form works by the following steps:

1. Initialize function $\hat{f}(\boldsymbol{x};\hat{\beta})$ to be fit, with randomly chosen parameters $\hat{\beta} = Rand^n$.
2. For a data-point $(y_i, \boldsymbol{x}_i)$ compute loss-value: $C = \frac{1}{2}(y_i - \hat{f}(\boldsymbol{x}_i;\hat{\beta}))^2$.
3. Compute the gradient of the loss $\mu = \nabla_x C$.
4. Update the beta values with gradient value times a predetermined learning rate: $\beta \rightarrow \beta + l_r\, \mu$.
5. Repeat 2-4 over data points until some accuracy criteria is met.

This approach utilizes the fact that a gradient points towards the direction of steepest curvature. For the loss function this indicates the direction which minimizes the loss. By updating the parameters with this value, the function will move towards a local minimum. In some cases, this is can be sufficient, but in general the goal is to find a global minimum. The details of finding such a minimum is often more complicated, but in a general sense the method described here applies.

3.4.2 The backpropagation algorithm

Although there are various methods for training a neural network, the gradient decent algorithm is often used to fit neural networks to data. This description of this method also illustrates well the strategy of how to fit the potentially vast amounts of parameters of the neural network to a set of data. The gradient decent algorithm over a neural network works by computing what is known as s forward and a backwards pass. The forward pass is simply taking input and propagating the values until an output value is computed. The backwards pass is in essence computing the gradient of the model with respect to the parameters of the model as is described in step three of the gradient decent algorithm. Due to the potentially large number of parameters and the topology of the neural network, the computation of this gradient can be more complicated. As an initial example, consider the one-layer neural network as depicted in figure 3.7:

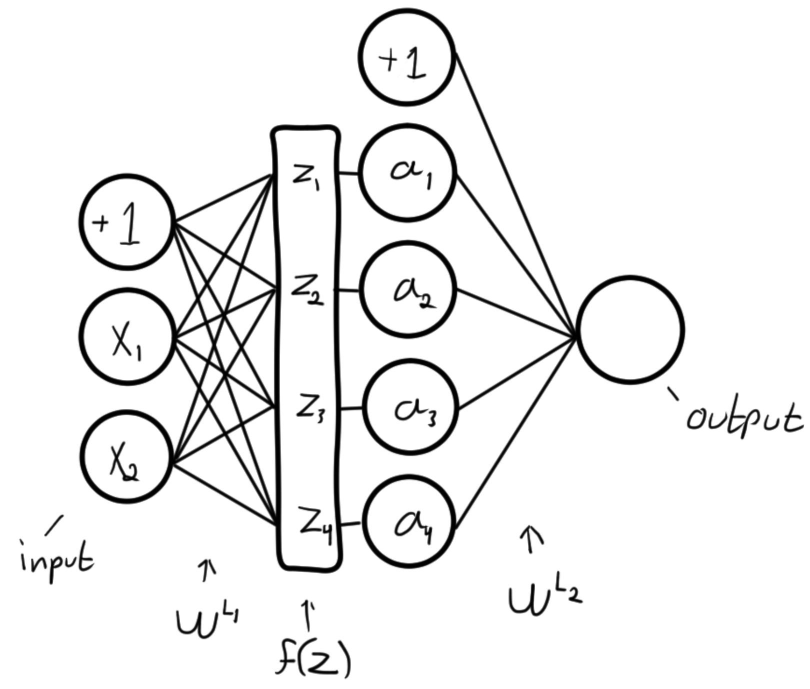


FIGURE 3.7: One-layer neural network

The output of this one-layer model can be calculated as

$$NN(x) = \omega^{L_2} \cdot f\left(\omega^{L_1} \cdot x\right). \tag{3.10}$$

When training a neural network, computing the output and storing the intermittent values for each node in the network is known as a *forward pass*. Here the first layer weights, are given by

$$\omega^{L_1} = \begin{pmatrix} \omega_{01}^{L_1} & \omega_{11}^{L_1} & \omega_{21}^{L_1} \\ \omega_{02}^{L_1} & \omega_{12}^{L_1} & \omega_{22}^{L_1} \\ \omega_{03}^{L_1} & \omega_{13}^{L_1} & \omega_{23}^{L_1} \\ \omega_{04}^{L_1} & \omega_{14}^{L_1} & \omega_{24}^{L_1} \end{pmatrix}. \tag{3.11}$$

In this scheme, for any given weight, the superscript denotes the layer instance, the subscripts denote the incoming node in first position and outgoing node in the second. Additionally, weights with index zero in the first position denotes the weight from the bias node. Hence, it is important to note the vector matrix transformation:

$$z = \omega^{L_1} \cdot x = \omega^{L_1} \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}. \tag{3.12}$$

That is an operation on the augmented vector x where the bias unit is injected as the first element. This scheme is convenient for expressing operations on the network including the bias term. Also, the intermittent vector z is defined to be the weighted transform of the input values prior to applying the activation function. Similarly, the second weight layer can be expressed as:

$$\omega^{L_2} = \begin{pmatrix} \omega_0^{L_2} & \omega_1^{L_2} & \omega_2^{L_2} & \omega_3^{L_2} & \omega_4^{L_2} \end{pmatrix}. \tag{3.13}$$

Since in this case there is only one output, there is no need for a second subscript index. Further a^{L_1} also known as a hidden layer, is defined to be the vector of node values after the activation function is applied appended with the layer bias value. That is:

$$a^{L_1} = \begin{pmatrix} 1 \\ f(z_1) \\ f(z_2) \\ f(z_3), \\ f(z_4) \end{pmatrix}. \tag{3.14}$$

In this instance, there is one hidden layer, which means there is one layer of hidden units and two layers of weights. a one-layer neural network with enough nodes can approximate any function. Using the quadratic error $C = \frac{1}{2}(y - f(\mathbf{x}))^2$. The gradient of the neural net with respect to weight parameters can be found by employing the chain rule (Goodfellow, Bengio, and Courville, 2016). For convenience consider first the total derivative, which is the generalization of the derivative. This derivative is defined as the linear equation which approximates the vector valued function at a particular point x. In essence, this derivative can be regarded as the Jacobian matrix of the function. Symbolically this derivative can be expressed as

$$\frac{dC}{d\mathbf{x}} = \frac{dC}{da^{L_1}} \odot \frac{da^{L_1}}{dz^{L_1}} \cdot \frac{dz^{L_1}}{d\mathbf{x}}. \tag{3.15}$$

Where the symbol $\odot$ is employed to denote the Hadamard product, which is the element wise product of matrices. This equation evaluates to:

$$\frac{dC}{d\mathbf{x}} = \frac{dC}{da^{L_1}} \odot f'(z^{L_1}) \cdot \omega^{L_1}. \tag{3.16}$$

This total derivative evaluates to a row vector (covector), and by taking the transpose one obtains the gradient.

$$\nabla_{\mathbf{x}} C = (\omega^{L_1})^T \cdot f'(z^{L_1}) \odot \nabla_{a^{L_1}} C. \tag{3.17}$$

A more verbose way to see that equation 3.17 is valid is by considering the partial derivatives of the network with respect to network parameters. In order to express this symbolically, begin by defining $\delta_j^{L_i}$ as a mediate value so that:

$$\delta_j^{L_i} = \frac{\partial C}{\partial z_j^{L_i}}. \tag{3.18}$$

That is, the partial derivative of the cost function C with respect to z-component j in layer L_i Since this examples consists of two layers L_1, L_2 where L_2 constitutes the output layer, one can write out the $\delta_j^{L_2}$ as:

$$\delta_j^{L_2} = \sum_k^{n_{L_2}} \frac{\partial C}{\partial a_k^{L_2}} \frac{\partial a_k^{L_2}}{\partial z_j^{L_2}}. \tag{3.19}$$

Recall that the $a_k^{L_i}$ are simply the activation function f applied to $z_k^{L_i}$, which means that that the rightmost partial derivative in equation only evaluates to a nonzero value when $k = j$. The expression then becomes

$$\delta_j^{L_2} = \frac{\partial C}{\partial a_j^{L_2}} \frac{\partial a^{L_2}}{\partial z_j^{L_2}}, \tag{3.20}$$

$$= \frac{\partial C}{\partial a_j^{L_2}} f'(z_j^{L_2}). \tag{3.21}$$

Now since, the gradient is a vector of partial derivatives, one can see that evaluating equation 3.21 over all j in L_2 y can be denoted as:

$$\delta^{L_2} = \nabla_{a^{L_2}} C \odot f'(z^{L_2}). \tag{3.22}$$

The vector δ^{L_i} is usually referred to as the *error* of layer i. Notice that this is only half of equation 3.17. In the same manner as for a component of L_2 error $\delta_j^{L_2}$, one can derive component i for the error of the next layer L_1. First, consider the formula in the case for a general network with potentially more than two layers. Suppose that the error δ^{l+1} is known and one wishes to express the error of the preceding (next) layer l:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}, \tag{3.23}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}, \tag{3.24}$$

$$= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}. \tag{3.25}$$

Following the calculation of z values in equation 3.12, in a forward pass the z^{l+1} values were calculated as $\omega^{l+1} a^l$. For a single z_k component in layer $l + 1$ this is expressed as:

$$z_k^{l+1} = \sum_j \omega_{kj}^{l+1} a_j^l + b_k^{l+1}, \tag{3.26}$$

$$= \sum_j \omega_{kj}^{l+1} f(z_j^l) + b_k^{l+1}. \tag{3.27}$$

Where the b_k^{l+1} denotes the bias term. The derivative of this yields:

$$\frac{\partial z_k^{l+1}}{\partial z_j^{l+1}} = \omega_{kj}^{l+1} f'(z_j^l). \tag{3.28}$$

Hence

$$\delta_j^l = \sum_k \omega_{kj}^{l+1} \delta_k^{l+1} f'(z_j^l), \tag{3.29}$$

$$\rightarrow \delta^l = ((\omega^{l+1})^T \delta^{l+1} \odot f'(z^l). \tag{3.30}$$

A similar chain of arguments can be applied to demonstrate the following two properties.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l, \tag{3.31}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{3.32}$$

Now, recall that the example was a simpler two-layer structure. For this example this means that $z^{L_1} = \boldsymbol{x}$ (and $f'(x) = \mathbf{1}$, since the activation function was not applied directly to $\boldsymbol{x}$). Inserting 3.22 into 3.30 yields:

$$\delta^x = \nabla_x C = ((\omega^{l+1})^T \nabla_{a^{L_2}} C \odot f'(z^{L_2}). \tag{3.33}$$

Which is equivalent to equation 3.17. The gradient values for weight parameters can be obtained from the layer errors and equation 3.32. For the given example this equates to:

$$\nabla_{\omega^{L_2}} C = \delta^{L_2} \otimes (a^{L_1})^T, \tag{3.34}$$

$$\nabla_{\omega^{L_1}} C = \delta^{L_1} \otimes (\mathbf{x})^T. \tag{3.35}$$

Where $\otimes$ signifies the vector outer product. Recall from section 3.4.1, that these gradients can be used to update the parameters, when training the network. Now, this method is illustrated for a one-hidden-layer neural net example. For a general n-hidden-layer network, this process of calculating weight gradients amounts to the following:

- Suppose network with hidden-layers $L_1, L_2, ...L_n$.
- Calculate first delta value $\delta^{L_n} = f'(z^{L_n}) \odot \nabla_{a^{L_N}} C$.
- Propagate and calculate deltas for all layers: $\delta^{L_{i-1}} = f'(z^{L_{i-1}}) \cdot (\omega^{L_i})^T \delta^i$.
- Calculate gradients $\nabla_{\omega^{Li}} C = \delta^i \otimes (a^{i-1})^T$.

Typically, to limit number of computations, the intermittent values such as the Z-values are computed during the forward pass, and stored until called upon in the backwards pass. Now, with the sum total of the methodologies described give a general overview of how a neural network algorithm works. Note also, that the forward and backwards pass algorithm allows for finding the derivative of the neural network with respect to the input, which will be important later one when the PiNN methodology is described.

3.4.3 Considerations for neural network training

There are a number details to keep in mind when applying and training a neural network. Not necessarily particular to neural network models, but regression models in general is the "cleanliness" of the data. That is, a machine learning model is only as good as the data it is fed. As such, a number of precautions should be considered in particular of the appropriateness of the data being trained upon. Normalization is the act of re-scaling the data withing some range, typically so that the maximum occurring value maps to one, and conversely the minimum maps to zero (another common mapping is $[1, -1]$). The necessity of normalization varies (Baeldung, 2020). In some cases it might not be necessary at all. As an example where it may be necessary to normalize is when one has a (regression) network of one-dimensional output, or if all outputs are in the same range.

The size of the network may also be important. Even though the universal approximation theorem states that a single layer neural network is capable of approximating any function, it is important to note that one may achieve better result by regulating the size and amount of layers. Recall that one can describe network layers as a series of linear weight and nodes and a nonlinear activation function. For the flexibility of the network one may find better results by limiting the size of layers and increasing amount (Simard, Steinkraus, and Platt, 2003)

3.4.4 Feature space and activation functions

A central component of Neural Network models is the activation function. This function acts as the non-linear component of a network model. That is, a chain of linear mappings between layers of neurons, in itself results in another linear map. By introducing an activation function between the layers, the mappings become non-linear and cannot be reduced to one linear map. The benefit of this is the increased flexibility of the *solution space*. Illustratively, a simple neural network classifier may assign labels to points belonging to a particular group (Note that a neural net *classifier* is not the same thing as a neural net *regression*, however the way activation function behaves are analogous, but easier to explain for the classifier). These points exist in a space called the*feature space*. A linear classifier essentially constructs hyperplanes separating these points in feature space so that the group of points is separated by this hyperplane, and can hence be assigned it's appropriate label. However, most often these features are not necessarily linearly separable, meaning that a flat hyperplane will not suffice. Adding activation functions and hence non-linearity between the layers allows this border more flexibility. With this flexibility a neural network can adapt to data with more complex feature distribution.

The type of activation function utilized may vary upon the training task, examples of which can range from classification, regression, and image recognition. In particular for the regression case, common examples include activation functions such as: Linear, ReLu, Logistic (sigmoid), and Hyperbolic Tangent.

3.4.5 Optimizer algorithms

As described in section 3.4.2, gradient decent is perhaps the simplest way to train/fit a neural network. The algorithm which dictates the strategy of changing parameters of models in order to minimize or maximize some value is known as an optimizer algorithm. In the neural network context, there are several optimizers, with different benefits and drawbacks. Of these optimizers, the ADAM (Doshi, 2019) algorithm is generally understood to be the most generally efficient for simple learning tasks. Another algorithm which has seen much use in the context of PiNN is the LBFGS-(Limited Memory Broyden Fletcher Goldfarb Shanno) algorithm. In particular, the LBFGS algorithm has been a favourite in PiNN research, as seen early on in the developments by Lagaris, Likas, and Fotiadis, 1998. As Cuomo et al., 2022 writes, even though the research is not settled, the most commonly used optimizer algorithms is ADAM and LBFGS, where generally the difference between the performance of the two algorithms is dependent upon neural network model size. ADAM seems to be more effective for for larger size models, whilst LBFGS is more efficient for smaller models.

3.5 Regularization

A prevalent issue in mathematical modelling is the problem of over-fitting. Overfitting as the name suggest is when the model is fit to noise or disturbances in the data which does not principally originate from the process which the model is meant to regress to. In other word, the model picks up on too particular instances in the data, following a solution curve which is less generalized, and fits the given data too well and will yield worse predicting results on unseen data (Theodoridis, 2020). This issue is of large concern in principle all form of regression analysis, including the field of machine learning. Typically this occurs when a model is over parameterized, meaning it has to more trainable parameters than true parameters dictating the underlying process. There are various ways to remedy this, some dependent upon the type of model utilized, while others are more general. Regularization is one of the more general techniques, which can be applied to most machine learning algorithms. As Theodoridis, 2020 writes, regularization is a mathematical tool applied as to impose a priori information on the structure of a model solution. In essence, regularization is a method which restricts the model in some way, hindering over-fitting.

3.6 Runge-Kutta schemes

The realization of the PiNN methodology further on in this thesis is partly dependent on discrete time iterative methods. This is due to two reasons, the first being that in order to test the methodologies, simulated data was generated using principles dependent on these techniques. The other reason is the specific PiNN technique referred to as Discrete Time PiNN 3.8.3, builds upon the principles which underlines a general Runge Kutta scheme. Runge Kutta is considered as a family of numeric methods consisting of iterative calculation of discrete approximations to solutions to non-linear equations. As described in Iserles, 2008a, general explicit- and then implicit-scheme Runge-Kutta schemes can be described as follows. First suppose any non-linear differential equation given as:

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}), \tag{3.36}$$

$$\mathbf{y}(t_0) = \mathbf{y}_0. \tag{3.37}$$

Suppose one wishes to find the solution $\mathbf{y}$ of this system. First, consider a *quadrature* which is a method of replacing an integral with a finite sum. Let ω be a *weight function*, which is a positive function on $[a, b]$ such that

$$0 < \int_a^b \omega(\tau) d\tau < \infty, \tag{3.38}$$

$$| \int_a^b \tau^j \omega(\tau) d\tau | < \infty, j = 1, 2, ... \tag{3.39}$$

A weighted integral of a function $g(t)$ can then be approximated as:

$$\int_a^b g(\tau)\omega(\tau) d\tau \approx \sum_{j=1}^{v} b_j g(c_j). \tag{3.40}$$

$b_1, b_2, ..., b_v$ denote the quadrature weights, and $c_1, c_2, ..., c_v$ is the nodes, which are dependent upon ω a and b and independent of $g(t)$ A quadrature can be applied to the differential equation system 3.36-3.37. First consider the time stepping scheme, integrating from t_n to $t_{n+1} = t_n + h$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} f(\tau, \mathbf{y}(\tau)) d\tau = \mathbf{y}_n + \int_0^1 f(t_n + h\tau, \mathbf{y}(t_n + h\tau)) d\tau. \tag{3.41}$$

The latter integral can be replaced by a quadrature.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \sum_{j=1}^{v} b_i f(t_n + c_j h, \mathbf{y}(t_n + c_j h)). \tag{3.42}$$

3.6.1 Explicit RK

One may notice that equation 3.42 is dependent upon unknown values. Namely $\mathbf{y}(t_n + c_j h)$, for $c_1, c_2, ..., c_v$. These values must be approximated. Let each approximation of $\mathbf{y}(t_n + c_j h)$ be denoted as ξ_j for $j = 1, 2, ..., v$ An *explicit* Runge-Kutta method is the process of approximating each xi_j. Let $c_1 = 0$, then $\xi_1 = \mathbf{y}_n$. A chain of approximations can be set up, and the value of $\mathbf{y}_n$ is updated.

$$\xi_1 = \mathbf{y}_n, \tag{3.43}$$

$$\xi_2 = \mathbf{y}_n + ha_{2,1}\boldsymbol{f}(t_n, \xi_1), \tag{3.44}$$

$$\xi_3 = \mathbf{y}_n + ha_{3,1}\boldsymbol{f}(t_n\xi_1) + ha_{3,2}\boldsymbol{f}(t_n + c2h, \xi_2) \tag{3.45}$$

$$\vdots \tag{3.46}$$

$$\xi_v = \mathbf{y} + h\sum_{I=1}^{v-1} a_{v,i}\boldsymbol{f}(t_n + c_ih, \xi_i), \tag{3.47}$$

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + h\sum_{j=1}^{v} b_j\boldsymbol{f}(t_n + c_jh, \xi_j). \tag{3.48}$$

The matrix $A = (a_{j,i})_{j,i=1,2,...,v}$ where missing values are set to zero (hence lower triangular), is known as the Runge-Kutta (RK) matrix, while b_i and c_i are still known as weights and nodes. Also, v is known as the number of *stages*. $A, \mathbf{b}, \mathbf{c}$ are usually summarized in a *butcher table* (RK tableaux).

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array}$$

TABLE 3.5: General butcher table

3.6.2 Implicit RK

The method of implicit Runge-Kutta is defined similarly to the explicit methods. In the explicit scheme, each xi_i is estimated from the preceding xi_{i-1}. In an implicit scheme, each ξ_i approximation depends on all of the other ξ_j. In more detail, consider:

$$\xi_i = \mathbf{y}_n + h\sum_{i=1}^{v} a_{j,i}\boldsymbol{f}(t_n + c_ih, \xi_i), j = 1, 2, ..., v, \tag{3.49}$$

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + h\sum_{j=1}^{v} b_j\boldsymbol{f}(t_n + c_jh, \xi_j). \tag{3.50}$$

In this scheme the **A** matrix is arbitrary, with the requirement

$$\sum_{I=1}^{v} a_{j,i} = c_j, j = 1, 2, ..., v. \tag{3.51}$$

Now, instead of a recursive chain of approximations, the expressions 3.49-3.50 constitutes a system of vd equations, for $\mathbf{y} \in \mathbb{R}^d$. These systems of equations are more computationally expensive to solve compared to the explicit methods, but an implicit scheme can have advantages such as superior stability, and performance over *stiff equations* (Iserles, 2008b). There are various strategies for solving the system of equations yielded from an implicit scheme, as an example one can utilize iterative methods to recursively update the equations.

3.7 Inverse problems

Inverse problems is the concept of determining the factors necessarily involved in creating a state of a system. As, the name suggest, it is the task of: Given a state of a system, determined the determining factors involved in the process which produced that state. Generally, the inverse problem task is considered a difficult problem, where finding true solutions may require sophisticated algorithms. As described by Theodoridis, 2020, most tasks in machine learning can be considered inverse problems. Further, most inverse problems are typically *ill-posed*, as opposed to well-posed. Well-posed means that a problem has the properties:

1. A solution exists
2. The solution is unique
3. The solution is stable with regards to the initial conditions.

3.8 Physics informed neural networks

M. Raissi, P. Perdikaris, and G. Karniadakis, 2019 coined the term "Physics informed neural networks" (PINN). Their description of the method and applications has set the foundation for the recent developments for the combined use of neural networks and differential equations. The techniques main feature is performing regression and model estimation utilizing the properties of differential equations and neural networks. Often, in the contexts of physical modelling, the dynamics of the system is understood to follow some set of differential equations. The main idea behind physics informed neural networks is to leverage the required properties of a solution of the differential equation as information on which to, in some sense, regularize a neural network model. Recall from section 3.5 regularization in regression modeling is the act of limiting some characteristic of the model parameters such as amount or size, typically to prevent overfitting. This is usually achieved by adding parameter cost functions to the regression model. The reason for doing this varies, and notably it introduces some bias to models which normally would have been unbiased. In the physics informed approach, the differential equation acts as the regularization cost of the parameters of the model.

For the purposes of this thesis, a distinction is made. Although the term Physics Informed Neural Networks (PiNN) has been widely adopted as the catch all name for utilizing loss from differential equations in a machine learning context (Cuomo et al., 2022), onward through the rest of this thesis, PiNN will refer to the specific framework set by M. Raissi, P. Perdikaris, and G. Karniadakis, 2019. Further, the use of a machine learning model purely trained to satisfy a differential equation will be referred to as a *Neural Solver*.

3.8.1 Neural networks as solution models of differential equations

An early development by Lagaris, Likas, and Fotiadis, 1998, describes how shallow neural networks can be applied as components of trial solutions to differential equations, which can be fitted in such a way that the trial solution satisfies the differential equation. M. Raissi, P. Perdikaris, and G. Karniadakis, 2019 further develops a similar methodology as their coined PINN methodology, leveraging modern computational capabilities. To gain an understanding of the main idea behind the

PINN methodology one may first consider the neural solver process used by (Lagaris, Likas, and Fotiadis, 1998)

Suppose one wishes to model a system governed by a differential equation of the general form:

$$G(\mathbf{x}, \Psi(\mathbf{x}), \nabla\Psi(\mathbf{x}), \nabla^2\Psi(\mathbf{x})) = 0. \qquad (3.52)$$
$$\mathbf{x} \in D$$

Where $\mathbf{x} = (x_1, x_2, ..., x_n), D \subset R^N$ is the definition domain and $\Psi(\mathbf{x})$ is the sought solution to the differential equation.

The method of collocation is adopted, that is, discretization of the domain D and its boundary S into sets of points $\hat{D}, \hat{S}$. Suppose a trial solution $\Psi_t(\mathbf{x})$ taking the square error of the differential equation, the task becomes to minimize:

$$L_{DE} = \sum_{||\mathbf{x}_i \in \hat{D}} G(\mathbf{x}_i, \Psi(\mathbf{x}_i), \nabla\Psi(\mathbf{x}_i), \nabla^2\Psi(\mathbf{x}_i))||^2. \qquad (3.53)$$

The trick to this method is to employ the neural network in the trial solution. Dependent on the differential equations and boundary conditions, one can design a trial solution Ψ_t which satisfies boundary conditions and incorporates a neural network. Case in point one

$$\Psi_t(\mathbf{x}) = A(\mathbf{x}) + F(\mathbf{x}, N(\mathbf{x}, \omega)). \qquad (3.54)$$

As shown in section 3.4.2 derivatives of general neural nets can be obtained. The realisation of this technique can be adapted in a traditional neural network training implementation. The network is trained by taking predictions of $\Psi_t(x_i)$ for $x_i \in \hat{D}$. Correspondingly, derivatives of trial solution must also be calculated. Then calculate loss value from equation 3.53, and compute gradients with respect to the neural network. These gradients can the be used to update the network as described in the gradient decent algorithm (section 3.4.1). By constructing the trial solution in such a way that the boundary conditions are satisfied, this algorithm may find a solution of the differential equation.

3.8.2 The PiNN methodology

Continuous time

Similarly, the PiNN methodology set by M. Raissi, P. Perdikaris, and G. Karniadakis, 2019 also utilizes the neural network as a solution to the relevant differential equation. The main difference being the approach of this methodology is to compute a data-driven solution. In particular, given a situation where the goal is to find a model which incorporates some given data, and satisfies partial differential equation of the general form:

$$u_t + \mathcal{N}[u] = 0, x \in \Omega, t \in [0, T]. \qquad (3.55)$$

Where in this description, the convention of subscripts denoting derivatives is utilized. Hence, u_t denotes the t derivative of $u(t, x)$ which denotes the hidden/latent solution to the equation. Ω denotes a subset of $\mathbb{R}^D$. N (not to be confused as to denote neural network) is a non-linear differential operator, which in essence, can be equated with the "x-derivative" components of the differential equation. Further, f is defined to be the left hand side of the equation $f = u_t + \mathcal{N}[u]$. One can then define the solution which satisfies u and f over the data as two neural networks with shared parameters. In principle this yields what is coined as a *Physics Informed Neural Network* (PiNN). The loss function over data for this network can be realized as the combined mean square error loss:

$$MSE_{pinn} = MSE_u + MSE_f. \tag{3.56}$$

Where

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2. \tag{3.57}$$

And

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2. \tag{3.58}$$

Where the sets $\{t_u^i, x_u^i, u^i\}_{i=1}^{N_u}$ and $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ respectively denote the training data on u and f. Additionally, boundary- and initial conditions may be added as additional loss terms in the overall loss-value, in order to make sure these conditions are satisfied in the solution. Although these two methodologies are similar, the PiNN framework differs in the sense that is data-driven. meaning that the method itself depends on some sets of data. In a sense, the goal is to perform a machine learning task over data sampled from a process with known/supposed governing equations. In this process the equations themselves are added to serve as a realization mechanism, which ideally improves upon the learning task. On the other hand, the process described by Lagaris, Likas, and Fotiadis, 1998 (section 3.8.1) corresponds to method of using neural networks as pure equation solvers, and hence can be referred to as a neural solver method.

3.8.3 Discrete time PiNN

An additional methodology set by M. Raissi, P. Perdikaris, and G. Karniadakis, 2019, denoted as *discrete time Physics informed Neural Networks*, can also be considered. This methodology builds upon a generalization of implicit Runge-Kutta schemes. First Let $u(t, x)$ denote the sought solution to a differential equation of the general form 3.55. Then consider applying the implicit form RK approximation scheme 3.49, 3.50. Recall that $\xi_i = \mathbf{y}(t_n + c_j h)$ is the ith stage approximation for the solution of a differential equation under the RK scheme . Let ξ_i^n denote the approximation of $u(n + c_i, x)$ at stage i and time-step n. Additionally set u^{n+1} to denote the end stage $n + 1$ approximation. The general-form (arbitrary butcher table) Runge-Kutta scheme for this set-up can then be denoted as:

$$\xi_i^n = u^n - h\sum_{j=1}^{v} a_{i,j}\mathcal{N}[\xi_j^n], \ \ i = 1, ..., q, \tag{3.59}$$

$$u^{n+1} = u^n - h\sum_{j=1}^{q} b_j\mathcal{N}[\xi_j^n]. \tag{3.60}$$

These two equations can be rearranged. Let u_i^n denote the ith stage approximation of u^n

$$u_i^n = \xi_i^n + h\sum_{j=1}^{v} a_{i,j}N[\xi_i^n], \ \ i = 1, ..., v, \tag{3.61}$$

$$u_{v+1}^n = u^{n+1} + h\sum_{j=1}^{v} b_jN[\xi_j^n], \tag{3.62}$$

A neural network can be used to predict the ξ_i^n stages and the end-stage approximation u^{n+1}

$$NN(x, n) = [\xi_1^n(x), \xi_2^n(x), ..., \xi_v^n(x), u^{n+1}(x)]. \tag{3.63}$$

Using an appropriate butcher table and equations 3.61, 3.62, these values are used to calculate $[u_1^n(x), ..., u_v^n(x), u_{v+1}^n(x)]$ as the final output. Loss values can be computed using sample data $x_{sample}^n, u_{sample}^n$.

$$Loss = \frac{1}{n^2}SSE_n + \frac{1}{n_b^2}SSE_b. \tag{3.64}$$

With, $\{x^{n,i}, u^{n,i}\}_{i=1}^{N_n} \in [x_{sample}^n, u_{sample}^n]$:

$$SSE_N = \sum_{j=1}^{v+1}\sum_{i=1}^{N_n} |u_j^n(x^{n,j}) - u^{n,i}|^2. \tag{3.65}$$

And SSE_b, correspond to square error from points on the boundary, and boundary condition specified for the differential equation. Optimizing over this loss yields a *Discrete Time Physics Informed Nerual Network.*

3.8.4 Parameter discovery

An aspect of numeric modelling is estimating appropriate governing parameters of the differential equations. This problem is realized as an instance of an inverse problem, meaning that one wishes to *learn* the parameters of a physical model which has driven forth a realization of this model in the form of recorded data.

Continuous time PiNN

By adding parameters as trainable variables in the PiNN methodology, they can be estimated from an optimal fit. In more detail, consider the PiNN methodology as a model constructed as a neural network model with additional parameters

$\lambda_1, \lambda_2, ..., \lambda_\tau = \boldsymbol{\lambda}$:

$$PiNN(x) = f(NN(x), \boldsymbol{\lambda}). \tag{3.66}$$

Suppose data consistent with a differential equation with parameters

$$\boldsymbol{\lambda}_{true} = \lambda_1^{true}, \lambda_2^{true}, ..., \lambda_\tau^{true}. \tag{3.67}$$

Optimizing a PiNN model over the dual loss from the differential equation and datapoints, will yield the model

$$PiNN_{fitted}(x) = f(NN(x), \hat{\hat{\boldsymbol{\lambda}}}). \tag{3.68}$$

The converged parameters of the model will then be estimations of the parameters governing the underlying data.

Discrete time PiNN

Discrete PiNN can also be adapted for parameter discovery. Consider again a differential equation of the from $u_t + \mathcal{N}[u, \boldsymbol{\lambda}], \boldsymbol{\lambda} = \lambda_1, .., \lambda_\tau$. Apply the general form Runge Kutta and obtain the following schemes:

$$u^n = \xi_i^n + h \sum_{j=1}^{v} a_{i,j} N[\xi_i^n, \boldsymbol{\lambda}], \; i = 1, ..., v, \tag{3.69}$$

$$u^{n+1} = \xi_i^n + h \sum_{j=1}^{v} (a_{ij} - b_j) N[\xi_j^n, \boldsymbol{\lambda}], \; i = 1, .., v. \tag{3.70}$$

Again, a neural network is placed to predict the RK stages

$$NN(x, n) = [\xi_1^n(x), \xi_2^n(x), ..., \xi_v^n(x)]. \tag{3.71}$$

The main difference in this scheme is the provision of two sets of sampled datapoints: $u_{sample}^n, u_{sample}^{n+1}$. Using equations 3.69, 3.70, the output is the two vectors:

$$[u_1^n(x), .., u_v^n(x), u_{v+1}^n], \tag{3.72}$$

$$[u_1^{n+1}(x), .., u_v^{n+1}(x), u_{v+1}^{n+1}]. \tag{3.73}$$

The neural network with the added parameters $\boldsymbol{\lambda}$ can be trained by minimizing means/sum of square errors.

$$Loss = SSE_n + SSE_{n+1}. \tag{3.74}$$

Where SSE is calculated as in 3.65.

3.9 Compartment models

Compartment models is a general class of mathematical models, which generally consists of modelling the transmission of properties between compartments. Typically, these models can be applied in order to describe physical processes such as disease spread in a population, fluid flow systems, or system of traffic. Compartment models as a way do discrete and solve problem which otherwise would be much harder to solve. The SIR model (BlackwoodJulie and Childs, 2018) is a typical example of a simple compartment model, which describes an infectious disease scenario. SIR , an abbreviation for susceptible, infectious and recovered, and these groups acts as the compartments subdividing a population. Disease spread and recovery is then modelled as simple differential equations dependent on respective compartment's population sizes. In detail, one can state the SIR model as:

$$\frac{dS}{dt} = -\frac{\beta IS}{N}, \tag{3.75}$$
$$\frac{dI}{dt} = \frac{\beta IS}{N} - \gamma I, \tag{3.76}$$
$$\frac{dR}{dt} = \gamma I. \tag{3.77}$$

Where parameters β and γ denote infection rate and recovery rate, and N denotes population size. This model is a non-linear differential model. Although there are methods of analytically solving special cases of these equations (Kröger and Schlickeiser, 2020), in practice numeric methods in order to solve the model. In infectious disease modelling, this model is usually expanded by the addition of compartments. One might augment the SIR model with compartments such as: dead (D), vaccinated (V), exposed (E), etc. The addition of compartments then can be modelled with different transition dynamics between compartments, such as population within the vaccinated (V) compartment have a greatly reduced chance of transitioning into the infectious (I) compartment ($\lambda_{V\to I} << \beta_{S\to I}$). The same methodology is not limited to disease spread however, other examples may include discretized flow systems, modelling of the circulatory system, and modelling predator-prey population dynamics (Thompson and Freedman, 1982). As in the case for circulatory system, continuous diffusion systems, usually modelled as partial differential equations (PDE), can be discretized using the finite difference method (LeVeque, 2007). The discrete version of these models can be realized as a variant of compartment models.

3.9.1 General multi-compartment models

Illustratively, one can describe a general multi-compartment model as follows. Suppose a system of n compartments, which contain some intrinsic property x_i. Suppose that this property varies dependent over time t and values for *connected* compartments with some parameters $\lambda_{x_i \to x_j}$, so that one has:

$$\frac{dx_i}{dt} = \sum_{j=1}^{n} (\lambda_{x_j \to x_i} f(x_i, x_j) - \lambda_{x_i \to x_j} g(x_i, x_j)). \tag{3.78}$$

Which in informal terms states: The change in x_i is the sum of property flowing in and property flowing out. The functions f and g denote combinations of compartment values. As an example, in the SIR model, equation 3.75, $f(S,I) = \frac{S \cdot I}{N}$

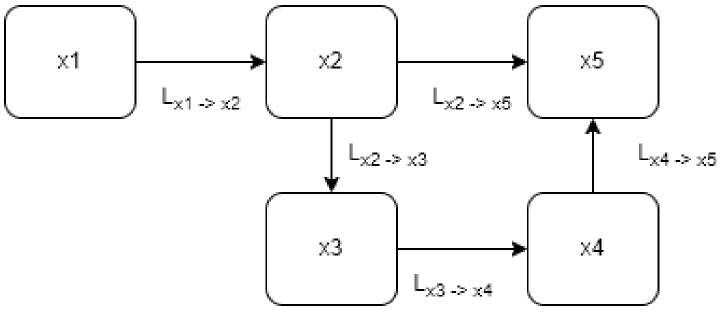


FIGURE 3.8: A compartment structure

An arbitrary example is given in 3.8. Symbolically this case can be described as

$$\frac{dx_1}{dt} = -L_{x_1 \to x_2} \cdot x_1, \tag{3.79}$$
$$\frac{dx_2}{dt} = L_{x_1 \to x_2} \cdot x_1 - L_{x_2 \to x_3} \cdot x_2 - L_{x_2 \to x_5} \cdot x_2, \tag{3.80}$$
$$\frac{dx_3}{dt} = L_{x_2 \to x_3} + x_2 \cdot x_1 - L_{x_3 \to x_4} \cdot x_3, \tag{3.81}$$
$$\frac{dx_4}{dt} = L_{x_3 \to x_4} \cdot x_3 - L_{x_4 \to x_5} \cdot x_4, \tag{3.82}$$
$$\frac{dx_5}{dt} = L_{x_2 \to x_5} \cdot x_2 + L_{x_4 \to x_5} \cdot x_4. \tag{3.83}$$

3.9.2 Numerical solutions to compartment differential equations

Compartment models can easily be solved with numerical methods such as time stepping schemes, an example of which would be the Runge-Kutta methods 3.6. A simple variant of RK schemes is the Euler method which corresponds to a 1-step RK-method, which can be realized both explicitly and implicitly. Illustratively the simplest case of an explicit Euler method solution can be given. Suppose the above compartment model system 3.79 -3.83. Suppose also an initial condition $IC = \mathbf{x}_0 = [x_0^1, x_0^2, x_0^3, x_0^4, x_0^5]^T$ (Now superscript denotes variable and subscript denotes timestep.). Further, suppose a step size h. The Euler scheme is realized as:

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + hf(t_n, \boldsymbol{y}_n). \tag{3.84}$$

$f(t_n, \boldsymbol{y}_n)$ will in this case correspond to the right hand side of equations 3.79 -3.83. Hence

$$x_1^1 = x_0^1 + h(-L_1 \cdot x_0^1), \tag{3.85}$$
$$x_1^2 = x_0^2 + h(L_1 \cdot x_0^1 - L_2 \cdot x_0^2 - L_3 \cdot x_0^2), \tag{3.86}$$
$$x_1^3 = x_0^3 + h(L_2 \cdot x_0^2 \cdot x_0^1 - L_4 \cdot x_0^3), \tag{3.87}$$
$$x_1^4 = x_0^4 + h(L_4 \cdot x_0^3 - L_5 \cdot x_0^4), \tag{3.88}$$
$$x_1^5 = x_0^5 + h(L_3 \cdot x_0^2 + L_5 \cdot x_0^4). \tag{3.89}$$

The $\boldsymbol{x}_1$ can be used to calculate the next step. This is repeated until a solution is found.

Chapter 4

Implementation and Case studies

This chapter concerns the realization and use cases for PiNN through four case problems. These cases are presented with distinct problem formulations, and a solution to these cases are found by means of implementing the PiNN methodology. Overall the goal of these case studies is to obtain an understanding of the methodology and how it can be adapted to specific case problems which were found to be of interest. These cases were selected in a way as to investigate the utility of the method, and to prepare an overall methodology which could be adapted into a real-world case problem. The realized case problems range in size and scope.

Case one, Neural Solver The first case is as a simple demonstration on how to realize the main idea behind PiNNs, training a neural network to satisfy a differential equation

Case Two, Lotka Votlerra PiNN Case two is also a demonstration, realizing both the continuous- and discrete-time PiNN methodologies highlighting the regularization capabilities of the former.

Case three, PiNN-SIR This case further builds upon the demonstrated methodology, developing and performing tests in order to scope out a larger framework which ultimately was to be applied in case four. Specifically this framework applies PiNN to the SIR compartment model. Subsequently, in order to gauge the overall performance of the parameter discovery and regularization properties, the PiNN methodology is tested over simulated SIR models of varying underlying parameters.

Case four, PiNN bolus simulation Case four is realized as the culmination of the knowledge developed throughout the thesis, adapting adapting the refined framework in a real-world biomedical case-problem. This case problem builds upon a base compartment method used to predict the cardiac output (CO) of a patient, given some small set of patient data including height weight and gender. This model is augmented using the PiNN methodology, adapted with the compartment model realization from case three. In addition, an attempt at further augmenting the methodology is proposed. By introducing additional parameters into the original model and utilizing the PiNN methodology's parameter discovery property, it is posited that the method may find a better solution, and perhaps the learned parameters would yield qualitative predictions.

4.1 Coding and frameworks

Throughout the course of the thesis project, various implementations of the PiNN methodology were created in python for several problems which was of interest to investigate, culminating into the four case-problems. Although inspiration was found in already established implementations, all of the realized implementations for the presented case problems were constructed from the bottom up in Python utilizing the PyTorch framework for neural computations. Although there are existing python packages tailored for PiNNs such as Deepxde and SciAnn (Cuomo et al., 2022), a more bottom up implementation was found to be more flexible and better serving to the goal of developing an understanding of PiNN. Tensorflow and Pytorch are known as excellent deep learning tool-kits, and these packages both support automatic differentiation capabilities for computing gradients as well as integrated functionality for setting up and training neural networks. After several revisions of implementations, PyTorch was ultimately the framework in which the final revisions were created in. Consequently, the specifics of the methodology is constructed under the structure of this framework. However, the overall techniques utilized can be transferred between these frameworks without significant difference in structure and function of the code.[1]

4.2 Overall PiNN implementation structure

The general shared structure of PINN implementations, are similar to the structure of typical neural network training methods, and can be summarized as follows:

Define Neural Network Architecture Using any deep learning framework one defines the network model and sets the layer parameters and activation functions.

Construct a loss function A loss function which takes outputs and derivatives (up to highest degree in D.E.) from the network model, and yields the loss. The loss shall correspond to a difference between the network outputs and a function which satisfies the D.E. In the case where one also wishes to incorporate data, the loss function can be adapted to include loss from predictions over data.

Train the network Using an optimizer algorithm also provided in the framework, the network can be trained to minimize the loss function.

4.3 Case one PiNN as neural solver

4.3.1 Problem statement

Recall that the essential idea behind PiNN, is to construct a loss term for a neural network training loop in such a way that reducing this loss function results in a neural network which satisfies a given differential equation. The best way to illustrate the specific implementation is to present a basic case problem, and correspondingly show the specific components of the implementation. As a simple example, consider the first example case in Lagaris, Likas, and Fotiadis, 1998, with the one variable differential equation:

[1]Code for case-problems can be found at: `https://github.com/lukerlars/PiNN_masterthesis`

$$\frac{d}{dx}\Psi + (x + \frac{1+3x^2}{1+x+x^3}\Psi) = x^3 + 2x + x^2\frac{1+3x^2}{1+x+x^3}. \tag{4.1}$$

This differential equation has an analytic solution:

$$\Psi_a = x^2 + \frac{e^{\frac{-x^2}{2}}}{1+x+x^3}. \tag{4.2}$$

The solution to this differential equation can be found by fitting a neural network.

4.3.2 Case 1 Solution

The goal is then to fit a neural network over the differential equation 4.1, and the net should then become an approximation of 4.2. As stated, one can begin by defining the neural network architecture.

```
class Neural_net(torch.nn.Module):
    def __init__(self):
        super(Neural_net, self).__init__()

        self.layer1 = torch.nn.Linear(1,20)
        self.tanh = torch.nn.Tanh()
        self.layer2 = torch.nn.Linear(20,1)

    def forward(self, x):
        x = self.layer1(x)
        x = self.tanh(x)
        x = self.layer2(x)
        return x
```

Here a small neural net is defined as as subclass of the torch module class. In the initializing method two layers of 20 neurons are created, along with the activation function which in this case is the hyperbolic tangent function. Next, the functionality of the PINN is realized:

```
class case1_pinn():
    def __init__(self, epochs):
        self.model = Neural_net()

        self.domain =
            torch.linspace(0,1,100,requires_grad=True)
            .reshape(-1,1)

        self.optimizer = torch.optim.LBFGS(params =
            self.model.parameters(),
                lr =0.001, max_iter=200)

        self.c0 = 1.0
        self.epochs = epochs


    def de_loss(self):
        def coef(x):
            return (x + (1+3*x**2)/(1+x+x**3))
        def expr(x):
            return x**3 + 2*x +
                x**2*((1+3*x**2)/(1+x+x**3))
        pred = self.model(self.domain)
        dpred = torch.autograd.grad(
                pred, self.domain,
                grad_outputs=
                    torch.ones_like(self.domain),
                retain_graph=True,
                create_graph=True
            )[0]

        z0 = torch.mean((dpred + coef(self.domain)*pred -
            expr(self.domain))**2)

        ic  = (self.c0 - pred[0])**2

        return z0 + ic

    def train(self):
            self.model.train()
            for epoch in range(self.epochs):
                def closure():
                    self.optimizer.zero_grad()
                    loss = self.de_loss()
                    loss.backward()
                    return loss
                self.optimizer.step(closure=closure)
                print(self.de_loss())
```

This class consist of the main functionality for an instance of a PINN model. In the initializing method, a neural network model is created, along with the function domain, a torch optimizer and some constants. The de_loss method (differential equation loss) constitutes the loss function between the neural model and the differential equation. In this method, the "pred" variable constitutes the prediction output over the function domain variable, and correspondingly "dpred" is the differential of this output with respect to domain input, obtained using the auto-differentiation functionality provided by Pytorch. Ultimately, the method returns "z0" and "ic", which respectively denote the calculated loss from the differential equation, and a loss value from the initial condition.

A training instance can be run by

```
c1_inst = case1_pinn(epochs = 30)
c1_inst.train()
```

In this case, the function domain is set to $x \in [0,1]$. Running the model over 30 epochs and making a prediction yields

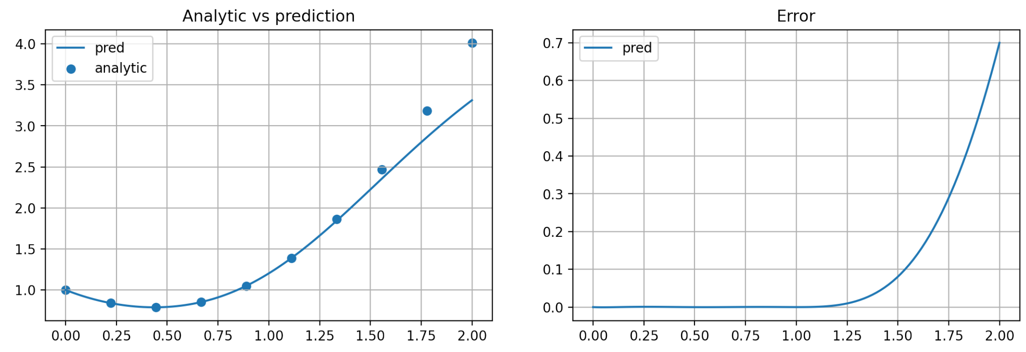


FIGURE 4.1: Case one output and error

From figure 4.1, one can surmise that a relatively accurate solution was found within the training domain. This demonstrates that the methodology of a neural solver works, in a somewhat simple case.

4.4 Case two Lotka Volterra equations

This case demonstrates the regularization property of PiNN when the underlying parameters are known. In addition the discrete time PiNN methodology is demonstrated.

The Lotka Volterra equations (Knolle, 1976), also known as the predator prey model, is a pair of first order differential equations modelling dynamics of biological systems. As the name suggests, these equations model the relationship of two populations in which one population acts as a predator upon the other which acts as prey. This model can generally also be considered as a type of compartment model. The definition of the Lotka Volterra equation can be stated as:

$$\frac{dx}{dt} = \alpha x - \beta xy, \tag{4.3}$$

$$\frac{dy}{dt} = \delta xy - \gamma y. \tag{4.4}$$

Suppose, fixed parameter values: $\alpha = 0.25, \beta = 0.0, \delta = 0.25, \gamma = 2$ The solution to the LV-system will then have a solution in the shape as depicted in figure 4.2. Consider consider the LV case when there is a small sample from a population distribution where parameters are known.

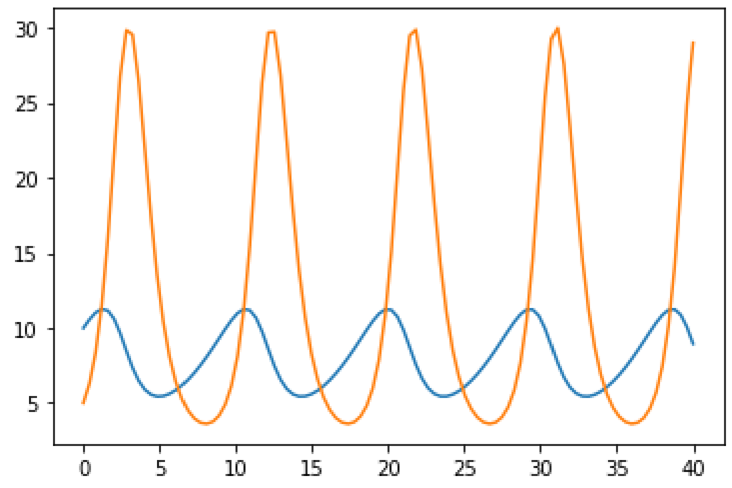


FIGURE 4.2: LV-system true solution

Training a regular neural network over sparse data from this population may yield a fit as depicted in figure 4.3

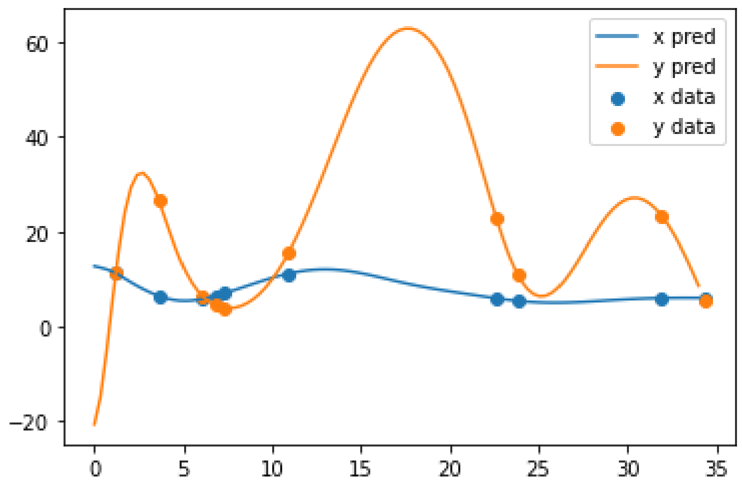


FIGURE 4.3: Predator Prey model, 10 data-points only neural net

By incorporating a PiNN with fixed parameters as a regularizing mechanism, a better fitting model may be found. The same implementation methodology can be used as in case 2, the difference being to keep the underlying parameters fixed. Figure 4.4 show such a solution. The specific implementation used to obtain this solution is presented in case three. This model is trained so that it is trained over a *combined* loss function. By taking sampled data as input, computing loss in relation to this data, one would perhaps obtain a solution akin to figure 4.3. By additionally constructing a loss function which as described in case one and combining this with the loss from the data, results in the plot seen in figure 4.4

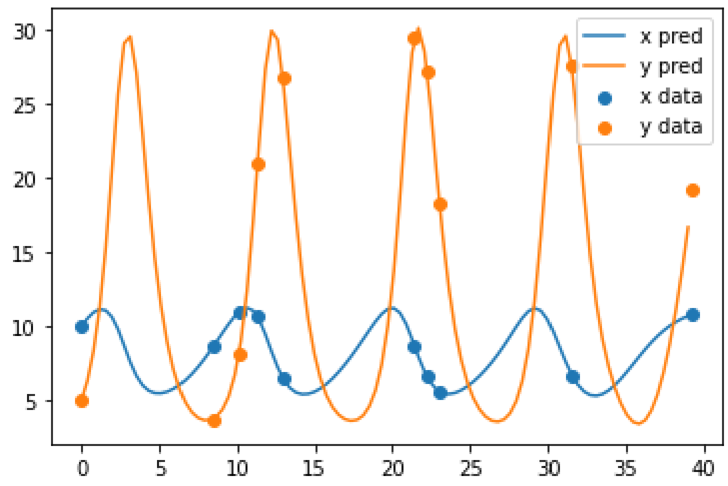


FIGURE 4.4: Predator Prey model PiNN,10 data-points, fixed parameters

This demonstrates the ability of PiNN to act as a regularization mechanism. The output of the PiNN model has higher accuracy with fewer data-points, leveraging the a priori information in the form of a differential equation.

4.4.1 LV discrete time PiNN

Additionally, the same LV model can be realized as a *discrete time physics informed neural network*. Recall the methodology described in section 3.8.3. This methodology can be realized as follows.

Implementation

As described in M. Raissi, P. Perdikaris, and G. Karniadakis, 2019, keeping specific details from their implementation aside, their solution for the discrete time PiNN methodology is performed upon a differential equation where the solution is a two-variable function $u(x,t)$. The solution is found by sampling random points along the x variable, and calculating each time step approximation $u_i^n(x), u_{v+1}^n(x)$ for these points. The LV-system is a simpler case, as the solution space is not isomorphic to the plane but rather two one-dimensional curves. The implementation can then be adapted as follows. Sample the simulated solution at random t-points t_0^i, use then the discrete time PiNN methodology to predict the output point at the next time steps t_1^1.

Consider the following initialization method.

```
    class rk_pp_pinn:
     def __init__(self, dt, xdata,t_dat, q = 100):
         self.model = Duo_net(n_in=2,n_out=q+1)
         self.adam_optimizer = torch.optim.Adam(params = self.model.
                                               parameters(), lr = 0.001)
         butcher_file = np.float32(np.loadtxt('Butcher_IRK100.txt',
                                               ndmin = 2))

         self.IRK_weights = torch.Tensor(np.reshape(butcher_file[0:q**2+
                                               q], (q+1,q)))

         x0s,x1s = xdata
```

```
        self.x0s = torch.Tensor(x0s)
        self.x1s = torch.Tensor(x1s)
        self.t_dat = t_dat
        self.dt = dt
        self.epochs = 10

        self.alpha = 0.25
        self.beta = 0.02
        self.delta = 0.25
        self.gamma = 2
```

Here, the IRK_weights denote the RK-weights from a butcher-table, in this case a 100 stage method is used. The neural network model component is the same as previously depicted, only that this network model consists of two separate networks. This allows for the model to output a solution for both x and y for one t value input. The loss function can be realized as:

```
    def loss(self):
        x1, y1 = self.model(self.x1s)
        x = x1[:,:-1]
        y = y1[:,:-1]


        F0 = self.alpha*x -self.beta*x*y
        F1 = self.delta*x*y -y*self.gamma

        x0 = x1 - self.dt*torch.matmul(F0,self.IRK_weights.T)
        y0 = y1 - self.dt*torch.matmul(F1,self.IRK_weights.T)

        l0 = torch.mean((self.x0s[:,0].reshape(-1,1) - x0)**2)
        l1 = torch.mean((self.x0s[:,1].reshape(-1,1) - y0)**2)

        return l0 + l1
```

Training this model, results in a discrete time PiNN. Figure 4.5, depicts the t_1^i predictions of a time step $dt = 4$.

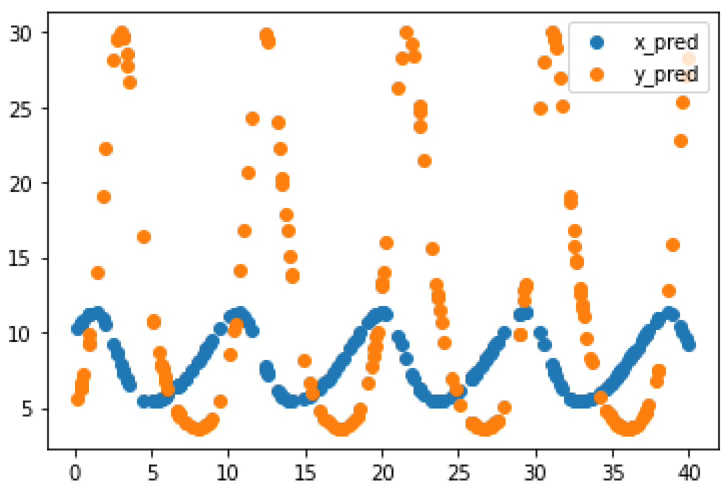


FIGURE 4.5: Discrete Pinn

As seen in figure 4.5, this demonstrates the solution found by the discrete-time PiNN methodology is able to find an accurate approximation to the real solution of the LV-system as depicted in 4.2.

4.5 Case three the SIR model

The field of infectious disease modelling gained an increase in interest following the COVID-19 epidemic. Naturally, there has been some research in applying the PiNN methodology to the SIR and its derived models. Notably, Shaier, Maziar Raissi, and Seshaiyer, 2021 carried over the PiNN methodology and investigated how PiNN's can be utilized for parameter discovery in the infectious disease context. In their analysis, the SIDR and higher order compartment models was adopted. Using simulated data modelling several infectious diseases, the PiNN methodology was used to fit a neural model, coined as a Disease Informed Neural Network (DINN). The study demonstrates the methodology's ability to learn model parameters over cases ranging in number of compartments, and parameter features. The methodology was found to easily find parameters in cases of low dimension and with few parameters, with more difficulty for the over data with many sharp oscillations. Further Schiassi et al., 2021, adapts a variant of the PiNN methodology coined as PINN-TFC (PiNN - Theory of Functional Connections), leveraging the approach of extreme-machine learning, and alternative to the traditional method of training neural networks. This methodology was found to be of high accuracy, in the problem of parameter-discovery applied to SIR, and the higher order variants Susceptible-Exposed-Infectious-Recovered (SEIR) and SEIR-Susceptible (SEIRS). The methodology of Shaier, Maziar Raissi, and Seshaiyer, 2021 can be regarded as a straight forward adaption of the PiNN methodology (M. Raissi, P. Perdikaris, and G. Karniadakis, 2019) for infectious compartment models. From this, the methodology can be adapted into a case study, attempting to recreate the findings of Shaier, Maziar Raissi, and Seshaiyer, 2021, for the smaller SIR model.

4.5.1 Problem statement

In this case the main goal two-fold. The first goal is to estimate a solution to the SIR model using the PiNN methodology and simulated data. The model shall include β and γ as variables, which will be estimated from the data. The effectiveness of PiNN as an inverse-problem parameter discovery system will be investigated. Secondly, PiNN as a regularization mechanism will be considered. That is, in context where sample sizes of underlying data is low, will the addition of the *differential equation loss* aid the model's accuracy, and/or prevent over-fitting?.

4.5.2 PyTorch implementation

The case with the SIR model requires a neural network taking input of dimension one corresponding to time, and three dimensional output (S,I,R). Adapting a methodology similar to the one used in Shaier, Maziar Raissi, and Seshaiyer, 2021 the data-driven solution to the SIR case can be realized. What follows is a suggested outline for performing one such implementation. In this case as well, in order to exemplify how the method can be realized, code snippets of the method implemented with PyTorch is given.

Note that in this case, a four-layer network was adapted, with hyperbolic tangent as activation function. All layers incorporate 20 neurons. This network structure is not necessarily the canonical optimal structure, meaning that aspects of size and activation function may be subject to discussion.

Consider the PiNN class. with the following initialization method.

```
class sir_pinn:
    def __init__(self, epochs, data):
        self.epochs = epochs
        self.model = Neural_net(n_out=3)
        self.domain = torch.linspace(0,int(max(data[0])),100,
                                    requires_grad=True).reshape
                                    (-1,1)

        self.beta_unconstr = torch.nn.Parameter(torch.randn(1))
        self.gamma_unconstr = torch.nn.Parameter(torch.randn(1))

        # Adding gamma and alpha to model trainable variables
        self.model.register_parameter(name='beta', param = self.
                                    beta_unconstr)
        self.model.register_parameter(name='gamma', param = self.
                                    gamma_unconstr)
        # Configuring optimizers
        self.lbfgs_optimizer = torch.optim.LBFGS(params = self.model.
                                    parameters(), lr = 0.001,
                                    max_iter = 500)
        self.adam_optimizer = torch.optim.Adam(params = self.model.
                                    parameters(), lr = 0.0001)

        self.t_dat = torch.Tensor(data[0]).reshape(-1,1)
        self.S_dat = torch.Tensor(data[1])
        self.I_dat = torch.Tensor(data[2])
        self.R_dat = torch.Tensor(data[3])

        #find values for normalization

        self.maxes = {}
        self.mins = {}

        for id,d in enumerate((self.S_dat, self.I_dat, self.R_dat)):
            self.maxes[id] = max(d)
            self.mins[id] = min(d)

        self.N = self.maxes[0]

        #normalize
        self.S_norm = self.normalize(0, self.S_dat)
        self.I_norm = self.normalize(1, self.I_dat)
        self.R_norm = self.normalize(2, self.R_dat)

        self.c0 = torch.tensor([max(self.S_norm), min(self.I_norm), min
                                    (self.R_norm)])
```

This initialization method is set up as to incorporate both data and differential equation loss. As previously defined, a neural network model is set output with one input dimension for time domain, and 3 outputs corresponding to each of the three values considered in the SIR model. The time domain is set to 100 points between 0 and $max(t^{data})$. β and γ parameters are added to the torch-model trainable variables. Two optimizer instances are configured, one using the LBFGS algorithm and another using ADAM. Maximum and minimum values from data is collected and used to normalize the the data. The opted-for normalization strategy is the naive-$[0,1]$ map, achieved by applying: $norm(x) = (x - x_{min})/(x_{max} - x_{min})$. Recall, that in a multi output case such as this, normalization is applied with the intent of stabilizing gradients.

β and γ parameters can be restricted to a specified range utilizing the python property decorator. The decorator ensures each time the parameters are called, torch tanh function is called over the beta and gamma parameters. this restricts these parameters to the range of $[-1,1]$:

```
    @property
    def beta(self):
        return torch.tanh(self.beta_unconstr)

    @property
    def gamma(self):
        return torch.tanh(self.gamma_unconstr)
```

In order to be conveniently able to get the derivative of the neural network output with respect to the input, the automatic-gradient functionality in PyTorch can be wrapped as the following function:

```
    def __wrap_grad(self, f,x):
        return torch.autograd.grad(f,x,
        grad_outputs=torch.ones_like(x),
        retain_graph=True,
        create_graph=True)[0]
```

The loss value from the differential equation component follows as:

```
    def de_loss(self):
      S, I, R = (x.reshape(-1,1) for x in
          torch.unbind(self.model(self.domain), dim =1))

      dsir_dict ={}
      for id, val in zip(('dS', 'dI', 'dR'),(S, I, R)):
          dsir_dict[id] =
              self.__wrap_grad(val.reshape(-1,1),
                  self.domain)

      dS, dI, dR = dsir_dict.values()

      S = self.S_min + (self.S_max - self.S_min) * S
      I = self.I_min + (self.I_max - self.I_min) * I
      R = self.R_min + (self.R_max - self.R_min) * R

      z1 = dS + ((self.beta / self.N) * S * I)  /
          (self.S_max - self.S_min)
      z2 = dI - ((self.beta / self.N) * S * I -
          self.gamma * I) / (self.I_max - self.I_min)
      z3 = dR - (self.gamma * I ) /
          (self.R_max - self.R_min)

      return torch.mean(z1**2) + torch.mean(z2**2)
          + torch.mean(z3**2)
```

This method begins by calling model over the function domain. Calling the model results in an array output of dimensions $(range(t), 3)$, in which each column correspond to the normalized output of the SIR model. Columns are retrieved from this output, and the derivatives of these values are obtained using the gradient wrapper function. In order to calculate the correct differential value, the output values are unnormalized using the maximum and minimum values collected during initialization. The differential loss for each component is then calculated as the *zn* values. Note that the values that correspond to the right hand side of the differential equations are again scaled by the normalizing constants. This is due to the fact that the values dS, dI, dR in the code correspond to the derivative of the *normalized* values. Set $\tilde{S}$ as the normalized S, the derivative of this value $\frac{d}{ds}\tilde{S} = \frac{d}{dS}(S_{min} + S(S_{max} - S_{min})) = S_{max} - S_{min}$. Since the neural network is to predict over normalized values, the error must correspond accordingly. Hence $dS = \beta\frac{SI}{N} \rightarrow d\tilde{S} = (S_{max} - S_{min})^{-1}\beta\frac{SI}{N}$. Finally, the mean square error for all three compartments are yielded.

Next, loss from data is defined rather straight-forwardly:

```
    def data_loss(self):
        S, I, R = torch.unbind(self.model(self.t_dat), dim =1)
        z1 = torch.mean((self.S_hat - S)**2)
        z2 = torch.mean((self.I_hat - I)**2)
        z3 = torch.mean((self.R_hat - R)**2)

        return z1 + z2 + z3

    def combined_loss(self):
        return self.de_loss() + self.data_loss()
```

Note that all of these methods are defined within the PiNN class. With these methods defined, its possible to train the SIR model as a Physics Informed Neural Network. Figure 4.6, illustrates the output of the method for 10 sampled simulation data-points. In this plot, the random sample of simulation data occurs as the scattered dots, and the prediction output of the model is depicted as the connected line plots.

10 Random Sampled Data-Points, 100 Domain Points

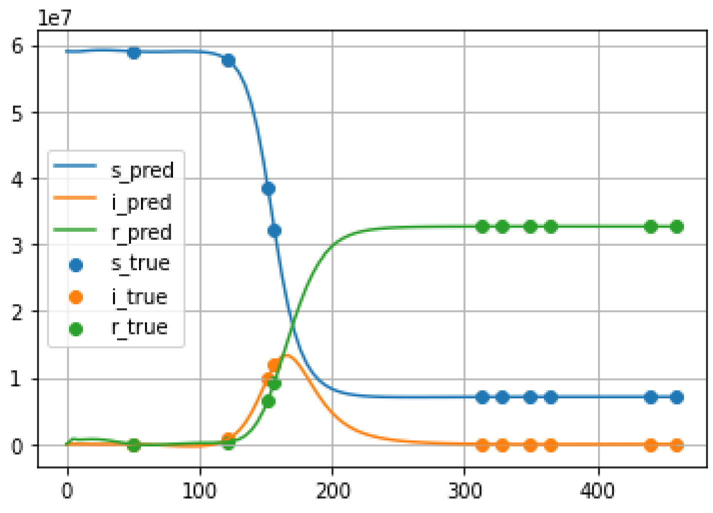


FIGURE 4.6: Results 10 random sampled data-point

4.5.3 Performance testing scheme

In order to gauge the performance of the implemented methodology, a testing scheme was devised. Simple tests can be run over simulated data-points. In python, the

ODE solver from Scipy, utilizing the LSODA algorithm (Petzold, 1983) can be used to generate numerical solutions to the system of equations. These generated points can then be sampled and used in the PiNN model in lieu of real-world data. Case in point, data for different values of gamma and beta parameter can be generated. Tests are run with intent of illustrating how the two loss components from sampled simulation data and differential equation domain-points, affect the overall model performance. The following testing scheme was employed.

Created simulation data-sets

- 5 Noise free simulation data-sets with varying parameters
- 2 Noisy data-sets with different levels of noise

Table 4.1 denotes the noise free simulated data sets. These sets were created with the intention of investigating the methods robustness for varying parameter in the underlying data. Table 4.2 denotes the generated data sets for noisy data.

Set-Name	Parameters
b005g002i001	$\beta = 0.05, \gamma = 0.02, S_0 = 0.99, I_0 = 0.01$
b005g002i050	$\beta = 0.05, \gamma = 0.02, S_0 = 0.50, I_0 = 0.50$
b050g005i050	$\beta = 0.50, \gamma = 0.05, S_0 = 0.50, I_0 = 0.50$
b070g016i001	$\beta = 0.70, \gamma = 0.16, S_0 = 0.99, I_0 = 0.01$
b070g016i020	$\beta = 0.70, \gamma = 0.16, S_0 = 0.80, I_0 = 0.20$

TABLE 4.1: Noise free data-sets with varying parameters

Set name	Parameters
b020-g005-i020-n070	$\beta = 0.2, \gamma = 0.05, I_0 = 0.20, S_0 = 0.80, Noise = 0.7\sigma$
b020-g005-i020-n149	$\beta = 0.2, \gamma = 0.05, I_0 = 0.20, S_0 = 0.80, Noise = 1.4\sigma$

TABLE 4.2: Noisy sets

For this test there are two qualities of interest to be tested. One is the reverse problem of *parameter discovery*. How well the method fits to the underlying data, and to what degree does the method retrieves the correct parameters. The second aspect is *regularization*, in particular, does the inclusion of differential information aid the models predictive power? From these qualities, four types of models were tested. Each generated set represents a round of testing. For rounds 1-5, the interest was to see the performance of the methods over perfect data, and if the performance varies over the underlying parameters. The second sets consist of data with added noise. These test are to investigate the parameter-discovery performance over more realistic data.

Model Name	Properties
100DA100DE	Model with 100 simulated data-points, and 100 domain-points for calculating differential equation loss.
010DA100DE	Model with 10 simulated data-points, and 100 equation domain points.
010DA000DE	Model with only given 10 data points corresponds to a normal neural net with scarce data.
000DA100DE	Model with 100 domain-points only, and fixed parameters. Corresponds to a *neural solver*
010DA1000DE	Model with 10 data points and 1000 domain-points. Used only in noisy data test

TABLE 4.3: Test models, name and description

To get a semblance on how the models improve over training, the results are reported twofold. First, the models are run over 10 epochs of 1000 training steps, with and ADAM optimizer of PyTorch default parameters with learning rate set to 0.0001. Then a second category *Train Until Satisfied (TUS)* is reported. What this entails is the models are run until the metrics have converged, and there is not much improvement of loss values. Each test report the metrics summarized in table 4.4

Metric	Explanation
10 Ep Training MSE	Training loss-value at end of 10th epoch
10 Ep Test MSE	MSE measure from trained model and test set after 10 epochs of training
10 Ep Beta	Estimated β value after 10 epochs of training
10 Ep Gamma	Estimated γ value after 10 epochs of training
TUS Train MSE	Training loss-value at last TUS epoch
TUS Test MSE	MSE measure form trained model and test set after last TUS epoch
TUS Beta	Estimated β after last TUS epoch
TUS Gamma	Estimated γ after last TUS epoch

TABLE 4.4: Test metrics

4.5.4 Performance Testing Results

This section reports and summarizes the results and findings of the testing rounds. Results form rounds 1,2 and 6 are summarized in tables 4.5 4.6, 4.7, and corresponding plots are given in figures 4.7, 4.8 and 4.9. For each of these listed rounds, the performance of each tested model is reported. Due to coincidence in the overall qualitative conclusions from each test, the reports from rounds 3-5 are given concurrently, while considerations from round 7 is given along with the report for round 6. For convenience, the models are referred to by reference names as given in table 4.3.

Round 1

Round 1. N = 1, I0 = 0.01, S0 =0.99, beta = 0.05, gamma = 0.02

Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	0,0093	5,8371E-04	-0,4163	-0,1712
010DA100DE	0,0124	2,9255E-03	-0,4091	-0,1899
010DA000DE	4,96E-05	0,0234	NA	NA
000DA100DE	9,52E-07	0,7304	NA	NA

Test Name	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	4,28E-06	4,1211E-06	0,0493	0,0199
010DA100DE	2,99E-06	1,7839E-04	0,0473	0,0200
010DA000DE	5,76E-06	0,0023	NA	NA
000DA100DE	6,55E-07	0,7568	NA	NA

TABLE 4.5: Round 1 results

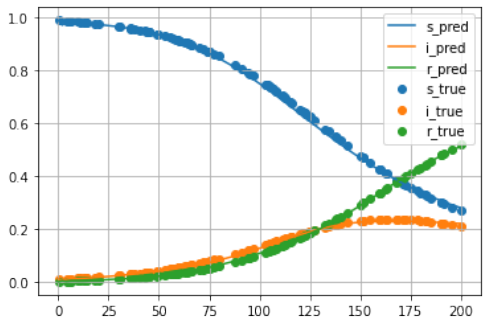
The results from round one is summarized in table 4.5, plots of the estimated solutions are illustrated in 4.7.

100DA100DE The model consisting of 100 data and 100 differential domain points performs somewhat well as can be seen in the plots, and observed from the test MSE. Notice that the discovered parameter is relatively close to actual value, with a relative error of 1.4% for β and 0.5% for γ at TUS. Also its worth noting that the first 10 epochs fails to discover appropriate parameters even though the model can be considered accurate relative to the underlying sample model as can be seen from the test MSE score. Test error score is also the lowest for all of the models at TUS.

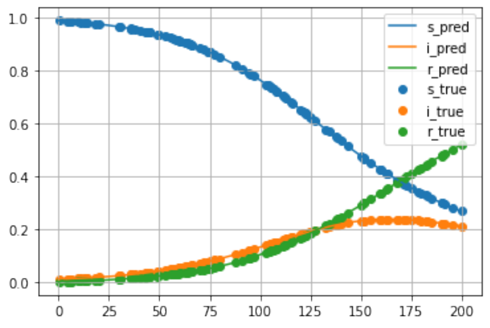
010DA100DE The model with fewer sampled data-points performs similarly to the larger model when it comes to parameter discovery. In the same manner, it fails to predict accurate parameters after 10 epochs, but comes reasonably close at TUS with a relative error for β at 5%, and 0% for γ. This model also suffers a lower test accuracy, than the larger, both at 10 epochs and at TUS. There are noticeable artefacts of instability around the initial conditions as is visible from observing the plots in 4.7. These instabilities seemingly gets better with training as evident from the TUS run on the same model, however the effect is still present.

010DA000DE As hypothesized, the predictive properties of the pure data model performed worse than the models including differential loss. In detail, scoring a lower higher (lower is better) test-set MSE score than both preceding models. This model performs seemingly well, with some evident instabilities in the initial condition. Also worthy of note is the test set MSE score for the 10 data-points-only model and the 10-data-points-100-DE-points model at TUS which is the same score as 010DA100DE, the few-data-points with differential loss model. This suggest in this case that whatever regularization may be present, it has small effect on a random sample test.

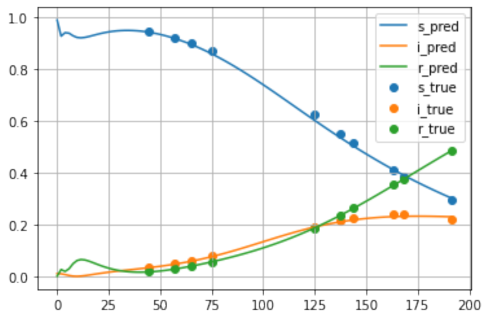
000DA100DE Lastly, one can observe that the neural solver model was not able to find an appropriate solution. Observe from the plots in figure 4.7, the solution found for the model seems to have set the prediction for I and R straight to a constant value of zero, for almost all equation domain points. Hence, the solution technically satisfies the SIR equation system 3.75 - 3.9.



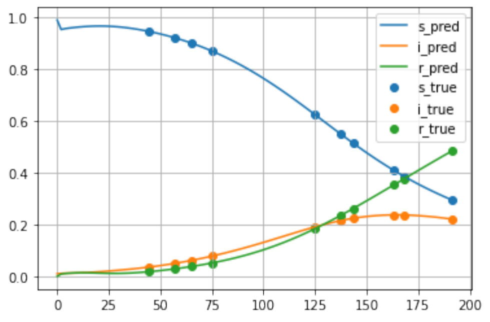
100DA100DE 10 Epochs



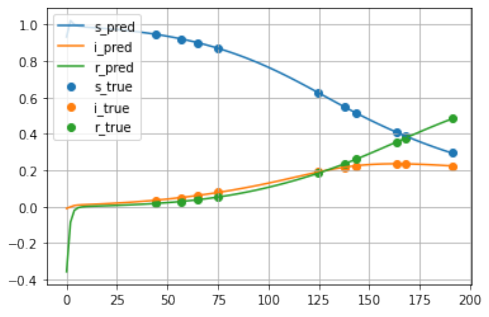
100DA100DE TUS



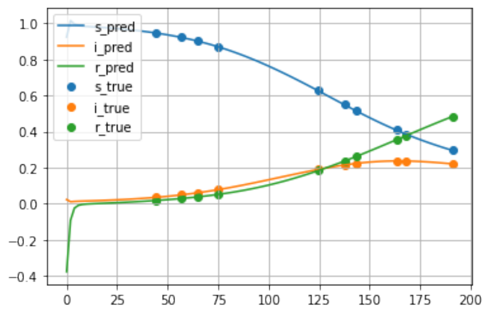
DA010DE100 10 Epochs



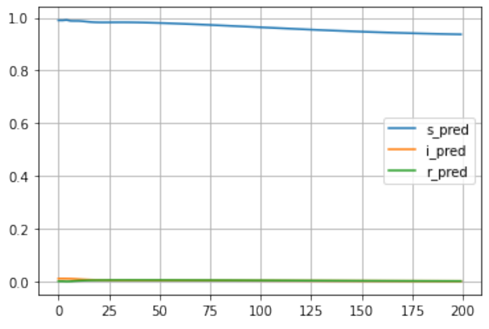
010DA100DE TUS



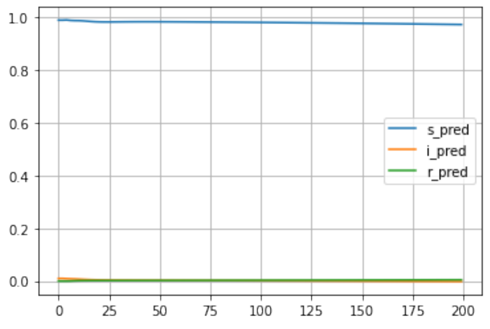
010DA000DE 10 Epochs



010DA000DE TUS



000DA100DE 10 Epochs



000DA100DE TUS

FIGURE 4.7: Round 1 Plots

Round 2

The results from round 2 is summarized in table 4.6, and with plots found in figure 4.7. For this round, the parameters for the underlying simulation data is the same as in round 1, the difference being the initial conditions whereas in round 1 $I_0 = 0.01$, while in round 2 $I_0 = 0.5$ ($N = 1$ for all tests).

Round 2. N = 1, I0 = 0.5, S0 =0.5, beta = 0.05, gamma = 0.02				
Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	5,9907e-06	3,5543e-06	0,2166	0,0199
010DA100DE	5007,13	361,52	0,0005	0,0157
010DA000DE	1,090e-05	536,449	NA	NA
000DA100DE	6,7332-05	0,4522	NA	NA
	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	4,9905E-07	4,2878E-07	0,2241	0,0200
010DA100DE	555,27	46,7371	5,6652e-06	0,0234
010DA000DE	6,9807e-07	534,72	NA	NA
000DA100DE	4,7870-08	0,0421	NA	NA

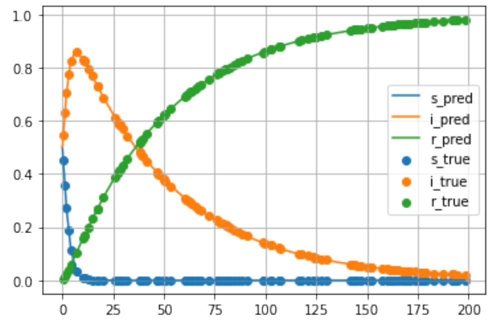
TABLE 4.6: Round 2 results

100DA100DE The 100 data-points 100 differential domain points model performs worse than in the previous case when it comes to parameter discovery. As evident in table 4.6, although the learned γ parameter is precise for both 10 epochs and TUS, the discovered β parameter deviate largely from the true underlying values of the simulation data, with a relative error of 348% at TUS, which is safe to say, is far form the optimal value. Notice also, this prediction slightly worsens upon more epochs of training as the TUS values if farther away from the true β. By observing the plots 4.8, more can be said about the performances of the models. The 100DA100DE model follows the solution exactly, which is good to keep in mind while considering the other plots.

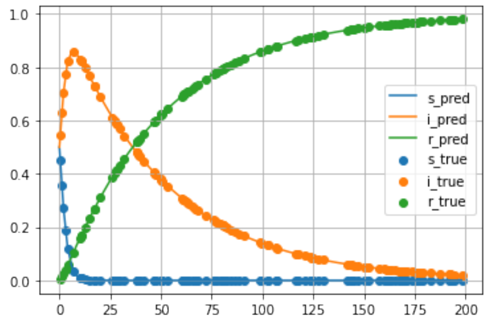
010DA100DE Naturally the model with 10 data-points and 100 domain points also struggle to find the correct parameters, where in this particular case the estimated β parameter approaches 0 upon further training. Again, the pure data models test-set scores at TUS are the same as for the 10 data-point model including differential loss. As with round 1, there is evidence of irregularities in the D010DE100 model. Notice that this model sharply overshoots the peak I value right after t_0. Also notice, the fairly irregular shape of the output. This irregular shape may be due to the model setting the β parameter to zero. This will effectively free up the differential loss component associated with this parameter, and will not inhibit the irregular output shape. One thing about this test to be noted is the sensitivity to the sampled underlying data. Running the same test with a different samples set may in some cases perform better, and (some cases worse). This is especially affected by the degree of sample points falling within critical areas of the true underlying solution. The model defining areas being the areas of rapid increase and then decrease occurring at about time step $t \in [0, 20]$. A small random sample such as 10 data-points is unlikely to fall within this critical area such as the overall shape of the underlying model is clearly defined.

010DA000DE Although in this test, the PiNN models seem to experience some problem, a pure data model would fare worse as evident from the test-set MSE. Even though from the plots this model seems not to be a bad fit, it scores relatively high on the test set indicating a less than stellar fit. Since this pure data model (of few points) is trained upon the same set as the 010DA100DE model, this supports the theory of the few-data-point PiNN model model being sensitive to sample locations. Also, since both these models are arguably inaccurate, how come the 010DA100DE model fair a bit better on the test metrics (especially at TUS)? Observe again the plots of the model, and regard the 100DA100DE as the closest to the true plot. 010DA000DE, that is a pure data model does not satisfy the initial conditions. Even though 010DA100DE seems rather inaccurate, it seems closer to satisfy the IC.

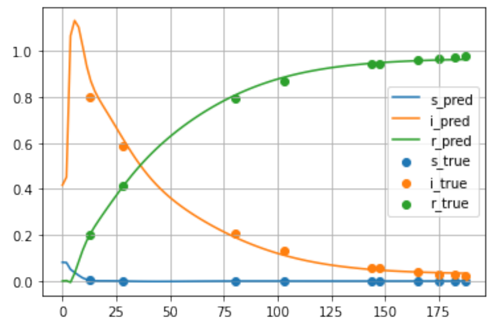
000DA100DE In this round model 000DA100DE (neural solver) Is able to find a solution. By judging the plots 4.8, the overall shape of the solution is not perfect. By this, consider the shape outlined by the 100DA100DE model as the true solution. Notice that the neural solver undershoots the peak in the prediction of I, and turns less sharp in I and R than the true solution. This solution found at TUS also does not seem to improve by applying even more training steps. However, still the test MSE yields a score of 0.0421 at TUS, which is the second best score of this round.



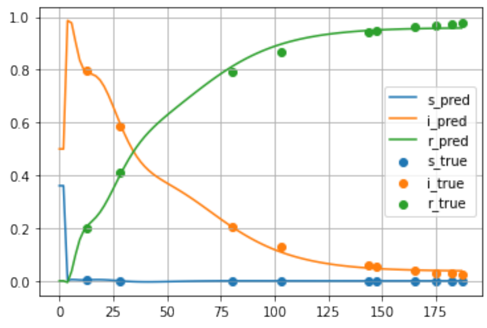
100DA100DE 10 Epochs



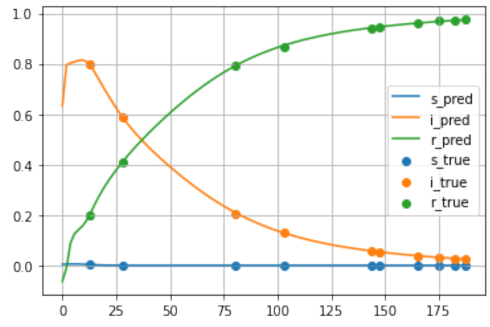
100DA100DE TUS



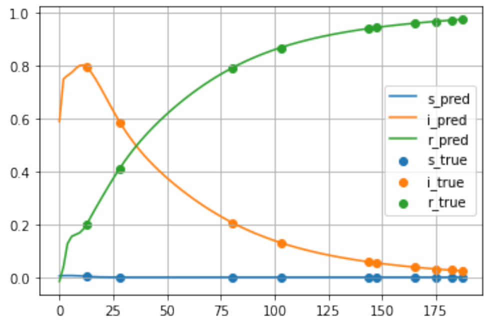
010DA100DE 10 Epochs



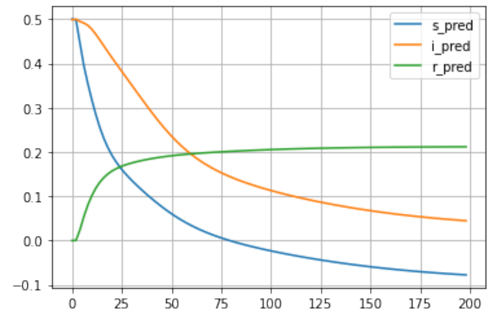
010DA100DE TUS



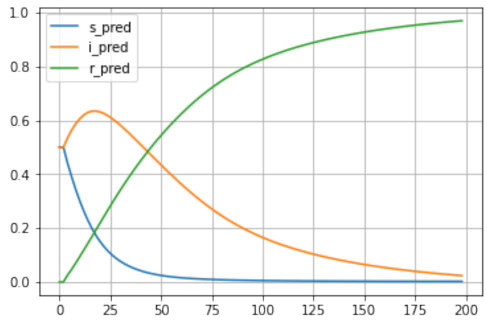
010DA000DE 10 Epochs



010DA000DE TUS



000DA100DE 10 Epochs



000DA100DE TUS

FIGURE 4.8: Round 2 Plots

Rounds 3 - 5

Refer to appendix A for the plots and result tables of tests 3-5. The results for rounds 3-5 are reported in tables A.1 - A.3. The performance of the models can be summarized accordingly.

100DA100DE When it comes to this model, the main focus is the performance of parameter estimation, overall seem to perform surprisingly poor. For tests 3-5 the model most accurate when it comes to finding the γ parameter. With emphasis on the TUS values which ideally should have converged to the correct parameter, the model seems sporadic in it's predictions. The model seems to be having particular issue finding the correct beta parameter. The closest in this case is in round 4, where the correct beta parameter was 0.7, and the predicted beta at TUS was found to be 0.588. The largest deviation was in the following round 5 with the same underlying beta and a prediction of 0.298. The gamma prediction is more accurate but still some degrees of as is the case in round 5 with $\gamma_{\text{true}} = 0.16$

010DA100DE Since this model is trained on significantly fewer data-points than the previous which had difficulty finding the correct underlying parameters, it's natural for this model to also struggle at the same task. Incidentally, this turns out to be the case for all of the tests. Overall the model struggles to learn the correct β parameters, but generally gets reasonably close to the correct γ parameters, not lagging much behind the 100DA100DE model at TUS. The regularization property seems to be present and yield some increase in generality of the output.

010DA00DE Performance for this model is mainly judged by test-set MSE. Considering this metric over rounds 3-5 large variations are observable. In round 3, the model performs on par with the 010DA100DE model, while round 4,5 performance is lower. Note that this model generally has the highest test-MSE scores all around, indicating that there indeed is a beneficial regularization effect present for the 010DE100DA model.

000DA100DE Since this model corresponds to a neural solver, it's accuracy largely depend upon it being able to solve the compartment system. Observing the test MSE from round 3-5, it seems as the method was able to find a solution for round 3 and 5, since these score range < 1 at TUS. Still, tests 3,5 shows the solution not reaching the same accuracy levels as the other models, indicating the same case as in round 2, as this model reaching some solution resembling the true solution but falling short in magnitude. In particular, this model struggles for the test where the relative magnitude of the initial condition I_0 value is small.

Rounds 6 and 7 noisy data

Table 4.7 shows results tables for rounds 6, while figure 4.9 depict the plots. Values an plots for round 7 can be found in appendix A. These tests utilize noisy data, having the same underlying simulated data with different levels of noise. Round 6 incorporates $0.7\sigma_{\text{noise free}}$ and round 7 1.4σ. Figure 4.9 depicts the output plots for round 6.

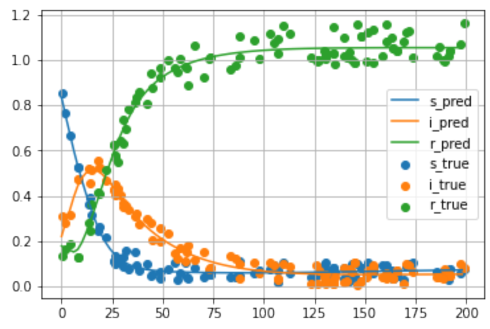
Round 6,$N = 1, I_0 = 0.20, S_0 = 0.80, \beta = 0.20, \gamma = 0.05, Noise = 0.7\sigma$				
Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	0,00682	0,00681	-0,0524	-0,0178
010DA100DE	0,01265	0,05119	0,3996	0,295
010DE1000DE	0,00331	0,01757	0,084	0,0379
	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	0,00559	0,0061	0,1356	0,0431
010DA100DE	0,00293	0,02292	0,0717	0,0404
010DE1000DE	0,00296	0,02078	0,0841	0,0383

TABLE 4.7: Round 6 results

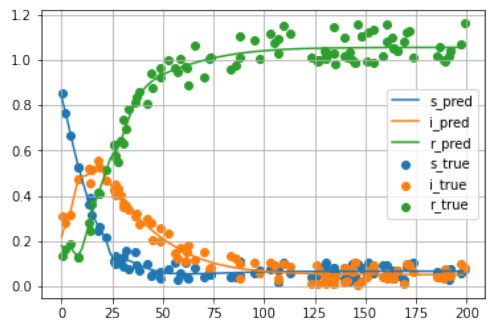
100DA100DE As discussed for the previous rounds, the parameter discovery performance was found to be surprisingly disappointing even for a model trained over noise free data. Then as expected, adding noise to the simulation data further inhibits the models parameter discovery capacity. The test also show that amount of added noise can worsen this effect. Consider the TUS run discovered parameters for test 6 and 7. In round 6, a case can be made for the model learning somewhat close to the real parameters, predicting both γ and β with a respective error ratio of 32% and 14%. In round 7 the parameter prediction accuracy i severely reduced for the β parameter with a relative error ratio of 56%, and for γ 37%.

010DA100DE The parameter discovery ability for the few data-points model can be said to be sensitive to noise. This is most seen in the β parameter, where the model preforms worse than the model trained over 100 noisy data-points. However, when considering the evidently poor parameter discovery properties of this model on perfect data as given in rounds 1-5, it was not expected to do well in this test. From the plots, it actually looks as if more training steps has a detrimental effect as the 10 epochs run looks like a better generalized fit over the TUS run.

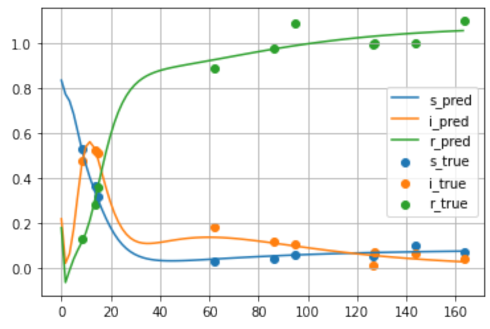
010DE1000DE This model is only used in these test-rounds 6-7, in order to see if increasing the domain-points on which differential equation loss is calculated has an effect on the model accuracy in the case with noisy data. As can be seen from the tables and the plots, this does not seem to be the case. Although from considering the plots, especially the 10 epochs run seems like a better generalized fit than its 100DE-points counterpart, however, at TUS run the plot is virtually identical, meaning that this addition of DE points had in this case no apparent benefit.



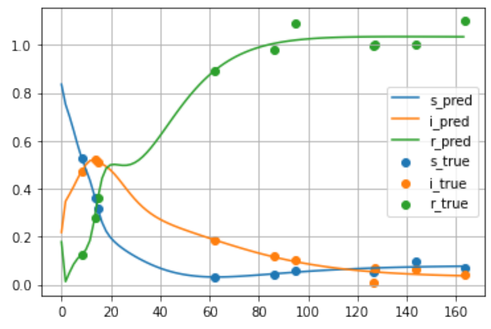
100DA100DE 10 Epochs



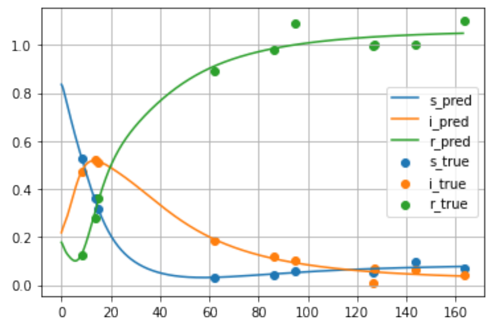
100DA100DE TUS



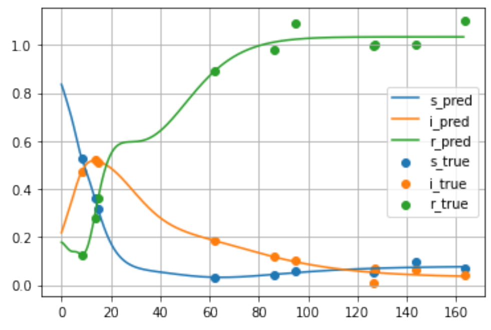
010DA100DE 10 Epochs



010DA100DE TUS



010DA1000DE 10 Epochs



010DA1000DE TUS

FIGURE 4.9: Round 6 Plots

4.5.5 Case 2 testing discussion

The testing seems to show variable performance results. In particular, the findings of accuracy in regards to parameter discovery seem less optimal as the results found by Shaier, Maziar Raissi, and Seshaiyer, 2021. In particular considering the latter's use of admittedly more advanced models, the results from the tests seem spurious. Overall the performance of the tests indicate the following. The parameter discovery component of the methodology works when conditions are ideal and when there is ample sampled data points. This seem to be the case when considering the results of round 1, however the convergence towards the true parameters seems somewhat lackluster. When there is less data available, the PiNN models struggle more to find the correct parameters, but does not lag too far behind the model instance were more sampled point are available. Also, in the sparse data case the performance in regards to regularization can be noted. The tests indicate that there are present regularization effects, which gives some benefit to test score over a naively trained model. Logically, this effect also seem to work better if the learned parameters are closer to the actual values.

As the main goal in part was to gauge the overall performance of the methodology, it is important to note potential factors, independent of the methodology, originating from the specific implementation, which may have influenced the results. Case in point, The size of the neural net opted for in the model may have an impact of the overall accuracy. However testing of the implementations, during their construction, indicated that overall network size and shape seemed to have little impact on accuracy. It may also seem superfluous to note the potential of human error, however, this is an ever-present source of error for most things in general. Indeed, as this thesis is written, errors in the specific implementation has been found and corrected. As it stands however, the results seem to reflect the overall performance of the methodology to the degree of which the author has understood it. Meaning that what is reported here, is judged to be representative.

Some criticism may also be directed towards the testing method which was opted for. In part, one may make the argument that the test type of TUS (Train Until Satisfied) is not an entirely rigorous concept, and may be a source of error. The reasoning for defining this as a test was the observation that the models converged towards values (minimum training loss, learned parameters) at different rates. When training machine learning models one can incorporate a callback function which tracks the training loss, and halt the training if either the loss value does not improve after a set of epochs or if it reaches a certain level of accuracy. However, the practicality of such an approach did not seem appropriator in this setting. Also, with the goal of gauging the general performance of the methodology, the evident inaccuracies the TUS testing concept would introduce was deemed as negligible. Finally, an additional relevant critique is the fact that the purely data trained models have not had any standard regularization technique applied to them. This means that overall the regularization effects of the PiNN model is compared against a *naive* model rather than a regularized model.

SIR neural solver difficulties

The neural solver component of the implementation is an aspect not addressed elsewhere. As can be observed from test-round 1, and as have been noted over the implementation process, a neural solver implementation of the SIR model seem particularity difficult to implement. An observation has been that in some cases a neural solver implementation utilizing only the DE-loss component with boundary conditions sometimes finds a solution which is seemingly close to the real solution, while other times, the model sets the I component of the model as a constant 0. A suggested reason for this effect is as follows. Recall the SIR model differential components as described in 3.75 -3.9, one may notice that the I term is present in all of the differential components. From the perspective of an optimizer algorithm, this becomes an opportunity. Figuratively, it wants the quickest way to set the loss value as close to zero as possible. When the neural solver implementation calculates the loss value for the SIR model the following occurs: The neural net yields a prediction of the compartment values S,I and R, as well as their derivatives dS, dI, dR with respect to the time domain input (t). For each compartment, the loss value is calculated as the difference between the yielded derivative and the right hand side of equations 3.75 -3.9. As an example $Loss_S = dS - \frac{\beta SI}{N}$. Additionally, the initial condition (boundary condition) can be added to the loss function as the squared difference between the neural net output at t_0, and $[S_0, I_0, R_0]$. In some cases this is sufficient for the neural solver to be able to find a solution. In other cases, the optimizer opts for setting the I component to a constant of zero. Since the I component is a constant zero it means that all of the right hand side expression of 3.75-3.9 also becomes zero. In the perspective of the optimizer, this is a valid solution. The resulting output may then look like what is depicted in figure 4.10

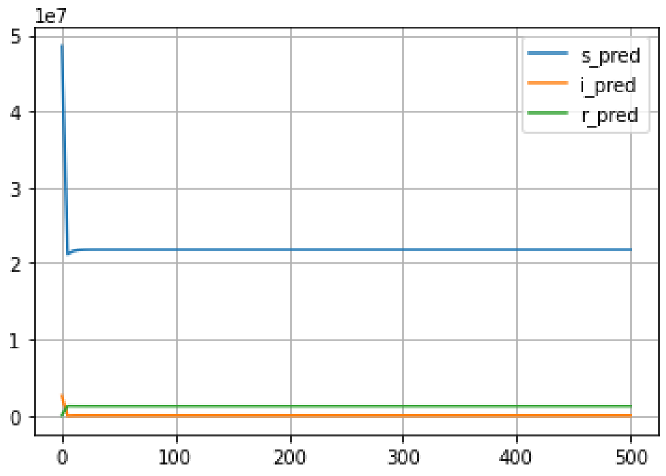


FIGURE 4.10: Neural solver SIR difficulties

Note this may results in large jumps in the output the next domain-point after the initial condition. This value will be nonzero and added to the loss. However, then the training steps has reached this point it is already to late. One way to attempt to mitigate this effect is to add more points to the initial condition. In some cases this works, as an example, the neural solver implementation in round 2, has the first two domain points assigned as IC, and is able to come up with a solution. Having only one point in this case results something more like what is depicted in figure 4.10. Also, as illustrated in round 1, this solver also has additional IC points, but since the initial condition is such a low relative value in I (1% of N), the addition of more IC point has less impact and the solver still resorts to reducing I to zero.

4.6 Biomedical compartment model: test bolus injection

In a biomedical context, the circulatory system may be modeled as a more or less complex compartment model. With this in mind, an observation was made in a project relating to modelling this system and adjoining data: A pure data-driven model performed worse than a first principles simulated model for prediction of a certain characteristic. Notably as well, a metric which was to be estimated seemed to lie somewhere in-between a pure data driven estimate and the simulated model. The question then which naturally arose was if a physics informed neural network would fare any better than the two methods?

4.6.1 Circulatory compartment models

Perhaps a more detailed description of the simulated model is needed. Adapting the methods outlined in Hiorth, 2022, and Bae, Heiken, and Brink, 1998, consider figure 4.11 illustrating the circulatory system on the left and a compartment model scheme on the right. In this illustration the compartments represent different components connected by the circulatory system. A compartment model of this kind models the system as a a flow system, where the *mass conservation* law is upheld for in-flowing and out-flowing fluid.In its simplest form with a presupposed constant volumetric flow-rate Q over a set of compartments, the input and output flow velocity is only dependent on compartment volume. In radiology, *bolus injection* is a technique to investigate the diffusion of contrast agent within blood-flow. The procedure consist of injecting *contrast agent* at an initial inlet in the system, and the *enhancement* HU (concentration of contrast agent) is recorded at following terminals in the system.

Blood flow can be modelled as a compartment model, where each compartment represents a component of the circulatory system with varying volume. Let c_i denote contrast concentration in compartment i. The concentration of contrast liquid for this compartment can then be modelled as:

$$\frac{dc_i}{dt} = \frac{Q_{in}c_{i-1} - Q_{out}c_i}{v_i]} = \frac{Q(c_{i-1} - c_i)}{v_i}. \tag{4.5}$$

Figure 4.11 (Kyongtae, 2010) illustrates a reference representation of the circulatory system (left) as a large compartment model (right)

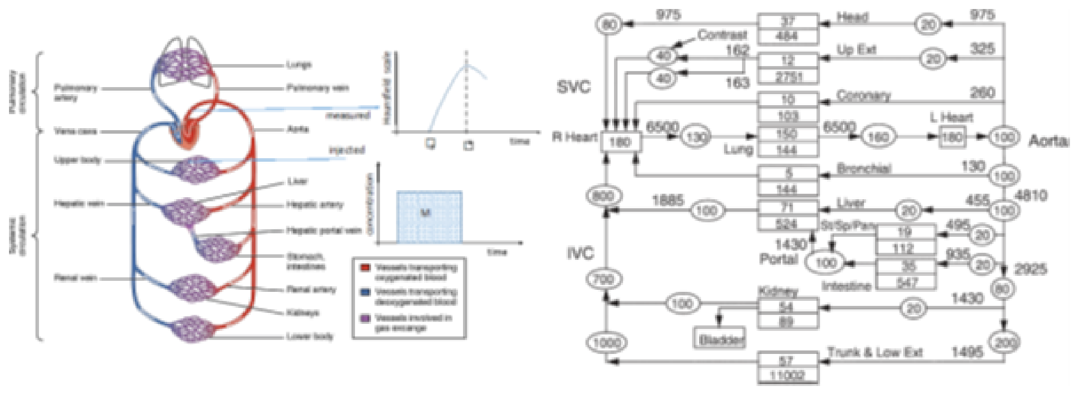


FIGURE 4.11: Circulatory system and compartment model

4.6.2 Model subsystem

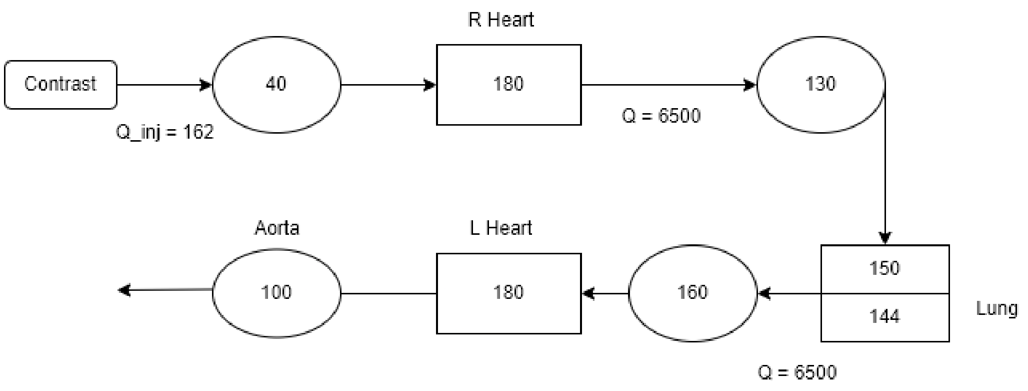


FIGURE 4.12: Circulatory Subsystem

Consider the subsystem 4.12, of the larger circulatory system as given in 4.11. This subsystem models the blood-flow path between the heart and the lungs, with a contrast agent injection at the inlet of the Vena cava before the right-half of the heart.

The subsystem is modeled as 8 compartments including the lungs and the heart. A procedure of contrast injection was performed on a set of patients. In this test, 20 mL of contrast agent (350 mg/L) was injected at an injection rate of 6 mL/s, and the contrast enhancement (HU) at the Aorta was recorded with MR and CT scans. In addition, the patients *cardiac output* CO was also recorded.

4.6.3 Test bolus simulation

With the intent of determining a patients CO from predetermining factors: weight, height and gender, a simulation for this subsystem can be performed. In particular, the test injection protocol can be simulated over a reference model so that one has a reference prediction of enhancement in the last compartment. The reference model is given as in Hiorth, 2022 and (Bae, Heiken, and Brink, 1998), and figure 4.11, which models a 70 kg male, with approximately 5 liters of blood. In this reference model, CO is set to 6500 l/min, and the volume of compartments are given as in figure 4.12. For a particular patient, the reference model can be adjusted according to patients blood-volume. Provided a patients height, weight, and sex, blood volume can be estimated with the following formulas:

$$BV_{male} = 665.13H^{0.725}W^{0.4245} - 1229, \tag{4.6}$$
$$BV_{female} = 698.95H^{0.725}W^{0.4245} - 1954. \tag{4.7}$$

Once a patients blood volume is found, the relative volume of compartments can be adjusted from the reference model. The CO can be determined by a value matching scheme. Suppose one have the recorded patient enhancement findings in the last compartment, and in particular the *peak enhancement*. By simulating the test bolus injection of a patient-adjusted model, and varying the CO in the simulation, the CO should be determined by matching the time-of-peak in contrast agent enhancement in the simulation with the patient record. In more detail, the simulation can be constructed as follows. Suppose 8 compartments with volumes $V_1, ..., V_8$. An injection is given of 20 ml of contrast agent at 350 mg/L at a rate of 6ml/s, for 4 seconds, with a saline chaser for 8 seconds. See figure 4.13, illustrating the injection signature over a simulation time of 100 seconds (note that this is not to scale).

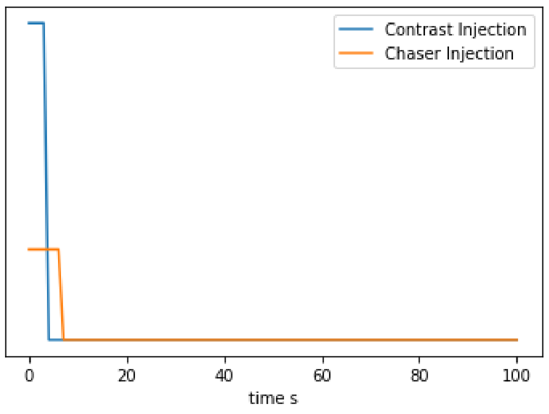


FIGURE 4.13: Injection of contrast agent and saline chaser 100 seconds (not to scale)

Set a constant CO as Q. Set $c_{ca}(t)$ to denote the amount of contrast agent in the initial injection (with signature as in figure 4.13), Q_{inj} as the injection flow, $c_{cr}(t)$ as the concentration of saline chaser (also with time signature as in figure 4.13), PS as a diffusion constant between the two lung compartments, and lastly c_i as the concentration of contrast agent in compartment i. The flow of contrast agent through the system can now be modelled, with the differential equation system:

$$\frac{dc_1}{dt} = \frac{c_{ca} - (Q_{inj} + c_{cr})c_1}{v_1}, \tag{4.8}$$

$$\frac{dc_2}{dt} = \frac{(Q_{inj} + c_{cr})c_1 - QC_3}{v_2}, \tag{4.9}$$

$$\frac{dc_3}{dt} = \frac{Q(c_2 - c_3)}{v_3}, \tag{4.10}$$

$$\frac{dc_4}{dt} = \frac{Q(c_3 - c_4 - PS(c_4 - c_5))}{v_4}, \tag{4.11}$$

$$\frac{dc_5}{dt} = \frac{PS \cdot Q(c_4 - c_5)}{v_5}, \tag{4.12}$$

$$\frac{dc_6}{dt} = \frac{Q(c_4 - c_6)}{v_6}, \tag{4.13}$$

$$\frac{dc_7}{dt} = \frac{Q(c_6 - c_7)}{v_7}, \tag{4.14}$$

$$\frac{dc_7}{dt} = \frac{Q(c_7 - c_8)}{v_8}. \tag{4.15}$$

A plot of the reference solution to this system is given in figure 4.14 (excluding the injection compartment c_1).

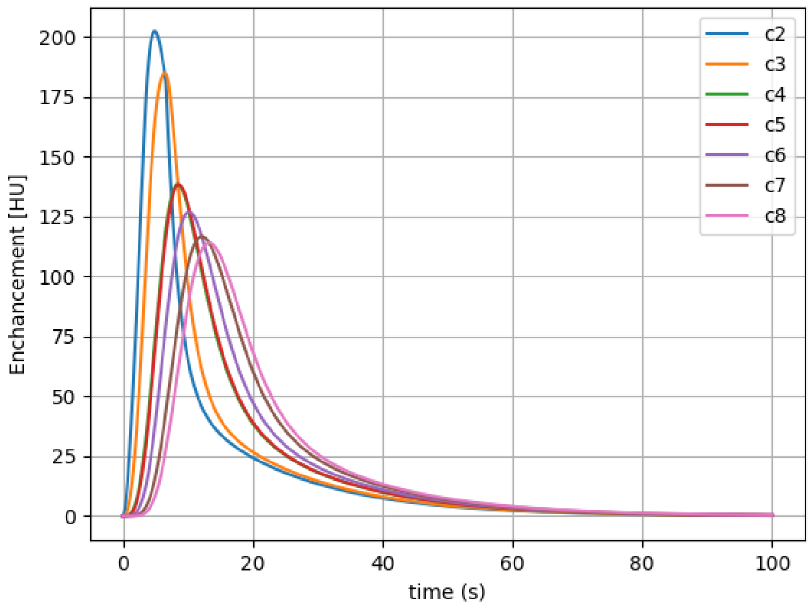


FIGURE 4.14: Reference enhancement $c_2, ..., c_8$ (c_1 injection compartment not shown)

Now, varying Q will mainly change the *time to peak* TTP. By matching the TTP of c_8 with data, one will obtain an estimate of the the cardiac output CO. Figure 4.15 depicts how this fit typically looks with patient data (with hidden figures as to protect patient information).

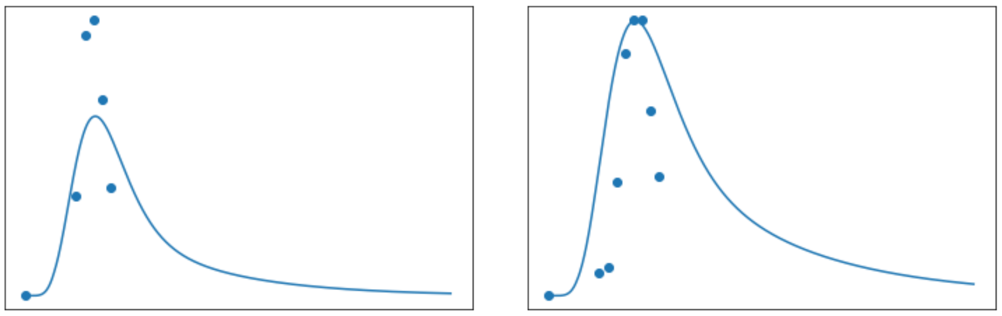


FIGURE 4.15: Two examples of typical TTP fit, (hidden numbers)

4.6.4 Gamma variate

A second methodology outlined in Hiorth, 2022 is the fitting over a *gamma variate* function, on patient data. Madsen, 1992 Shows that a tracer dilution curve can be modelled as a gamma variate function.

$$f(t;\alpha,\beta) = \frac{t^{\alpha}e^{-\frac{t}{\beta}}}{\beta^{\alpha+1}\Gamma(\alpha+1)}. \tag{4.16}$$

Where the Γ function:

$$\Gamma(\alpha+1) = \int_0^{\infty} x^{\alpha}e^{-x}dx. \tag{4.17}$$

The patient CO can be estimated as the area under the tracer dilution curve. Where a dilution curve follows some scaled gamma variate $A \cdot f(t;\alpha,\beta)$. Since the non-scaled gamma variate is a probability distribution (integrates to 1), this area will be A

4.6.5 Problem formulation

The test bolus simulation scheme turn out to not have optimal predictive power. As illustrated in figures 4.15, the fitted solution does not necessarily overlap neatly with patient data. The simulation predicted CO is reported to be lacking in accuracy. Sidestepping from the necessary critique of the method itself, one might consider various ways to improve upon the method. Due to the physical modelling nature of the methodology one can ask the question if there is some way to us PiNN to improve upon the model? Specifically there is two measurable goals. The first goal is to adopt the test bolus simulation model and PiNN in order to find an improved estimate of patient cardiac output CO (Q). The second goal is to see if the same model can be adapted to better predict the enhancement signature for a patient.

How to apply PiNN?

To be able improve upon the test bolus compartment model using PiNN, facets of the method in which PiNN can be applied and necessarily have a correctional influence. If the case is that the differential model is not appropriate or under specified in regard to the underlying process, then applying a PiNN will not have a beneficial effect. With this in mind consider again the bolus simulation compartment model. By varying the Q values, the effect on the output is mostly translation of TTP, where an increase in Q means a leftward translation of the curve. This means that in order to have a better prediction of Q the TTP of the improved model must shift away from the peak in patient data, which the original model is aligned to. Additionally, as can be seen in figure 4.15, typically the patient data distribution tend to be higher and narrower than the output of the original model. With these two problem to be solved, a hypothesized mending strategy is to *add* parameters $\boldsymbol{\lambda}$ to the original model, which can adjust the shape and position of the output. By adding these parameters strategically (well-posed), hopefully, the parameter fitting property of the PiNN methodology can find parameter values which improve upon either or both of the CO and output signature.

4.6.6 Strategies and implementation

Strategy 1

Two strategies are posited. The first strategy is to mirror SIR case and place a PiNN over all of the compartments of model 4.8 -4.8. As discussed, parameters $\boldsymbol{\lambda}$ will be added to the model in order to shape output. First consider the method without added parameters. Consider a *neural solver* solution to the equation system 4.8 - 4.15. A neural solver solution would work by means of a trial solution $NN(t) = \Psi(t) \to \mathbf{R}^8$ which initially is a random continuous mapping $t \to \mathbf{R}^8$. The loss value for this trial solution is computed from the degree of which $\Psi(t)$ satisfies the equation system. Now, an important aspect of the loss calculation must be considered. This model is *sequential* meaning that the way it would converge towards the correct solution ideally, is to first fit the first compartment, then use this solution to fit the next, and so on. Figure 3.4 illustrates an attempt of a pure neural solver over essentially seven free compartments. The solution seems correct for compartments $1-3$, and the following compartments, not so much.

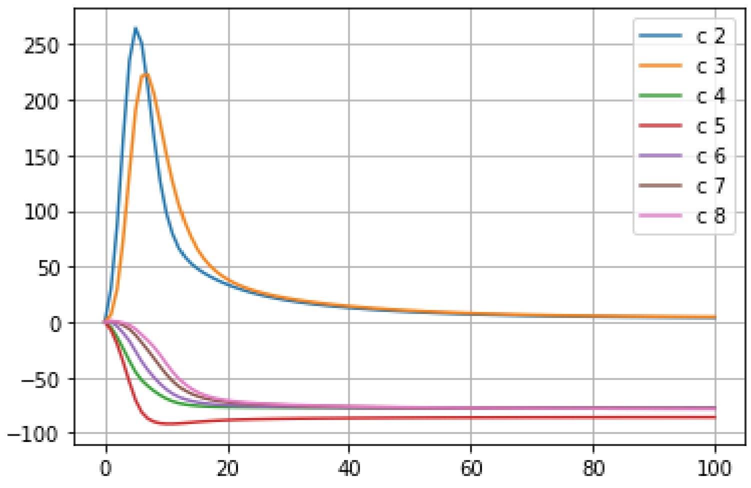


FIGURE 4.16: 8 compartments neural solver attempt

Recall that patient data should correspond with the output of compartment 8. Figure 4.17 illustrates the resulting fit by incorporating loss from patient data.

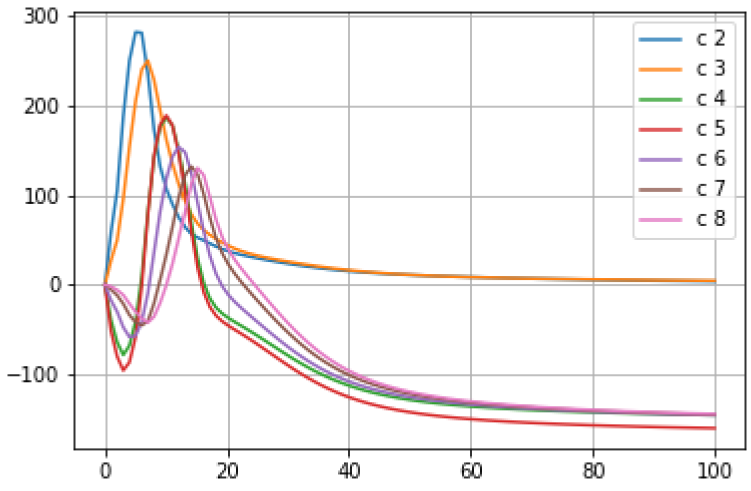


FIGURE 4.17: With data loss from compartment 8

As is obvious from observing the plots, the solution found by the method is also not correct. However, some interesting result can be gleaned. Notably observe that the enhancement peaks for the compartments are now seemingly closer to that of the simulated model (figure 4.14). The problem here might not necessarily lie in the methodology but rather the shape and scarcity of patient data. As an attempt to mend this, an adjustment to the method is suggested. As described in section 4.6.4, the enhancement curve can be modeled as a scaled gamma variate function. Utilizing this fact the idea is a follows: Fit a gamma-variate curve to the sparse patient data, and then using this curve as the loss component for the last compartment. The resulting curve from an experiment with this method is illustrated in figure 4.18.

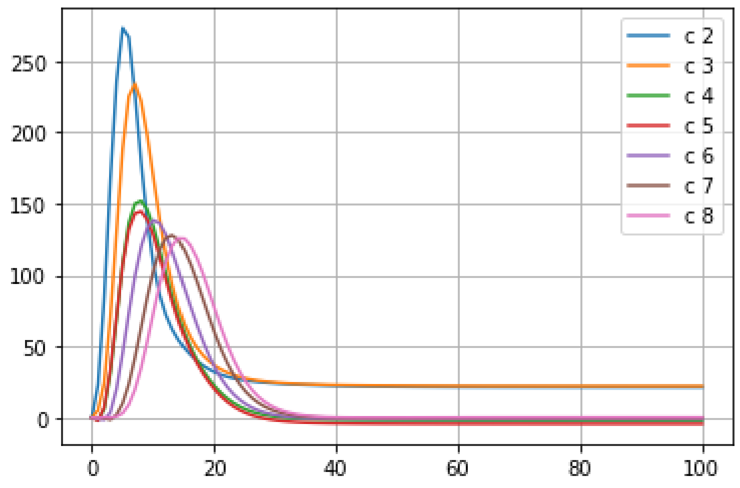


FIGURE 4.18: 8 Compartments With Gamma Variate $c8$ loss

Shape-wise, this looks more promising. However, note that the fit is still far from perfect. Interestingly, in this plot compartment $c2$ and $c3$ does not converge towards zero. This might be an indication of in-fighting in the model. That is, this scheme can be thought of as squeezing out the solution from two sides: One from satisfying the differential equation, and another from satisfying the gamma variate data. If the two criteria does not coincide into a coherent solution, it is reasonable to assume there is bound to be some artifacts in the fitted model.

Now with somewhat convergent model without variable parameters, the task becomes to see if there is a possibility of adding model variables in such a way that the fit becomes more accurate and also yields a better estimate of cardiac output (CO). Recall that the optimal fit in the solution to the numeric model, the optimal Q is found by matching the TTP of the last compartment to the TTP found in patient data. Additionally, the patients relative blood-volume, dictates the relative scaling of compartment volumes to the reference model. Therefore there are two suggested parameters s_q, s_v. These parameters can be employed as scaling factors for Q and $[v_1, ..., v_2]$, in the differential loss function of the PiNN implementation.

Strategy 2

Strategy 2 builds upon the approach used in the SIR case, an adoption of the method used by Shaier, Maziar Raissi, and Seshaiyer, 2021, which presupposes underlying data, either simulated or sampled. Recall that in strategy 1, the PiNN method was implemented to solve for all 8 compartments, whilst sample data was only provided for compartment 8. For this strategy, simulation data is pre-generated from the original simulation model. Further in this specific case the methodology can be simplified. Mainly, suppose that one can base a solution upon the 8-compartment simulation data. The strategy involves only utilizing the simulation data from compartment 7, and using a PiNN to predict compartment 8. In essence, one takes the simulation data generated by the original model which was to some degree not optimal, and *correcting* using PiNN with added parameters. The main reason for this strategy is to attempt to reduce the degrees of freedom which occurs in strategy 1. In this case there are three suggested parameters, s_v, s_q, s_{c_7} where s_v, s_q scales Q and compartment volumes $\mathbf{v}$ as in strategy 2, and s_{c_7} correspond to a scaling of the enhancement value of compartment 7 (c_7).

4.6.7 Testing Schemes

Testing schemes are set up for both strategies. Six records of patient data is provided. For the discussed strategies, interest lies in investigating if adding parameters, and interactions between these parameters can be beneficial to the prediction accuracy. In principle, if the parameters converge towards reasonable values, it can then be reasoned that the found parameter have a qualitative explanation. 2x4 models are constructed and trained for strategy 1, and 2x6 for strategy 2. The models are constructed to incorporate scaling of Q, $\mathbf{v}$, both, or none for strategy 1, and trained using pure data as well as to a fitted gamma variate function.

Model
S_q and S_v
S_v
S_q
No Params
S_q, S_v Gamma
S_v Gamma
S_q Gamma
No Params Gamma

TABLE 4.8: Strategy 1 models

Table 4.8 denote the trial models. In order to see if there is a present interaction effect between the parameters, models are constructed for each element of the power-set of parameters. The testing scheme for strategy 2 is similar. However, due to one more additional parameter, there are twice the amount of models that can be constructed.

Models	
S_q, S_v, S_c	γ/data
S_q, S_v	γ/data
S_q, S_c	γ/data
S_q	γ/data
S_v, Sc	γ/data
S_c	γ/data
S_v	γ/data
No Params	γ/data

TABLE 4.9: Strategy 2 models

Due to the large number of models, and relatively similar data between patients, strategy 2 will first only be run over data from one patient. The reasoning for this is that most of the models are likely to be sporadic and not accurate. If however, some of the models seem promising they can be carried over and tested on more than one patient.

4.6.8 Results

Each of the two strategies were tested in order to see if any of the added parameters converged towards reasonable values. Reasonable in this case would mean relatively small changes, as the parameters themselves scale values from a reference model.

Strategy 1

Model	Train Loss	S_q Estm	S_v Estm	Q Estm
S_q, S_v	0,00122	0.6966	1,9292	2938.9
S_v	0,00030	NA	1,9033	NA
S_q	0,00051	0.5897	NA	2457,8
No Params	0,00079	NA	NA	NA
S_q, S_v γ	0.00027	0.5024	1,9984	2119,4
S_v γ	0,00041	NA	1,9994	NA
S_q γ	7.54e-05	0,5661	NA	2388
No Params γ	0,00028	NA	NA	NA

TABLE 4.10: Strategy 1 performance

Results from strategy 1 reported for one patient in table 4.10. Learned parameters, training loss value, as well as scaled Q for model including the S_q term is reported. Overall, the results for table 4.10 indicates that strategy 1 does not work. Note that although results for only one patient is reported in this table, the results are similar for all patients. The results can be broken down for each term.

S_q It may seem as though the S_q values converged toward reasonable answers. However, for this parameter 0.5 was set as the minimum value. This indicates that the variable is likely converging towards 0. Although the true patient CO cannot be directly stated, it is safe to say that the estimated Q fro this strategy by far under-shoots the true Q value, in all models which include the S_q term.

S_v In strategy 1, all models including the S_v term tend towards 2, which is the set max value for this term in the implementation. This means overall that the S_v has not seemed to converge towards a reasonable answer, but rather increases indefinitely.

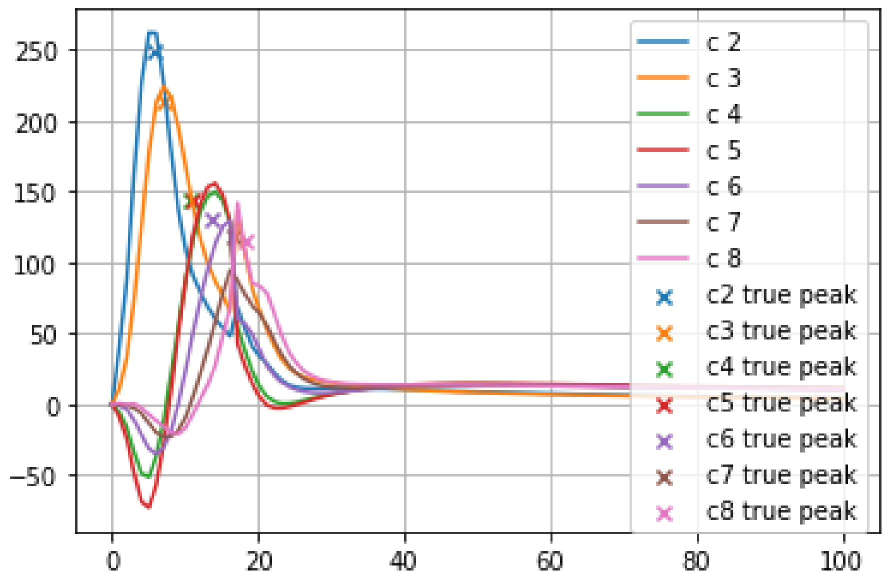


FIGURE 4.19: Strategy 1, No-parameter-model, trained on patient data

Notice figures 4.19 - 4.20. These plots show a prediction output of the parameter free model of strategy. Also noted on the plots, is the peak values from the numeric solution. Also note the relative proximity of the predicted peak values to the true values. In the pure data case this prediction does seems relatively inaccurate, however using the gamma variate end fit, the TTP points seems relatively close to that of the simulated solution.

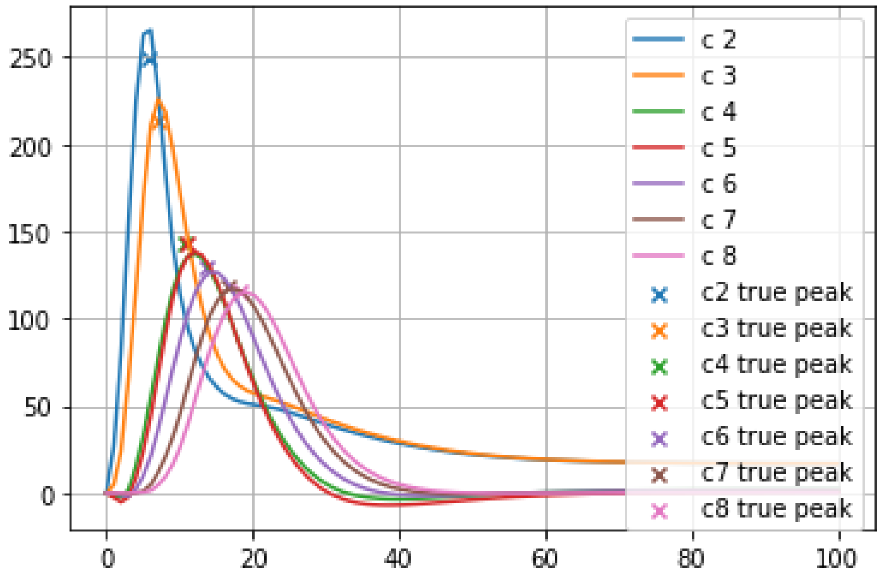


FIGURE 4.20: Strategy 1, No-parameter-model, trained on gamma variate end data

Strategy 2

Table 4.11 summarizes the performance of the parameter models of strategy 2, trained for one patient.

Model	Train Loss	S_q estm	S_v estm	s_{c_7} estm	Q
S_q, S_c, S_v	0,00059	0,82933	2,09613	0,51898	3498,6
S_q, S_v	0,00119	0,65260	2,40454	NA	2753,1
S_q, S_c	0,00181	0,29984	NA	0,39214	1264,9
S_q	0,00125	0,22125	NA	NA	933,4
S_v, S_c	0,00457	NA	5,91283	0,94582	NA
S_c	0,00203	NA	NA	0,62808	NA
S_v	0,00222	NA	7,99795	NA	NA
No Params	0,00774	NA	NA	NA	NA
$S_q, S_c, S_v\ \gamma$	0,00144	0,20500	2,26449	1,00000	864,8
$S_q, S_v\ \gamma$	0,00134	0,22289	2,56976	NA	940,3
$S_q, S_c\ \gamma$	0,00131	0,15377	NA	0,78783	648,7
$S_q\ \gamma$	0,00137	0,08539	NA	NA	360,2
$S_v, S_c\ \gamma$	0,00135	NA	6,25703	0,78874	NA
$S_c\ \gamma$	0,00339	NA	NA	0,72594	NA
$S_v\ \gamma$	0,00135	NA	9,33360	NA	NA
No Params γ	0,00748	NA	NA	NA	NA

TABLE 4.11: Strategy 2, one patient results

Again consider the parameters individually.

S_v As can be observed in table 4.11, the models which have the S_q term tend to yield low estimates of the parameter. This is most evident by observing the model which only incorporates S_q, which settles on the lowest value in both the pure data and gamma variate training instances.

S_q Likewise with S_q, this parameter also does not seem to converge towards any value but decreases indefinitely.

S_{c_7} This parameter actually seems to converge to a value around 0.62 in the straight data model, and 0.725, in the gamma variate training model.

Plots

Some plots are given to illustrate how the attempted variants in strategy 2 affect the solution curve. Figure 4.21 depicts the shape of a neural network without any form of differential equation loss fit, naively trained over patient data.

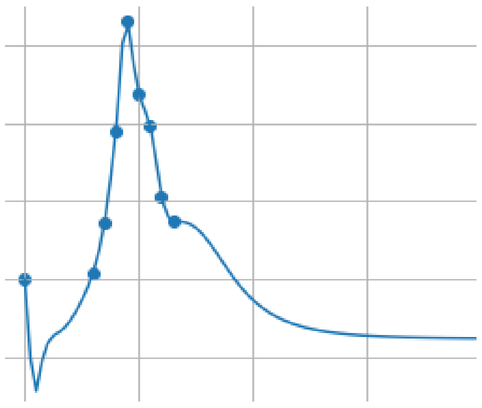


FIGURE 4.21: Pure data, only neural network model

The output all models which are trained upon a gamma fit of patient data follow the gamma curve exactly, see figure 4.22

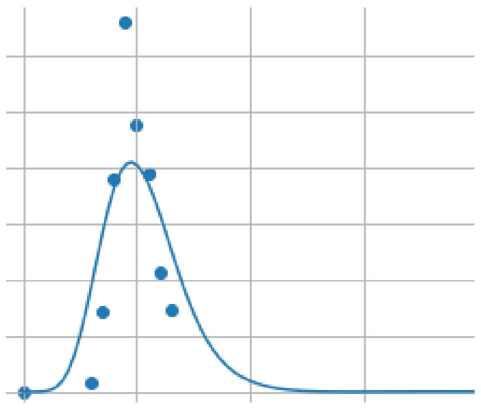


FIGURE 4.22: S_q, S_c model, gamma fit (hidden numbers)

Figure 4.23, depicts, the output of the strategy 1 model trained on patient data. Observe that there are some regularization effects present. Compared to a naive neural net fit, most of the models perform similarly. An argument can be made toward evidence of some regularization effects seen in the most of the models.

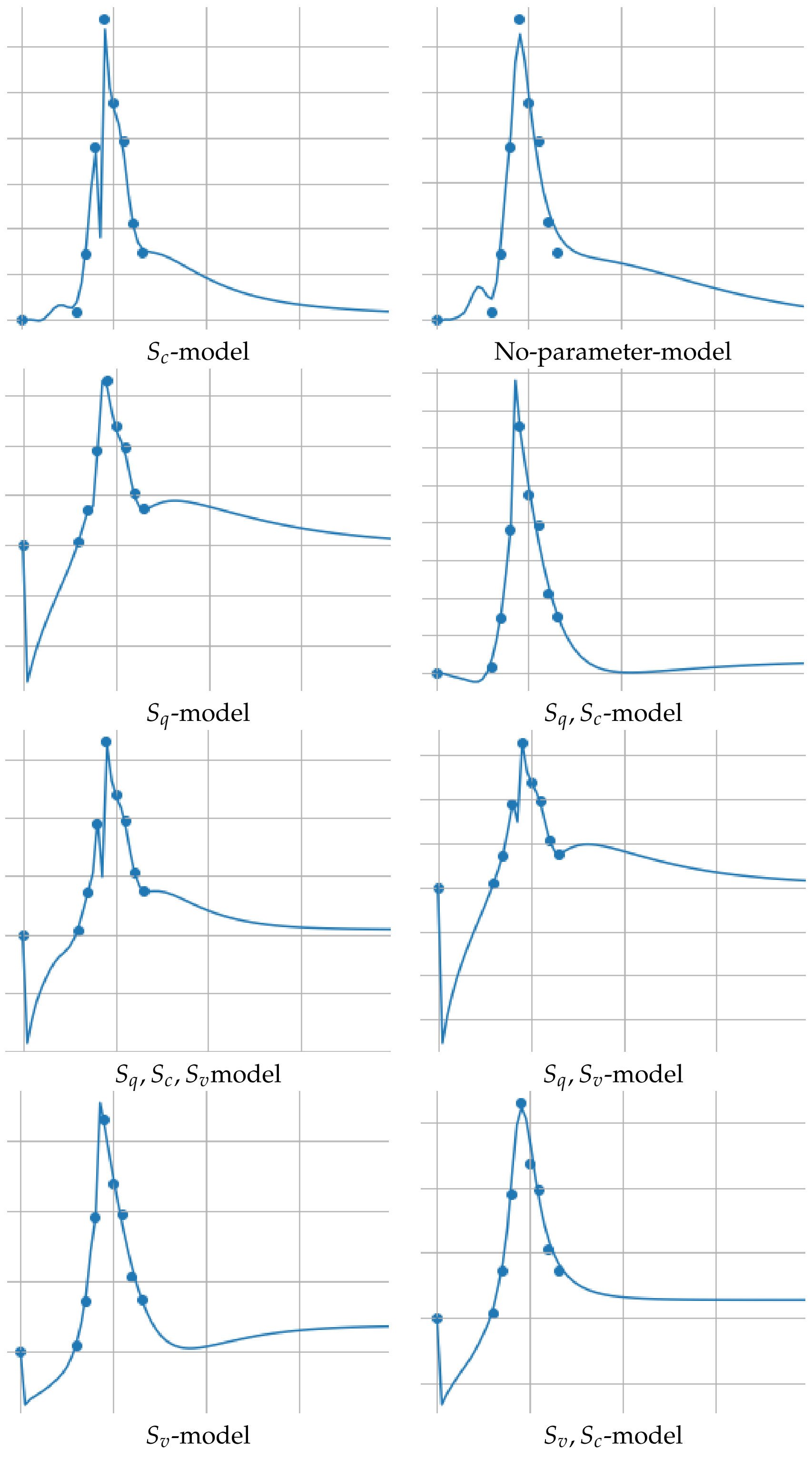


FIGURE 4.23: Strategy 2 plots, (hidden numbers)

As can be seen from the plots in figure 4.23, the plots are fairly varied. Recall figure 4.21, which corresponds to the naive data fit. Since the numbers are hidden from the plot in order to protect patient data, interpreting the plots may be somewhat challenging. However, there are still noticeable effects. Note that the first data-point corresponds to a zero value. Present in most of the fitted model, and the naive data fit is a downward spike below the first data-point corresponding to negative value output. Recall that since the compartment model in this case is supposed to model *enhancement*, a measure of concentration, negative values are meaningless. Shape-wise the no-parameter model and interestingly the S_q, S_v model seems the most promising. Note that this seems as fulfilling the goal of producing a model which better fits the data. However, even with the over-arching PiNN functionalities of the strategies, the models seems prone to over-fitting. In addition, this does not yield any reasonable prediction of patient CO, which was the other overall goal.

4.6.9 Case four summary and discussion

Two strategies have been proposed as potential realization of the PiNN methodology applied to a compartment model simulating the contrast agent enhancement signature of a test bolus injection. The original methodology was adapted in such a way that the PiNN methodology could build upon the original simulated model and incorporate real-world patient data. The results were varied but overall, a clear way to adapt the PiNN methodology, the original simulation model, and the patient data in an ultimately cohesive model was not achieved. However there were some interesting results. In particular, the PiNN applied to the compartment model of eight compartments, came close to matching the time of peak (TTP) values of the simulated model, giving some retribution to the methodology. An attempt to augment the methodology, was made i two strategies which involved attaching parameters to the original compartment model differential equation. Based upon the findings of Shaier, Maziar Raissi, and Seshaiyer, 2021 and the results of case three, demonstrating the parameter discovery property of the PiNN methodology, the goal was to see if these parameters as means to scale the underlying parameters of the original model would converge towards realistic predictions of the parameters they were meant to scale. Admittedly, it may have been naive to construct an artificial method by attaching arbitrary parameters to a model. Although it should be noted that it was not unforeseen that the proposed methodology would not work, what lacks in substantial predictive results feeds into a more general pitfall to consider when constructing a model of this kind. What seems to be a contributing factor to why this methodology will not work can be best realized by considering the main goal of the optimization. That is, reduce the loss function as close to zero as possible. The way these parameters are placed in the differential loss function makes this entire loss component proportional to the parameter: $Loss = \lambda \cdot f(x, ..)$. Then the simplest way to reduce the loss can be to just set $\lambda = 0$.

Methodology vs implementation

Recall that the problem statement of this case was to see if one could improve an established simulation model by applying PiNN. In case four, it is possible to use PiNN as a neural solver, only, that the methodology seems to find useful results until about the three first compartments, and the rest being sporadic. Recall also how the original simulated model worked in relations to data: Set up reference compartment model in regards to blood volume, increment Q and solve the differential compartment model with a numeric method. The Q value which matches the TTP of the solution with the one found in data is the estimated CO. Then ideally, by applying the PiNN methodology and utilizing the parameter discovery property, one would obtain an estimate of patient CO. This may work when having a clearly defined situation, with available data generated by a process with known physics, which is true for the SIR model of case three. In case four, there was no clearly discernible way to augment the original model in such a way that PiNN was appropriate as a method. An attempt to rectify this was provided by naively attaching parameters to the underlying differential model in hopes that the parameter discovery quality of the method as demonstrated in Shaier, Maziar Raissi, and Seshaiyer, 2021 Further, in order for this proposed methodology to work, there should be a reasonable way to augment this model in such a way that applying PiNN would yield a model better suited to the underlying problem. At first glance it may have seem that this was the case, only this introduces the problem that by adding a parameter, it alters the underlying differential model. This approach fails to incorporate and important distinction described by Karniadakis et al., 2021, as no model can be constructed without assumptions, its important for those assumption to append to the situation at hand. By introducing additional parameters to the model it may impose a bias, which then may cause incoherent results. Hence, care must be taken when introducing variables as to not introduce biases, or allow for the model loss function to be reducible to a zero-product.

Chapter 5

Conclusion

5.1 Case-problems

In this thesis four case problems has been presented with specific goals involving the use of Physics Informed Neural Networks. These cases were targeted towards the two larger goals of first gaining and understanding of the method, and ultimately see if the method can be applied in a real-world case. Case one, is a simple demonstration on how a neural network can be utilized to find solutions to differential equations. As such, the neural solver is demonstrated to be able to find an accurate solution within the interval on which it is trained. Case two was also a simple demonstration of the methodology working as a regularization mechanism. Case three was the application of PiNN to the SIR compartment model. Utilizing the methodology developed by M. Raissi, P. Perdikaris, and G. Karniadakis, 2019 and Shaier, Maziar Raissi, and Seshaiyer, 2021, the case task was to investigate the properties and benefits of the PiNN methodology. In more detail, in the specific case of a SIR model, the methodology's ability to estimate model parameters, and regularize neural network models was explored. With this goal in mind, seven rounds of tests were performed over simulation data sets of varying parameters where the last two tests had added noise. The results from these tests were varying. Overall, some of the same findings as Shaier, Maziar Raissi, and Seshaiyer, 2021, were found. However, the specific realization of the methodology fails to gain confident results. In part, the parameter discovery property was demonstrated, although the overall accuracy can be deemed unsatisfactory. Case four was realized as an attempt to apply the PiNN methodology in a real world compartment model case. The main goals of this case was to see if an established physical model could be improved by applying PiNN. Although the tests were inconclusive some interesting results were discovered. In part, the PiNN model was found to be able to nearly approximate the time to peak found in the simulated model

5.1.1 When is PiNN most sensible?

It may be fair to note the fact that the chosen case problems perhaps fall short in show-casing the advantages of PiNN. All of the case problems are not only to an extent solvable with traditional numeric methods, but also depend on solutions achieved by these means in order to demonstrate the PiNN methodology. A benefit of PiNN is the ability to adapt the flexibility achievable with a neural network in order to solve hard (non-linear) differential equations. Another benefit it the ability of leveraging of prior information as a differential equation in order to regularize what would otherwise have been an under-performing machine learning model. Utilizing the PiNN methodology of M. Raissi, P. Perdikaris, and G. Karniadakis, 2019, where the model incorporates both underlying data and a differential model, the validity of the applied differential equation is assumed. Hence consideration of the appropriateness of the utilized differential equation, to the sampled data becomes important. As an example, in case four, it can be argued that the differential equation which was applied was invalidly specified. In this case, a PiNN model training instance compels the model to satisfy a set of data and a differential equation which to some degree fights against each other. The failure of this case-problem serves an important distinction. The PiNN methodology makes most sense when there is an appropriate differential equations directly relevant to the data.

5.2 Conclusion

Physics Informed Neural Networks has proven to be an exiting methodology linking together and mending problems from the fields of machine learning and numerical computing. The methodology has found various adaptions, improving upon machine learning modes by incorporating prior information in the form of differential equations as means to regularize machine learning models. In the same vein, the methodology is also able to leverage the high flexibility and adaptability befitted a ML model for solving difficult non-linear differential equations. With the potential of this methodology, the main goal of this thesis was to gain an understanding of the methodology trough case problems, and ultimately attempting to adapt the methodology in real world biomedical setting. With this goal in mind, several aspects and difficulties surrounding the methodology was unearthed. Applications of PiNN are demonstrated and the realizations demonstrate how to apply the methodology in order to utilize its regularization benefits and parameter discovery properties.

The literature suggests there are selected cases where clear benefits of PiNN over traditional methods become apparent in regards to computational resources and accuracy (Cuomo et al., 2022), (Karniadakis et al., 2021). Arguably, the case-problems of this thesis does not demonstrate clear benefits of the PiNN methodology over traditional methods. However, what the case-problems does demonstrate, are overall indications of the methodology working provided a well-conditioned problem. As is demonstrated in the second and third case problems, the methodology does condition a model to fit closer to a function satisfying a differential equation than an overall naively data-trained model. As in case three where the model fits additional parameters, this effect and benefit is seen when the model is able to learn the correct parameters. That being said, as case three further demonstrates and supported in the literature (Shaier, Maziar Raissi, and Seshaiyer, 2021), provided enough sampled data points the PiNN methodology is able to learn the correct parameters. Testing indicates that this property is sensitive to the underlying parameters and noise. Further, the case problems reveals details in regards to implementation and practices as well as some of the criteria necessary for the PiNN model to work optimally. In addition, PiNN as a neural solver, that is as a neural network model over a loss function valuing only degree of fit for a differential equation, was demonstrated to find solutions to differential equations. The shortcomings of this as means of solving a compartment model was also discussed, where in particular the initial conditions were identified as important factors in determining the models ability to find a solution. Overall, the cases does demonstrate that the methodology works, and that it can be leveraged as to learn parameters, and find solutions to differential equations. In addition it has shed light upon necessary considerations to make when applying the methodology. As the fourth case-problem demonstrates, the biases of the underlying differential equation employed must be considered. All case problems also highlight important details in regards to implementation, as all of these cases were realized from a bottom up approach in python utilizing PyTorch.

Hopefully, the overall takeaway from this thesis is an understanding on how to apply PiNN, in the specific sense by means of calculating differential equation loss in order to train a neural network, and in the more general sense by considering how a model should be set up so that the methodology works in a sensible manner. Important properties and implementation details have been unearthed, exposing partly the potential benefit of the method and perhaps illuminating a path for future adaptions.

Bibliography

Agarwal, Ravi P. and Donal O'Regan (2009). *Ordinary and Partial Differential Equations. With Special Functions, Fourier Series, and Boundary Value Problems*. Springer New York, NY. ISBN: 978-0-387-79146-3. DOI: https://doi.org/10.1007/978-0-387-79146-3.

Bae, KT, JP Heiken, and JA Brink (1998). "Aortic and hepatic contrast medium enhancement at CT. Part I. Prediction with a computer model". In: DOI: 10.1148/radiology.207.3.9609886.

Baeldung (2020). *Normalizing Inputs for an Artificial Neural Network*. URL: https://www.baeldung.com/cs/normalizing-inputs-artificial-neural-network (visited on 10/05/2022).

BlackwoodJulie and Lauren Childs (2018). "An introduction to compartmental modeling for the budding infectious disease modeler". In: *Letters in Biomathematics* 5.1, pp. 195–221. DOI: 10.30707/LiB5.1Blackwood. URL: https://lettersinbiomath.journals.publicknowledgeproject.org/index.php/lib/article/view/81.

Cuomo, Salvatore et al. (2022). "Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next". In: *CoRR* abs/2201.05624. arXiv: 2201.05624. URL: https://arxiv.org/abs/2201.05624.

Devlin, Jacob et al. (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

Doshi, Sanket (2019). *Various Optimization Algorithms For Training Neural Network*. URL: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6 (visited on 01/13/2019).

Fowler, David and Eleanor Robson (1998). "Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context". In: *Historia Mathematica* 25.4, pp. 366–378. ISSN: 0315-0860. DOI: https://doi.org/10.1006/hmat.1998.2209. URL: https://www.sciencedirect.com/science/article/pii/S0315086098922091.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hiorth, Aksel (2022). *A Model for Predicting Test Bolus Geometry and Time to Peak*.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

Iserles, Arieh (2008a). *A First Course in the Numerical Analysis of Differential Equations*. 2nd. USA: Cambridge University Press. ISBN: 0521734908.

— (2008b). *A First Course in the Numerical Analysis of Differential Equations*. 2nd ed. Cambridge Texts in Applied Mathematics. Cambridge University Press. DOI: 10.1017/CBO9780511995569.

Karniadakis et al. (2021). "Physics-informed machine learning. Nat Rev Phys 3, 422–440 (2021)." In: *Nature Reviews Physics*. DOI: https://doi.org/10.1038/s42254-021-00314-5.

Knolle, Helmut (1976). "Lotka-volterra equations with time delay and periodic forcing term". In: *Mathematical Biosciences* 31.3, pp. 351–375. ISSN: 0025-5564. DOI: https://doi.org/10.1016/0025-5564(76)90090-0. URL: https://www.sciencedirect.com/science/article/pii/0025556476900900.

Krishnapriyan, Aditi S. et al. (2021). "Characterizing possible failure modes in physics-informed neural networks". In: DOI: 10.48550/ARXIV.2109.01050. URL: https://arxiv.org/abs/2109.01050.

Kröger, M and R Schlickeiser (2020). "Analytical solution of the SIR-model for the temporal evolution of epidemics. Part A: time-independent reproduction factor". In: *Journal of Physics A: Mathematical and Theoretical* 53.50, p. 505601. DOI: 10.1088/1751-8121/abc65d. URL: https://doi.org/10.1088/1751-8121/abc65d.

Kyongtae, Bae T. (2010). "Intravenous contrast medium administration and scan timing at CT: considerations and approaches." In: DOI: doi:10.1148/radiol.10090908.

Lagaris, I.E., A. Likas, and D.I. Fotiadis (1998). "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Transactions on Neural Networks* 9.5, pp. 987–1000. ISSN: 1045-9227. DOI: 10.1109/72.712178. URL: http://dx.doi.org/10.1109/72.712178.

LeVeque, Randall J. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia. ISBN: 978-0-898716-29-0.

Madsen, M T (1992). "A simplified formulation of the gamma variate function". In: *Physics in Medicine and Biology* 37.7, pp. 1597–1600. DOI: 10.1088/0031-9155/37/7/010. URL: https://doi.org/10.1088/0031-9155/37/7/010.

Pang, Guofei, Lu Lu, and George Em Karniadakis (2019). "fPINNs: Fractional Physics-Informed Neural Networks". In: *SIAM Journal on Scientific Computing* 41.4, A2603–A2626. DOI: 10.1137/18m1229845. URL: https://doi.org/10.1137%5C%2F18m1229845.

Petzold, Linda (Mar. 1983). "Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations". In: *SIAM Journal on Scientific and Statistical Computing* 4. DOI: 10.1137/0904010.

Raissi, M., P. Perdikaris, and G.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. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis (2017). *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. arXiv: 1711.10561 [cs.AI].

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6, pp. 386–408.

Schiassi, Enrico et al. (2021). "Physics-Informed Neural Networks and Functional Interpolation for Data-Driven Parameters Discovery of Epidemiological Compartmental Models". In: *Mathematics* 9.17. ISSN: 2227-7390. DOI: 10.3390/math9172069. URL: https://www.mdpi.com/2227-7390/9/17/2069.

Shaier, Sagi, Maziar Raissi, and Padmanabhan Seshaiyer (2021). *Data-driven approaches for predicting spread of infectious diseases through DINNs: Disease Informed Neural Networks*. DOI: 10.48550/ARXIV.2110.05445. URL: https://arxiv.org/abs/2110.05445.

Simard, Patrice, Dave Steinkraus, and John Platt (Sept. 2003). “Best Practices for Convolutional Neural Networks”. In.

Stiasny, Jochen, Samuel Chevalier, and Spyros Chatzivasileiadis (2021). *Learning without Data: Physics-Informed Neural Networks for Fast Time-Domain Simulation*. DOI: 10.48550/ARXIV.2106.15987. URL: https://arxiv.org/abs/2106.15987.

Theodoridis, Sergios (2020). *Machine Learning A Bayesian and Optimization Perspective (Second Edition)*. Ed. by Sergios Theodoridis. Second Edition. Academic Press. ISBN: 978-0-12-818803-3. DOI: https://doi.org/10.1016/C2019-0-03772-7.

Thompson, Maynard and H. I. Freedman (1982). “Deterministic Mathematical Models in Population Ecology.” In: *American Mathematical Monthly* 89, p. 798.

Appendix A

SIR Test rounds plots and tables 3-5, 7

Round 3. N = 1, I0 = 0.50, S0 =0.50, beta = 0.50, gamma = 0.05

Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	4,54E-05	2,44E-06	0,0925	0,0461
010DA100DE	1,95E-05	5,82E-06	0,2762	0,0478
010DA000DE	2,76E-06	4,79E-05	NA	NA
000DA100DE	6,19E-05	0,58844	NA	NA
	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	2,66E-07	2,26E-07	0,1125	0,0500
010DA100DE	4,95E-08	8,74E-08	0,2506	0,0500
010DA000DE	1,00E-07	8.77e-08	NA	NA
000DA100DE	9,43E-07	0,06973	NA	NA

TABLE A.1: Round 3 Results

Round 4. N = 1, I0 = 0.01, S0 =0.99, beta = 0.70, gamma = 0.16

Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	1,52E-03	4,14E-03	-0,0196	0,0152
010DA100DE	2,22E-02	2,43E-01	0,1148	0,7243
010DA000DE	4,08E-05	1,7506	NA	NA
000DA100DE	7,31E-06	1,7195	NA	NA
	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	1,84E-05	3,08E-04	0,5884	0,1512
010DA100DE	2,45E-05	5,15E-03	0,0522	0,1536
010DA000DE	1,46E-07	0,0051	NA	NA
000DA100DE	2,63E-06	1,97311	NA	NA

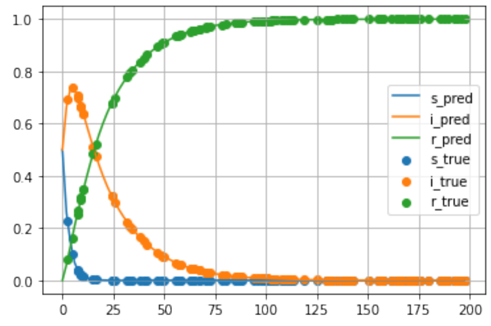
TABLE A.2: Round 4 Results Table

Round 5. N = 1, I0 = 0.20, S0 =0.80, beta = 0.70, gamma = 0.16				
Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	1,01E-05	2,81E-05	0,2791	0,1555
010DA100DE	7,62E+02	2,0768	0,8728	0,7341
010DA000DE	1,22E-05	15,5285	NA	NA
000DA100DE	1,87E-05	0,0099	NA	NA
	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	4,51E-08	2,98E-07	0,2982	0,1598
010DA100DE	1,14E-03	1,67E+01	0,0207	0,1965
010DA000DE	5,02E-06	16,7363	NA	NA
000DA100DE	1,35E-07	0,00865	NA	NA

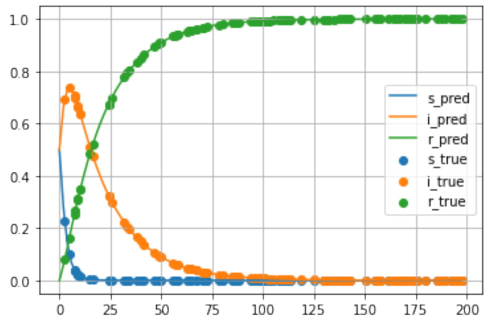
TABLE A.3: Round 5 Results Table

Round 7, $N = 1, I_0 = 0.20, S_0 = 0.80, \beta = 0.20, \gamma = 0.05, Noise = 1.4\sigma$				
Model Type	10 Ep Train MSE	10 Ep Test MSE	10 Ep Beta	10 Ep Gamma
100DA100DE	0,02157	0,01882	0,3797	0,1178
010DA100DE	0,11412	0,1462	0,0278	0,0404
010DE1000DE	0,10813	0,18638	0,0256	0,0402
	TUS Train MSE	TUS Test MSE	TUS Beta	TUS Gamma
100DA100DE	0,01864	0,01892	0,0879	0,0317
010DA100DE	0,02763	0,56877	0,0147	0,0277
010DE1000DE	0,03191	0,60218	0,0257	0,0395

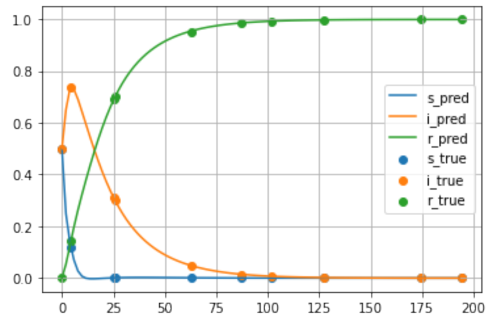
TABLE A.4: Round 7 results



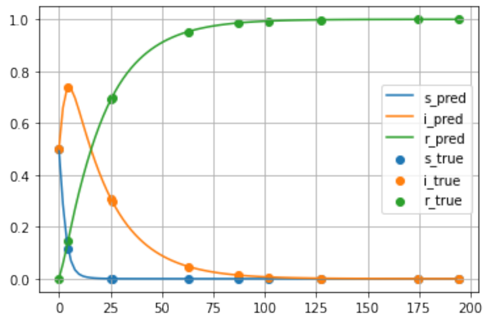
100DA100DE 10 Epochs



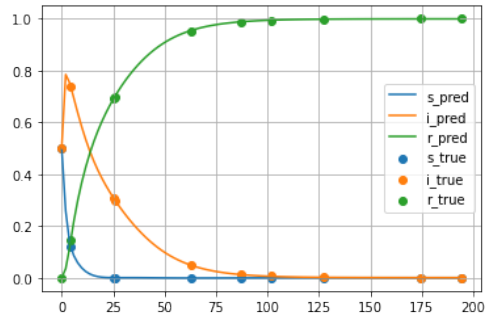
100DA100DE TUS



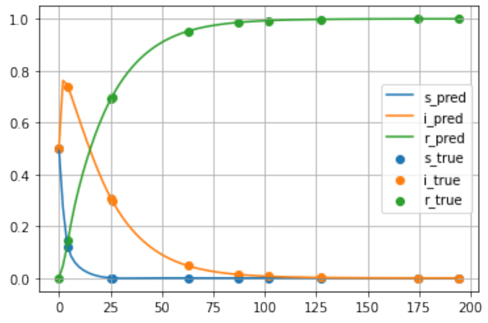
010DA100DE 10 Epochs



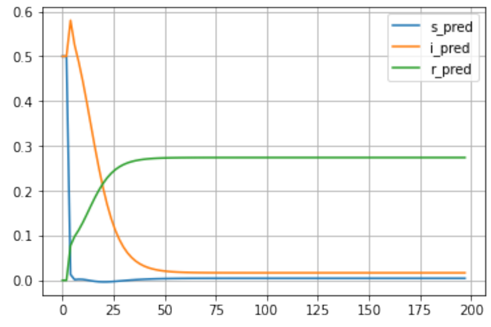
010DA100DE TUS



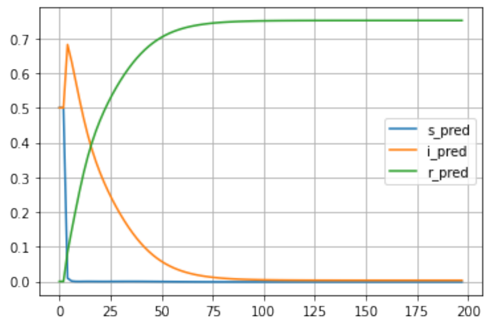
010DA000DE 10 Epochs



010DA000DE TUS

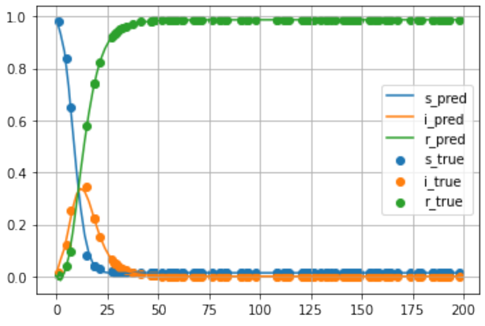


000DA100DE 10 Epochs

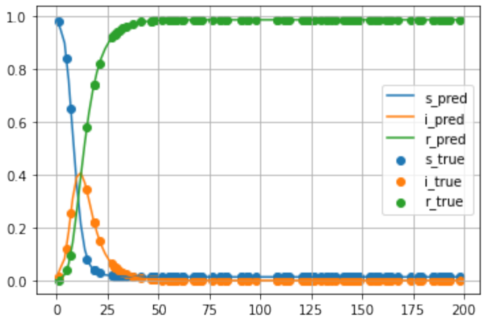


000DA100DE TUS

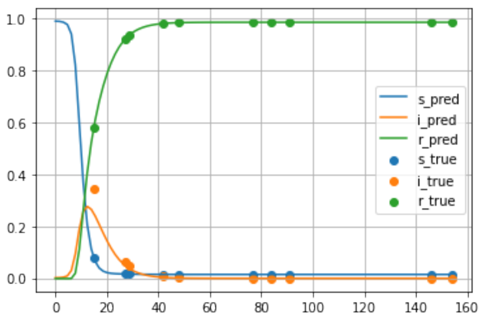
FIGURE A.1: Round 3 Plots



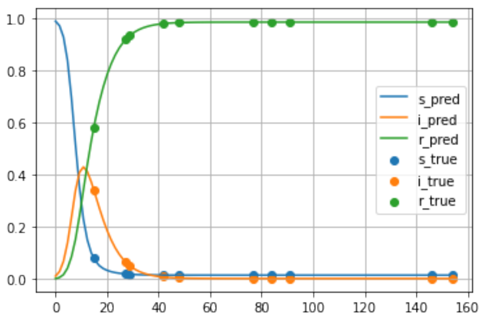
100DA100DE 10 Epochs



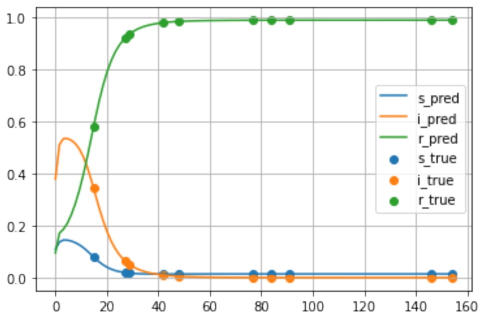
100DA100DE TUS



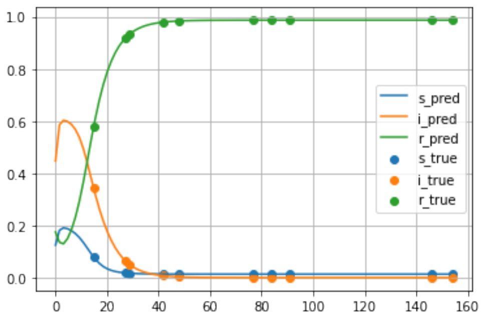
010DA100DE 10 Epochs



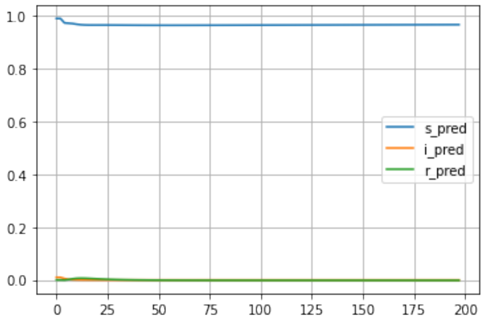
010DA100DE TUS



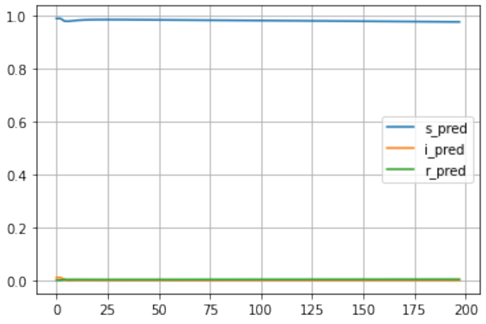
010DA000DE 10 Epochs



010DA000DE TUS

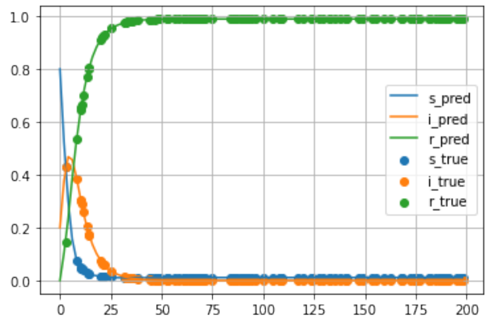


000DA100DE 10 Epochs

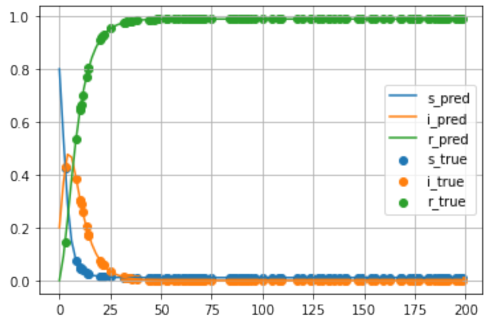


000DA100DE TUS

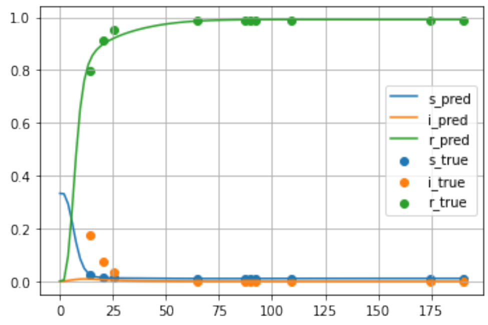
FIGURE A.2: Round 4 Plots



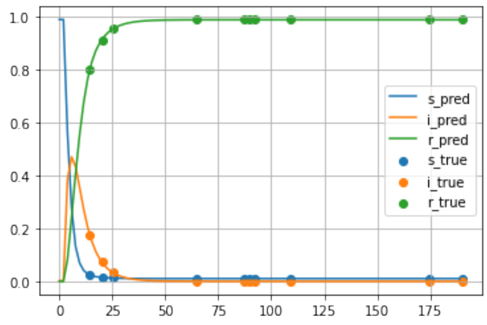
100DA100DE 10 Epochs



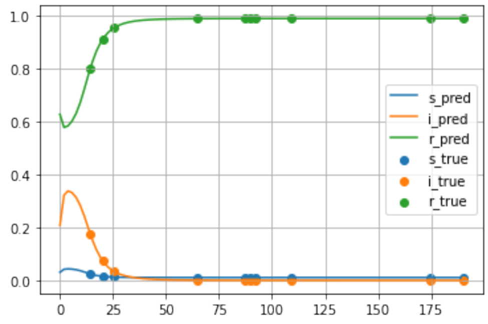
100DA100DE TUS



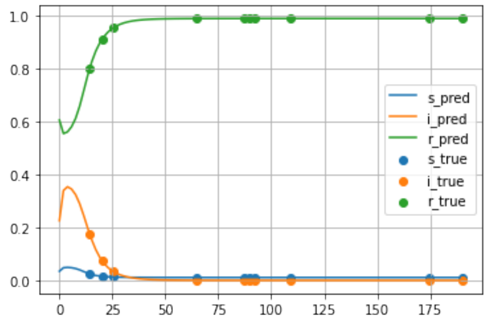
010DA100DE 10 Epochs



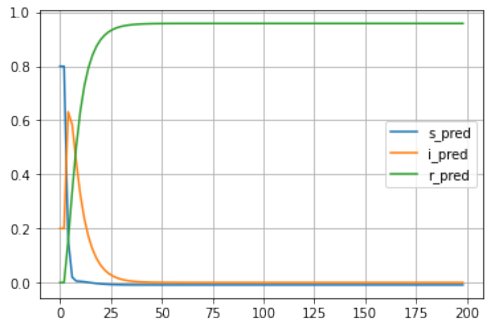
010DA100DE TUS



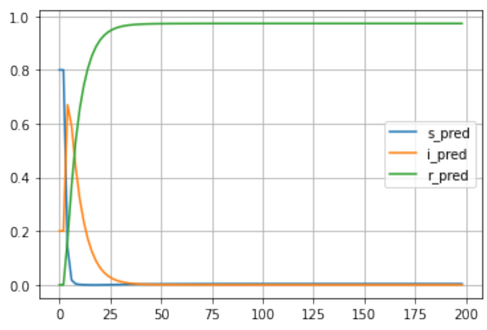
010DA000DE 10 Epochs



010DA000DE TUS

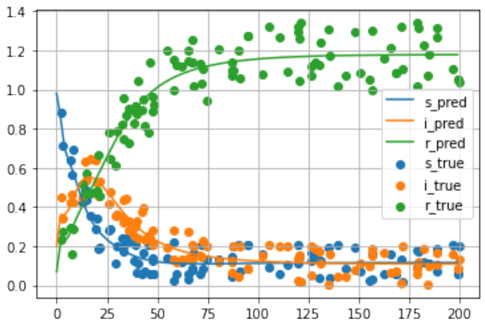


000DA100DE 10 Epochs

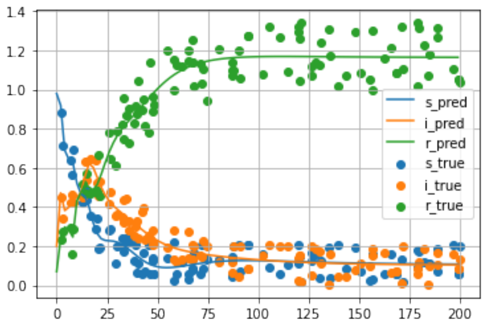


000DA100DE TUS

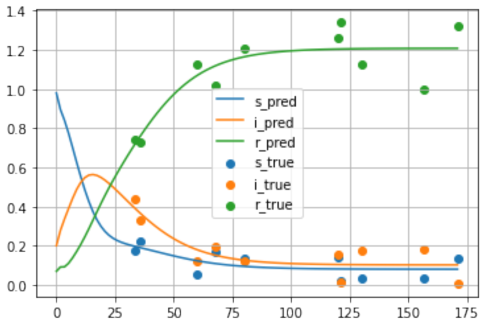
FIGURE A.3: Round 5 Plots



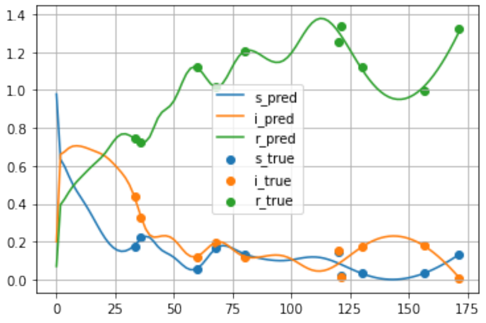
100DA100DE 10 Epochs



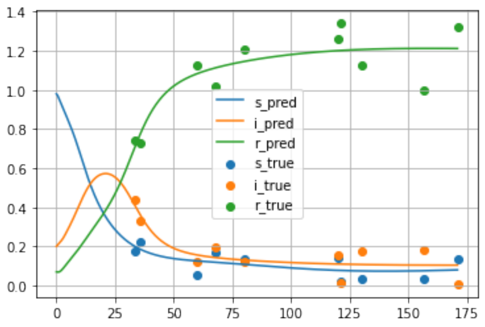
100DA100DE TUS



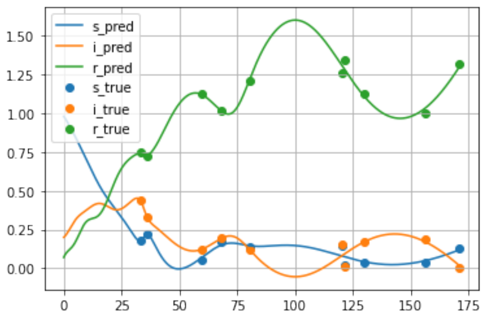
010DA100DE 10 Epochs



010DA100DE TUS



010DA1000DE 10 Epochs



010DA1000DE TUS

FIGURE A.4: Round 7 Plots

Appendix B

Python Code

Note, code is also available from:

https://github.com/lukerlars/PiNN_masterthesis

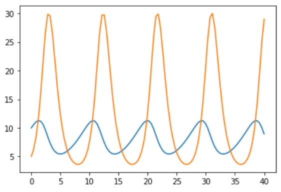
```
In [ ]: import numpy as np
        from scipy.integrate import odeint
        import matplotlib.pyplot as plt

        import torch
        from torch.autograd import grad
        import torch.nn as nn
        from numpy import genfromtxt
        import torch.optim as optim
        import matplotlib.pyplot as plt
        import torch.nn.functional as F
```

```
In [ ]: class Neural_net(torch.nn.Module):
            def __init__(self, n_in = 1, n_out =1):
                super(Neural_net, self).__init__()

                self.tanh = torch.nn.Tanh()

                self.layer1 = torch.nn.Linear(n_in,20)
                self.layer2 = torch.nn.Linear(20,20)
                self.layer3 = torch.nn.Linear(20,20)
                self.layer_out = torch.nn.Linear(20,n_out)

            def forward(self, x):
                x = self.layer1(x)
                x = self.tanh(x)
                x = self.layer2(x)
                x = self.tanh(x)
                x = self.layer3(x)
                x = self.tanh(x)
                x = self.layer_out(x)

                return x
```

```
In [ ]: class predprey_pinn:

            def __init__(self, epochs, data, c0):
                self.epochs = epochs
                self.model = Neural_net(n_out=2)
                self.domain = torch.linspace(0,int(max(data[0])),100, requires_grad=True).reshape(-1,1)

                self.lbfgs_optimizer = torch.optim.LBFGS(params = self.model.parameters(), lr = 0.001,max_iter = 500)
                self.adam_optimizer = torch.optim.Adam(params = self.model.parameters(), lr = 0.0001)

                self.alpha = 0.25
                self.beta = 0.02
                self.delta = 0.25
                self.gamma = 2

                self.t_dat = torch.tensor(data[0], dtype=torch.float).reshape(-1,1)
                self.x_dat = torch.tensor(data[1],dtype=torch.float)
                self.y_dat = torch.tensor(data[2], dtype=torch.float )

                self.maxes = {}
                self.mins = {}

                for id,d in enumerate((self.x_dat, self.y_dat)):
                    self.maxes[id] = max(d)
                    self.mins[id] = min(d)

                self.x_norm = self.normalize(0, self.x_dat)
                self.y_norm = self.normalize(1, self.y_dat)

                x0 = self.normalize(0,c0[0])
                y0 = self.normalize(1,c0[1])
                self.c0 = torch.tensor([x0,y0], dtype = torch.float)


            def normalize(self, id, unnormed):
                return (unnormed - self.mins[id])/(self.maxes[id]- self.mins[id])

            def un_normalize(self, id, normed):
                return normed*(self.maxes[id] -self.mins[id])+ self.mins[id]


            def wrap_grad(self, f,x):
                return torch.autograd.grad(f,x,
                grad_outputs=torch.ones_like(x),
                retain_graph=True,
                create_graph=True)[0]

            def de_loss(self):
                pred = self.model(self.domain)
                x,y = (d.reshape(-1,1) for d in torch.unbind(pred, dim =1))

                dx = self.wrap_grad(x, self.domain)
                dy = self.wrap_grad(y, self.domain)

                x = self.un_normalize(0,x)
                y = self.un_normalize(1,y)

                ls0 = torch.mean((dx - (self.alpha*x -self.beta*x*y)/(self.maxes[0]-self.mins[0]) )**2)
                ls1 = torch.mean((dy -(self.delta*x*y - y*self.gamma)/(self.maxes[1]-self.mins[1]))**2)
                ic = torch.mean((self.c0-pred[0])**2)

                return ls0 + ls1 + ic

            def data_loss(self):
                x,y = torch.unbind(self.model(self.t_dat), dim = 1)
                z1 = torch.mean((x - self.x_norm)**2)
                z2 = torch.mean((y- self.y_norm)**2)
                return z1 + z2



            def combined_loss(self):
                return self.de_loss() + self.data_loss()


            def plot_preds(self):
                x,y = torch.unbind(self.model(self.domain), dim = 1)
                x = self.un_normalize(0,x)
                y = self.un_normalize(1,y)
                plt.plot(self.domain.detach(), x.detach(), label = 'x pred')
                plt.plot(self.domain.detach(), y.detach(), label = 'y pred')

                plt.scatter(self.t_dat.detach(), self.x_dat, label = 'x data')
                plt.scatter(self.t_dat.detach(),self.y_dat, label = 'y data' )
                plt.legend()



            def lbfgs_train(self):
                        self.model.train()
                        for epoch in range(self.epochs):
                            def closure():
                                self.lbfgs_optimizer.zero_grad()
                                loss = self.combined_loss()
                                loss.backward()
                                return loss
                            self.lbfgs_optimizer.step(closure=closure)
                            print(f'Epoch {epoch}, loss: {self.combined_loss()}, beta_param: {self.beta}, gamma_param: {self.gamma}')
                        self.plot_preds()
```

```
    def adam_train(self):
            steps = 1000
            self.model.train()
            for epoch in range(self.epochs):
                for step in range(steps):
                    def closure():
                        self.adam_optimizer.zero_grad()
                        loss = self.combined_loss()
                        loss.backward()
                        return loss
                    self.adam_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, loss: {self.combined_loss()}, beta_param: {self.beta}, gamma_param: {self.gamma}')
            self.plot_preds()
```

```
In [ ]: alpha = 0.25
beta = 0.02
delta = 0.25
gamma = 2


def pp_ode(state, t):
    x,y = state
    dx = alpha*x -beta*x*y
    dy = delta*x*y - y*gamma
    return [dx, dy]

t = np.linspace(0,40,100)
sol = odeint(pp_ode,y0 =[10,5], t=t)

plt.plot(t, sol[:,0])
plt.plot(t, sol[:,1])

inp_dat = np.array([t, sol[:,0], sol[:,1]])
```



```
In [ ]: test_inst = predprey_pinn(epochs=10, data= inp_dat, c0 =[10,5])
```

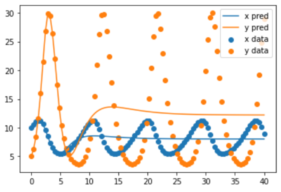
```
In [ ]: inp_dat.shape
```

```
Out[ ]: (3, 100)
```

Running adam train once:

```
In [ ]: test_inst.adam_train()
```

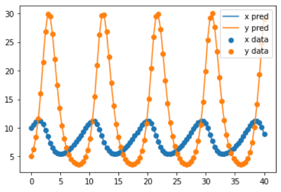
```
Epoch 0, loss: 0.23603315651416779, beta_param: 0.02, gamma_param: 2
Epoch 1, loss: 0.2267903834581375, beta_param: 0.02, gamma_param: 2
Epoch 2, loss: 0.219535470008850l, beta_param: 0.02, gamma_param: 2
Epoch 3, loss: 0.2114081084728241, beta_param: 0.02, gamma_param: 2
Epoch 4, loss: 0.20379692316055298, beta_param: 0.02, gamma_param: 2
Epoch 5, loss: 0.19838380813598633, beta_param: 0.02, gamma_param: 2
Epoch 6, loss: 0.19548077881336212, beta_param: 0.02, gamma_param: 2
Epoch 7, loss: 0.19437432289123535, beta_param: 0.02, gamma_param: 2
Epoch 8, loss: 0.1907251477241516, beta_param: 0.02, gamma_param: 2
Epoch 9, loss: 0.1887803077697754, beta_param: 0.02, gamma_param: 2
```



After several runs

```
In [ ]: test_inst.adam_train()
```

```
Epoch 0, loss: 0.0005453241756185889, beta_param: 0.02, gamma_param: 2
Epoch 1, loss: 0.00017405160l7889604, beta_param: 0.02, gamma_param: 2
Epoch 2, loss: 7.50718972994946e-05, beta_param: 0.02, gamma_param: 2
Epoch 3, loss: 5.512431380338967e-05, beta_param: 0.02, gamma_param: 2
Epoch 4, loss: 4.5014898205408826e-05, beta_param: 0.02, gamma_param: 2
Epoch 5, loss: 3.80473502445966e-05, beta_param: 0.02, gamma_param: 2
Epoch 6, loss: 3.25711807818152e-05, beta_param: 0.02, gamma_param: 2
Epoch 7, loss: 2.981994293804746e-05, beta_param: 0.02, gamma_param: 2
Epoch 8, loss: 2.495665285096038e-05, beta_param: 0.02, gamma_param: 2
Epoch 9, loss: 2.4334365662070923e-05, beta_param: 0.02, gamma_param: 2
```



Note: This converges much slower than just running on the data

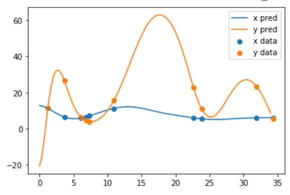
Test with fewer datapoints:

```
In [ ]: ids = np.random.choice(range(inp_dat.shape[1]), size = 10)
sample_data = inp_dat[:,ids]
```

```
In [ ]: test_inst2 = predprey_pinn(epochs=10, data = sample_data, c0 =[10,5])
```

```
In [ ]: test_inst2.adam_train()
```

```
Epoch 0, loss: 1.4953633353798068e-06, beta_param: 0.02, gamma_param: 2
Epoch 1, loss: 1.4808708783675684e-06, beta_param: 0.02, gamma_param: 2
Epoch 2, loss: 8.252820407506078e-06, beta_param: 0.02, gamma_param: 2
Epoch 3, loss: 1.4049605852051172e-06, beta_param: 0.02, gamma_param: 2
Epoch 4, loss: 1.4179156551108463e-06, beta_param: 0.02, gamma_param: 2
Epoch 5, loss: 3.3056655865948414e-06, beta_param: 0.02, gamma_param: 2
Epoch 6, loss: 1.3165924883651314e-06, beta_param: 0.02, gamma_param: 2
Epoch 7, loss: 1.2926308272653841e-06, beta_param: 0.02, gamma_param: 2
Epoch 8, loss: 1.2641256716960925e-06, beta_param: 0.02, gamma_param: 2
Epoch 9, loss: 1.3426794112092466e-06, beta_param: 0.02, gamma_param: 2
```



In [ ]:

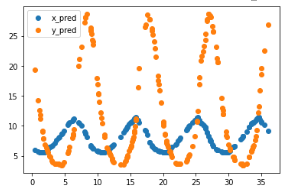
```python
class Duo_net(torch.nn.Module):
    def __init__(self, n_in = 1, n_out =1):
        super(Duo_net, self).__init__()

        self.tanh = torch.nn.Tanh()

        self.layer10 = torch.nn.Linear(n_in,20)
        self.layer11 = torch.nn.Linear(n_in,20)

        self.layer20 = torch.nn.Linear(20,20)
        self.layer21 = torch.nn.Linear(20,20)

        self.layer_out0 = torch.nn.Linear(20,n_out)
        self.layer_out1 = torch.nn.Linear(20,n_out)

    def forward(self, x):
        x0 = self.layer10(x)
        x0 = self.tanh(x0)
        x0 = self.layer20(x0)
        x0 = self.tanh(x0)
        x0 = self.layer_out0(x0)

        x1 = self.layer11(x)
        x1 = self.tanh(x1)
        x1 = self.layer20(x1)
        x1 = self.tanh(x1)
        x1 = self.layer_out0(x1)

        return x0, x1
```

In [ ]:

```python
class rk_pp_pinn:
    """ Instance of Runge kutta scheme PINN
    for predator prey model
    """

    def __init__(self, dt, xdata,t_dat, q = 100):
        self.model = Duo_net(n_in=2,n_out=q+1)
        self.adam_optimizer = torch.optim.Adam(params = self.model.parameters(), lr = 0.001)
        butcher_file = np.float32(np.loadtxt('Butcher_IRK100.txt', ndmin = 2))

        self.IRK_weights = torch.Tensor(np.reshape(butcher_file[0:q**2+q], (q+1,q)))
        self.IRK_times = torch.Tensor(butcher_file[q**2+q:])

        x0s,x1s = xdata
        self.x0s = torch.Tensor(x0s)
        self.x1s = torch.Tensor(x1s)
        self.t_dat = t_dat
        self.dt = dt
        self.epochs = 10

        self.alpha = 0.25
        self.beta = 0.02
        self.delta = 0.25
        self.gamma = 2

    def wrap_grad(self, f,x):
            return torch.autograd.grad(f,x,
            grad_outputs=torch.ones_like(x),
            retain_graph=True,
            create_graph=True)[0]

    def test(self):
        x,y = self.model(self.x0s)
        print(x.shape)



    def loss(self):
        x1, y1 = self.model(self.x1s)
        x = x1[:,:-1]
        y = y1[:,:-1]


        F0 = self.alpha*x -self.beta*x*y
        F1 = self.delta*x*y -y*self.gamma

        x0 = x1 - self.dt*torch.matmul(F0,self.IRK_weights.T)
        y0 = y1 - self.dt*torch.matmul(F1,self.IRK_weights.T)

        l0 = torch.mean((self.x0s[:,0].reshape(-1,1) - x0)**2)
        l1 = torch.mean((self.x0s[:,1].reshape(-1,1) - y0)**2)

        return l0 + l1


    def plot_preds(self):
        xs, ys = self.model(self.x1s)
        # plt.scatter(self.x0s[:,0].detach() , xs[:,-1].detach(), label = 'x_pred')
        # plt.scatter(self.x0s[:,1].detach() , ys[:,-1].detach(), label = 'y_pred')
        plt.scatter(self.t_dat, xs[:,-1].detach(), label = 'x_pred' )
        plt.scatter(self.t_dat, ys[:,-1].detach(), label = 'y_pred' )
        plt.legend()



    def adam_train(self):
            steps = 1000
            self.model.train()
            for epoch in range(self.epochs):
                for step in range(steps):
                    def closure():
                        self.adam_optimizer.zero_grad()
                        loss = self.loss()
                        loss.backward()
                        return loss
                    self.adam_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, loss: {self.loss()}, beta_param: {self.beta}, gamma_param: {self.gamma}')
            self.plot_preds()
```

Copying the data generation code again

```
In [ ]: alpha = 0.25
        beta = 0.02
        delta = 0.25
        gamma = 2

        N = 200

        def pp_ode(state, t):
            x,y = state
            dx = alpha*x -beta*x*y
            dy = delta*x*y - y*gamma
            return [dx, dy]

        t = np.linspace(0,40,10000)
        sol = odeint(pp_ode,y0 =[10,5], t=t)
        data_rk = np.array([sol[:,0], sol[:,1]]).T

        # idx_t0 = 20
        # idx_t1 = 180
        # dt = t[idx_t1] - t[idx_t0]

        d_id = 1000
        ids0 = np.random.choice(len(t)-d_id, N)
        ids1 = ids0 + d_id

        dt = t[ids1[0]] - t[ids0[0]]
        t0s = t[ids0]
        #t1s = t[ids1]

        x1s = data_rk[ids1]
        x0s = data_rk[ids0]
        rk_samples = [x0s,x1s]

In [ ]: rk_test = rk_pp_pinn(dt, rk_samples, t_dat=t0s, q = 100)

In [ ]: rk_test.adam_train()

        Epoch 0, loss: 0.0422130711376667, beta_param: 0.02, gamma_param: 2
        Epoch 1, loss: 0.038544364273548126, beta_param: 0.02, gamma_param: 2
        Epoch 2, loss: 0.04269007220864296, beta_param: 0.02, gamma_param: 2
        Epoch 3, loss: 0.10343065857887268, beta_param: 0.02, gamma_param: 2
        Epoch 4, loss: 0.0331563726067543, beta_param: 0.02, gamma_param: 2
        Epoch 5, loss: 0.032884035259485245, beta_param: 0.02, gamma_param: 2
        Epoch 6, loss: 0.03010696917772293, beta_param: 0.02, gamma_param: 2
        Epoch 7, loss: 0.028697315603494644, beta_param: 0.02, gamma_param: 2
        Epoch 8, loss: 0.027399497106671333, beta_param: 0.02, gamma_param: 2
        Epoch 9, loss: 0.026172582060098648, beta_param: 0.02, gamma_param: 2
```



```
In [ ]:

In [ ]: dt

Out[ ]: 4.000400040004003

In [ ]:
```

Disease Informed Neural Networks Tests

```
In [ ]: import numpy as np
        from scipy.integrate import odeint
        import matplotlib.pyplot as plt

        import torch
        from torch.autograd import grad
        import torch.nn as nn
        from numpy import genfromtxt
        import torch.optim as optim
        import matplotlib.pyplot as plt
        import torch.nn.functional as F
        import time
```

```
In [ ]: class Neural_net(torch.nn.Module):
            def __init__(self, n_in = 1, n_out =1):
                super(Neural_net, self).__init__()

                self.tanh = torch.nn.Tanh()

                self.layer1 = torch.nn.Linear(n_in,20)
                self.layer2 = torch.nn.Linear(20,20)
                self.layer3 = torch.nn.Linear(20,20)
                self.layer_out = torch.nn.Linear(20,n_out)

            def forward(self, x):
                x = self.layer1(x)
                x = self.tanh(x)
                x = self.layer2(x)
                x = self.tanh(x)
                x = self.layer3(x)
                x = self.tanh(x)
                x = self.layer_out(x)

                return x
```

```
In [ ]: class sir_pinn:
            def __init__(self, epochs, data, c0, nde = 100):
                self.epochs = epochs
                self.model = Neural_net(n_out=3)
                self.domain = torch.linspace(0,int(max(data[0])),nde, requires_grad=True).reshape(-1,1)


                # self.beta_unconstr = torch.nn.Parameter(torch.randn(1))
                # self.gamma_unconstr = torch.nn.Parameter(torch.randn(1))

                self.beta_unconstr = torch.nn.Parameter(torch.tensor([0.1], dtype=torch.float))
                self.gamma_unconstr = torch.nn.Parameter(torch.tensor([0.1], dtype=torch.float))


                # Adding gamma and alpha to model trainable variables
                self.model.register_parameter(name='beta', param = self.beta_unconstr)
                self.model.register_parameter(name='gamma', param = self.gamma_unconstr)
                self.lbfgs_optimizer = torch.optim.LBFGS(params = self.model.parameters(), lr = 0.001,max_iter = 500)
                self.adam_optimizer = torch.optim.Adam(params = self.model.parameters(), lr = 0.0001)


                self.t_dat = torch.Tensor(data[0]).reshape(-1,1)
                self.S_dat = torch.Tensor(data[1])
                self.I_dat = torch.Tensor(data[2])
                self.R_dat = torch.Tensor(data[3])

                #find values for normalization

                self.maxes = {}
                self.mins = {}

                for id,d in enumerate((self.S_dat, self.I_dat, self.R_dat)):
                    self.maxes[id] = max(d)
                    self.mins[id] = min(d)

                self.N = self.maxes[0]

                #normalize
                self.S_norm = self.normalize(0, self.S_dat)
                self.I_norm = self.normalize(1, self.I_dat)
                self.R_norm = self.normalize(2, self.R_dat)

                #self.c0 = torch.tensor([max(self.S_norm), min(self.I_norm), min(self.R_norm)])
                self.c0 = torch.tensor([self.normalize(0, c0[0]), self.normalize(1, c0[1]),  self.normalize(2,c0[2])])


            #Constrain parameters to be in range
            @property
            def beta(self):
                return torch.tanh(self.beta_unconstr)

            @property
            def gamma(self):
                return torch.tanh(self.gamma_unconstr)

            def wrap_grad(self, f,x):
                return torch.autograd.grad(f,x,
                grad_outputs=torch.ones_like(x),
                retain_graph=True,
                create_graph=True)[0]


            def normalize(self, id, unnormed):
                return (unnormed - self.mins[id])/(self.maxes[id]- self.mins[id])

            def un_normalize(self, id, normed):
                return normed*(self.maxes[id] -self.mins[id])+ self.mins[id]


            def de_loss(self):
                pred = self.model(self.domain)
                S, I, R = (x.reshape(-1,1) for x in torch.unbind(pred, dim =1))

                dsir_dict ={}
                for id, val in zip(('dS', 'dI', 'dR'),(S, I, R)):
                    dsir_dict[id] = self.wrap_grad(val.reshape(-1,1), self.domain)

                dS, dI, dR = dsir_dict.values()

                ic = torch.mean((pred[0] - self.c0)**2)
                S = self.un_normalize(0, S) # un-normalizing, necessary for calcuclating derivative
                I = self.un_normalize(1, I)
                R = self.un_normalize(2, R)

                z1 = dS + ((self.beta / self.N) * S * I)  / (self.maxes[0] - self.mins[0]) # Rescaling derivatives want d(norm(s))
                z2 = dI - ((self.beta / self.N) * S * I - self.gamma * I) / (self.maxes[1] - self.mins[1])
                z3 = dR - (self.gamma * I ) / (self.maxes[2] - self.mins[2])

                return torch.mean(z1**2) + torch.mean(z2**2) +torch.mean(z3**2) + ic

            def data_loss(self):
                S, I, R =  torch.unbind(self.model(self.t_dat), dim =1)
                z1 = torch.mean((self.S_norm - S)**2)
                z2 = torch.mean((self.I_norm - I)**2)
                z3 = torch.mean((self.R_norm - R)**2)
                return z1 + z2 + z3
```

```
    def combined_loss(self):
        return self.de_loss() + self.data_loss()

    def model_call(self):
        return self.model(self.domain)

    def plot_preds(self):
        S,I,R = torch.unbind(self.model(self.domain), dim =1)
        S = self.un_normalize(0, S)
        I = self.un_normalize(1, I)
        R = self.un_normalize(2, R)
        plt.plot(self.domain.detach(), S.detach(), label = 's_pred')
        plt.plot(self.domain.detach(), I.detach(), label = 'i_pred')
        plt.plot(self.domain.detach(), R.detach(), label = 'r_pred')
        plt.scatter(self.t_dat, self.S_dat, label = 's_true')
        plt.scatter(self.t_dat, self.I_dat, label ='i_true' )
        plt.scatter(self.t_dat, self.R_dat, label ='r_true' )
        plt.grid()
        plt.legend()


    def lbfgs_train(self):
            self.model.train()
            for epoch in range(self.epochs):
                def closure():
                    self.lbfgs_optimizer.zero_grad()
                    loss = self.combined_loss()
                    loss.backward()
                    return loss
                self.lbfgs_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, training loss: {self.combined_loss()}, beta_param: {self.beta}, gamma_param: {self.gamma}')
            self.plot_preds()


    def adam_train(self):
            steps = 1000
            self.model.train()
            for epoch in range(self.epochs):
                for step in range(steps):
                    def closure():
                        self.adam_optimizer.zero_grad()
                        loss = self.combined_loss()
                        loss.backward()
                        return loss
                    self.adam_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, training loss: {self.combined_loss()}, beta_param: {self.beta}, gamma_param: {self.gamma}')
            self.plot_preds()
```

```
In [ ]: class sir_pinn_fixed_p(sir_pinn):
            """ Subclass of SIR pinn to solve without data.
            """
            def __init__(self, epochs, data, c0, beta, gamma):
                super().__init__(epochs, data, c0)

                self.beta_fixed = beta
                self.gamma_fixed = gamma


            def de_loss(self):
                pred = self.model(self.domain)
                S, I, R = (x.reshape(-1,1) for x in torch.unbind(pred, dim =1))

                dsir_dict ={}
                for id, val in zip(('dS', 'dI', 'dR'),(S, I, R)):
                    dsir_dict[id] = self.wrap_grad(val.reshape(-1,1), self.domain)

                dS, dI, dR = dsir_dict.values()

                ic = torch.mean((pred[:2] - self.c0)**2)
                S = self.un_normalize(0, S) # un-normalizing, necessary for calcuclating derivative
                I = self.un_normalize(1, I)
                R = self.un_normalize(2, R)

                z1 = dS + ((self.beta_fixed / self.N) * S * I)  / (self.maxes[0] - self.mins[0]) # Rescaling derivatives want d(norm(s))
                z2 = dI - ((self.beta_fixed / self.N) * S * I - self.gamma_fixed * I) / (self.maxes[1] - self.mins[1])
                z3 = dR - (self.gamma_fixed * I ) / (self.maxes[2] - self.mins[2])

                return torch.mean(z1**2) + torch.mean(z2**2) +torch.mean(z3**2) + ic

            def combined_loss(self):
                return self.de_loss()

            def plot_preds(self):
                S,I,R = torch.unbind(self.model(self.domain), dim =1)
                S = self.un_normalize(0, S)
                I = self.un_normalize(1, I)
                R = self.un_normalize(2, R)
                plt.plot(self.domain.detach(), S.detach(), label = 's_pred')
                plt.plot(self.domain.detach(), I.detach(), label = 'i_pred')
                plt.plot(self.domain.detach(), R.detach(), label = 'r_pred')
                # plt.scatter(self.t_dat, self.S_dat, label = 's_true')
                # plt.scatter(self.t_dat, self.I_dat, label ='i_true' )
                # plt.scatter(self.t_dat, self.R_dat, label ='r_true' )
                plt.grid()
                plt.legend()

            def adam_train(self):
                    steps = 1000
                    self.model.train()
                    for epoch in range(self.epochs):
                        for step in range(steps):
                            def closure():
                                self.adam_optimizer.zero_grad()
                                loss = self.combined_loss()
                                loss.backward()
                                return loss
                            self.adam_optimizer.step(closure=closure)
                        print(f'Epoch {epoch}, training loss: {self.combined_loss()}')
                    self.plot_preds()
```

```
In [ ]: class sir_neural_net(sir_pinn):
            """ Subclass of SIR pinn to solve without de-loss.
            So as a normal NN model...

            """
            def __init__(self, epochs, data, c0):
                super().__init__(epochs, data, c0)

            def combined_loss(self):
                return self.data_loss()

            def plot_preds(self):
                S,I,R = torch.unbind(self.model(self.domain), dim =1)
                S = self.un_normalize(0, S)
                I = self.un_normalize(1, I)
                R = self.un_normalize(2, R)
                plt.plot(self.domain.detach(), S.detach(), label = 's_pred')
                plt.plot(self.domain.detach(), I.detach(), label = 'i_pred')
                plt.plot(self.domain.detach(), R.detach(), label = 'r_pred')
                plt.scatter(self.t_dat, self.S_dat, label = 's_true')
                plt.scatter(self.t_dat, self.I_dat, label ='i_true' )
                plt.scatter(self.t_dat, self.R_dat, label ='r_true' )
                plt.grid()
                plt.legend()

            def adam_train(self):
```

```
        steps = 1000
        self.model.train()
        for epoch in range(self.epochs):
            for step in range(steps):
                def closure():
                    self.adam_optimizer.zero_grad()
                    loss = self.combined_loss()
                    loss.backward()
                    return loss
                self.adam_optimizer.step(closure=closure)
            print(f'Epoch {epoch}, training loss: {self.combined_loss()}')
        self.plot_preds()
```

Retrieving Data

```
In [ ]: #raisi_data = genfromtxt('../Gitfolder/DINN/COVID_Tutorial.csv', delimiter=',') #in the form of [t,S,I,D,R]

        b005_g002_i001 = genfromtxt('Data/SIR_generated/b005_g002_i001', delimiter=',').T #1
        b005_g002_i050 = genfromtxt('Data/SIR_generated/b005_g002_i050', delimiter=',').T #2
        b050_g005_i050 = genfromtxt('Data/SIR_generated/b050_g005_i050', delimiter=',').T #3
        b070_g016_i001 = genfromtxt('Data/SIR_generated/b070_g016_i001', delimiter=',').T #4
        b070_g016_i020 = genfromtxt('Data/SIR_generated/b070_g016_i020', delimiter=',').T #5
```

```
In [ ]: def sampling(data, size= 200):
            ids = np.random.choice(range(data.shape[1]),size = size)
            sample_data = data[:,ids]
            traind, testd = (sample_data[:,:-sample_data.shape[1]//2], sample_data[:,sample_data.shape[1]//2:])
            return traind, testd
```

```
In [ ]: b005_g002_i001_train, b005_g002_i001_test = sampling(b005_g002_i001)
        b005_g002_i050_train, b005_g002_i050_test = sampling(b005_g002_i050)
        b050_g005_i050_train, b050_g005_i050_test = sampling(b050_g005_i050)
        b070_g016_i001_train, b070_g016_i001_test = sampling(b070_g016_i001)
        b070_g016_i020_train, b070_g016_i020_test = sampling(b070_g016_i020)
```

```
In [ ]: b005_g002_i001_C0 = b005_g002_i001[1:][:,0]
        b005_g002_i050_C0 = b005_g002_i050[1:][:,0]
        b050_g005_i050_C0 = b050_g005_i050[1:][:,0]
        b070_g016_i001_C0 = b070_g016_i001[1:][:,0]
        b070_g016_i020_C0 = b070_g016_i020[1:][:,0]
```

```
In [ ]: def test_loss(inst, test_data):
            """ computes MSE over test data
            """
            t = torch.Tensor(test_data[0]).reshape(-1,1)
            S = inst.normalize(0,torch.Tensor(test_data[1]))
            I = inst.normalize(1, torch.Tensor(test_data[2]))
            R = inst.normalize(2, torch.Tensor(test_data[3]))

            s_pred, i_pred, r_pred = torch.unbind(inst.model(t), dim =1)

            z0 = torch.mean((S-s_pred)**2)
            z1 = torch.mean((I-i_pred)**2)
            z2 = torch.mean((R-r_pred)**2)

            return z0 + z1 + z2
```

Testing scheme:

Have generated 5 sets of simulation data of varying underlying parameters β, γ and varying (I_0, S_0). R_0 always assumed to be zero.

Want to compare the performance of:

```
~ "Many" datapoints and "many" de-domain points.           Hypothesis: Performs well, can be considered best possible model.

~ "Few" datapoints and "many" de-domain points.            Hypothesis: Should also perform "well",

~ "Few" datapoints only                                    Hypothesis: Should perform "worse" than the two above

~ "Many" de-domaion points only, and fixed parameters      Hypothesis: Should be problematic: difficult to fit properly


(~ maybe also: only "few de-domain points", and fixed params) Hypotheis:  Also difficult?

(~ Try "few" data-points "many" domain points for real-world covid data) Hypothesis: Will find some parameter.
```

(Note: DE-domain: t points used when calculating differential equation loss under training)

For each of these models the following metrics shall be recorded:

```
10k steps (10 epochs, 1000 steps) training accuracy
10k steps test accuracy: MSE.
Save10k plot

TUS (Train Until Satisfied) - training accuracy MSE
TUS - Test accuracy MSE
Save TUS plot.
```

Round 1

Parameters: N = 1,

I0 = 0.01,

S0= 0.99

β = 0.05 γ = 0.02

sir-pinn instance naming scheme, examples:

```
r1_da100_de100 = Round 1,  100 datapoints, 100 de-domain points.

r1_da010_de100 =Round 1, 10 datapoints, 100 de-domain-points

r1_da010_de000 = you get it
```

```
In [ ]: # Plot of true solution
        plt.plot(b005_g002_i001[0], b005_g002_i001[1], label = 'S')
        plt.plot(b005_g002_i001[0], b005_g002_i001[2], label = 'I')
        plt.plot(b005_g002_i001[0], b005_g002_i001[3], label = 'R')
        plt.legend()
```

```
In [ ]: r1_da100_de100 = sir_pinn(epochs=10,data= b005_g002_i001_train, c0 = b005_g002_i001_C0)
        r1_da010_de100 = sir_pinn(epochs=10,data= b005_g002_i001_train[:,:10], c0 = b005_g002_i001_C0) # _train variabels are randomly selected
        r1_da010_de000 = sir_neural_net(epochs=10,data= b005_g002_i001_train[:,:10], c0 = b005_g002_i001_C0)
        r1_da000_de100 = sir_pinn_fixed_p(epochs=10, data= b005_g002_i001_train, c0 =b005_g002_i001_C0, beta = 0.05, gamma=0.02)
```

R1 "Many" datapoints and "many" de-domain points.

```
In [ ]: print('###### r1_da100_de100 10 epochs Run #######')
        r1_da100_de100.adam_train()
```

```
         print(f'Test loss {test_loss(r1_da100_de100,b005_g002_i001_test)}')
```

```
In [ ]: # Note, To get TUS values, run cell until satisfied
        print('###### r1_da100_de100 TUS Run #######')
        r1_da100_de100.adam_train()
        print(f'Test loss {test_loss(r1_da100_de100,b005_g002_i001_test)}')
```

R1 "Few" datapoints and "many" de-domain points.

```
In [ ]: print('###### r1_da010_de100 10 epochs Run #######')
        r1_da010_de100.adam_train()
        print(f'Test loss {test_loss(r1_da010_de100,b005_g002_i001_test)}')
```

```
In [ ]: # To get TUS values, run cell until satisfied
        print('###### r1_da010_de100 TUS Run #######')
        r1_da010_de100.adam_train()
        print(f'Test loss {test_loss(r1_da010_de100,b005_g002_i001_test)}')
```

R1 "Few" datapoints only

```
In [ ]: print('###### r1_da010_de000 10 epochs Run #######')
        r1_da010_de000.adam_train()
        print(f'Test loss {test_loss(r1_da010_de000,b005_g002_i001_test)}')
```

```
In [ ]: print('###### r1_da010_de000 TUS Run #######')
        r1_da010_de000.adam_train()
        print(f'Test loss {test_loss(r1_da010_de000,b005_g002_i001_test)}')
```

R1 "Many" de-domain points only, and fixed parameters

```
In [ ]: print('###### r1_da000_de100 10 epochs Run #######')
        r1_da000_de100.adam_train()
        print(f'Test loss {test_loss(r1_da000_de100,b005_g002_i001_test)}')
```

```
In [ ]: print('###### r1_da000_de100 TUS Run #######')
        r1_da000_de100.adam_train()
        print(f'Test loss {test_loss(r1_da000_de100,b005_g002_i001_test)}')
```

Round 2

Parameters: N = 1,

I0 = 0.5, S0= 0.5

β = 0.05 γ = 0.02

```
In [ ]: ### True solution plot

        plt.plot(b005_g002_i050[0], b005_g002_i050[1])
        plt.plot(b005_g002_i050[0], b005_g002_i050[2])
        plt.plot(b005_g002_i050[0], b005_g002_i050[3])
```

```
In [ ]: r2_da100_de100 = sir_pinn(epochs=10,data= b005_g002_i050_train, c0 = b005_g002_i050_C0)
        r2_da010_de100 = sir_pinn(epochs=10,data= b005_g002_i050_train[:,:10], c0 = b005_g002_i050_C0)
        r2_da010_de000 = sir_neural_net(epochs=10,data= b005_g002_i050_train[:,:10], c0 = b005_g002_i050_C0)
        r2_da000_de100 = sir_pinn_fixed_p(epochs=10, data= b005_g002_i050_train, c0 =b005_g002_i050_C0, beta = 0.05, gamma=0.02)
```

```
In [ ]:
```

R2 "Many" datapoints and "many" de-domain points.

```
In [ ]: print('###### r2_da100_de100 10 epochs Run #######')
        r2_da100_de100.adam_train()
        print(f'Test loss {test_loss(r2_da100_de100,b005_g002_i050_test)}')
```

```
In [ ]: print('###### r2_da100_de100 TUS Run #######')
        r2_da100_de100.adam_train()
        print(f'Test loss {test_loss(r2_da100_de100,b005_g002_i050_test)}')
```

R2 "Few" datapoints and "many" de-domain points.

```
In [ ]: print('###### r2_da010_de100 10 epochs Run #######')
        r2_da010_de100.adam_train()
        print(f'Test loss {test_loss(r2_da010_de100,b005_g002_i050_test)}')
```

```
In [ ]: print('###### r2_da010_de100 TUS Run #######')
        r2_da010_de100.adam_train()
        print(f'Test loss {test_loss(r2_da010_de100,b005_g002_i050_test)}')
```

R2 "Few" datapoints only

```
In [ ]: print('###### r2_da010_de000 10 epochs Run #######')
        r2_da010_de000.adam_train()
        print(f'Test loss {test_loss(r2_da010_de000,b005_g002_i050_test)}')
```

```
In [ ]: print('###### r2_da010_de000 TUS Run #######')
        r2_da010_de000.adam_train()
        print(f'Test loss {test_loss(r2_da010_de000,b005_g002_i050_test)}')
```

R2 "Many" de-domain points only, and fixed parameters

```
In [ ]: print('###### r2_da000_de100 10 epochs Run #######')
        r2_da000_de100.adam_train()
        print(f'Test loss {test_loss(r2_da000_de100,b005_g002_i050_test)}')
```

```
In [ ]: print('###### r2_da000_de100 TUS Run #######')
        r2_da000_de100.adam_train()
        print(f'Test loss {test_loss(r2_da000_de100,b005_g002_i050_test)}')
```

Round 3

Parameters:

N = 1,

I0 = 0.50,

S0= 0.50

β = 0.50 γ = 0.05

```
In [ ]: #Plot of true solution
        plt.plot(b050_g005_i050[0], b050_g005_i050[1])
        plt.plot(b050_g005_i050[0], b050_g005_i050[2])
        plt.plot(b050_g005_i050[0], b050_g005_i050[3])
```

In [ ]:
```
r3_da100_de100 = sir_pinn(epochs=10,data= b005_g002_i050_train, c0 = b050_g005_i050_C0)
r3_da010_de100 = sir_pinn(epochs=10,data= b005_g002_i050_train[:,:10], c0 = b050_g005_i050_C0)
r3_da010_de000 = sir_neural_net(epochs=10,data= b005_g002_i050_train[:,:10], c0 = b050_g005_i050_C0)
r3_da000_de100 = sir_pinn_fixed_p(epochs=10, data= b005_g002_i050_train, c0 =b050_g005_i050_C0, beta = 0.50, gamma=0.05)
```

R3 "Many" datapoints and "many" de-domain points.

In [ ]:
```
print('###### r3_da100_de100 10 epochs Run #######')
r3_da100_de100.adam_train()
print(f'Test loss {test_loss(r3_da100_de100,b050_g005_i050_test)}')
```

In [ ]:
```
print('###### r3_da100_de100 TUS Run #######')
r3_da100_de100.adam_train()
print(f'Test loss {test_loss(r3_da100_de100,b050_g005_i050_test)}')
```

R3 "Few" datapoints and "many" de-domain points.

In [ ]:
```
print('###### r3_da010_de100 10 epochs Run #######')
r3_da010_de100.adam_train()
print(f'Test loss {test_loss(r3_da010_de100,b050_g005_i050_test)}')
```

In [ ]:
```
print('###### r3_da010_de100 TUS Run #######')
r3_da010_de100.adam_train()
print(f'Test loss {test_loss(r3_da010_de100,b050_g005_i050_test)}')
```

R3 "Few" datapoints only

In [ ]:
```
print('###### r3_da010_de000 10 epochs Run #######')
r3_da010_de000.adam_train()
print(f'Test loss {test_loss(r3_da010_de000,b050_g005_i050_test)}')
```

In [ ]:
```
print('###### r3_da010_de000 TUS Run #######')
r3_da010_de000.adam_train()
print(f'Test loss {test_loss(r3_da010_de000,b050_g005_i050_test)}')
```

R3 "Many" de-domain points only, and fixed parameters

In [ ]:
```
print('###### r3_da000_de100 10 epochs Run #######')
r3_da000_de100.adam_train()
print(f'Test loss {test_loss(r3_da000_de100,b050_g005_i050_test)}')
```

In [ ]:
```
print('###### r3_da000_de100 TUS Run #######')
r3_da000_de100.adam_train()
print(f'Test loss {test_loss(r3_da000_de100,b050_g005_i050_test)}')
```

Round 4

Parameters:

N = 1,

I0 = 0.01,

S0= 0.99

β = 0.70 γ = 0.16

In [ ]:
```
# Plot of true solution

plt.plot(b070_g016_i001[0], b070_g016_i001[1])
plt.plot(b070_g016_i001[0], b070_g016_i001[2])
plt.plot(b070_g016_i001[0], b070_g016_i001[3])
```

In [ ]:
```
r4_da100_de100 = sir_pinn(epochs=10,data= b070_g016_i001_train, c0 = b070_g016_i001_C0)
r4_da010_de100 = sir_pinn(epochs=10,data= b070_g016_i001_train[:,:10], c0 = b070_g016_i001_C0)
r4_da010_de000 = sir_neural_net(epochs=10,data= b070_g016_i001_train[:,:10], c0 = b070_g016_i001_C0)
r4_da000_de100 = sir_pinn_fixed_p(epochs=10, data= b070_g016_i001_train, c0 =b070_g016_i001_C0, beta = 0.70, gamma=0.16)
```

R4 "Many" datapoints and "many" de-domain points.

In [ ]:
```
print('###### r4_da100_de100 10 epochs Run #######')
r4_da100_de100.adam_train()
print(f'Test loss {test_loss(r4_da100_de100,b070_g016_i001_test)}')
```

In [ ]:
```
print('###### r4_da100_de100 TUS Run #######')
r4_da100_de100.adam_train()
print(f'Test loss {test_loss(r4_da100_de100,b070_g016_i001_test)}')
```

R4 "Few" datapoints and "many" de-domain points.

In [ ]:
```
print('###### r4_da010_de100 10 epochs Run #######')
r4_da010_de100.adam_train()
print(f'Test loss {test_loss(r4_da010_de100,b070_g016_i001_test)}')
```

In [ ]:
```
print('###### r4_da010_de100 TUS Run #######')
r4_da010_de100.adam_train()
print(f'Test loss {test_loss(r4_da010_de100,b070_g016_i001_test)}')
```

R4 "Few" datapoints only

In [ ]:
```
print('###### r4_da010_de000 10 epochs Run #######')
r4_da010_de000.adam_train()
print(f'Test loss {test_loss(r4_da010_de000,b070_g016_i001_test)}')
```

In [ ]:
```
print('###### r4_da010_de000 TUS Run #######')
r4_da010_de000.adam_train()
print(f'Test loss {test_loss(r4_da010_de100,b070_g016_i001_test)}')
```

R4 "Many" de-domain points only, and fixed parameters

In [ ]:
```
print('###### r4_da000_de100 10 epochs Run #######')
r4_da000_de100.adam_train()
print(f'Test loss {test_loss(r4_da000_de100,b070_g016_i001_test)}')
```

In [ ]:
```
print('###### r4_da000_de100 TUS Run #######')
r4_da000_de100.adam_train()
print(f'Test loss {test_loss(r4_da000_de100,b070_g016_i001_test)}')
```

Round 5

Parameters:

N = 1,

I0 = 0.20,

S0= 0.80

β = 0.70 γ = 0.16

```
In [ ]: ## True solution plot
        plt.plot(b070_g016_i020[0], b070_g016_i020[1])
        plt.plot(b070_g016_i020[0], b070_g016_i020[2])
        plt.plot(b070_g016_i020[0], b070_g016_i020[3])
```

```
In [ ]: r5_da100_de100 = sir_pinn(epochs=10,data= b070_g016_i020_train, c0 = b070_g016_i020_C0)
        r5_da010_de100 = sir_pinn(epochs=10,data= b070_g016_i020_train[:,:10], c0 = b070_g016_i001_C0)
        r5_da010_de000 = sir_neural_net(epochs=10,data= b070_g016_i020_train[:,:10], c0 = b070_g016_i020_C0)
        r5_da000_de100 = sir_pinn_fixed_p(epochs=10, data= b070_g016_i020_train, c0 = b070_g016_i020_C0, beta = 0.70, gamma=0.16)
```

R5 "Many" datapoints and "many" de-domain points.

```
In [ ]: print('###### r5_da100_de100 10 epochs Run #######')
        r5_da100_de100.adam_train()
        print(f'Test loss {test_loss(r5_da100_de100,b070_g016_i020_test)}')
```

```
In [ ]: print('###### r5_da100_de100 TUS Run #######')
        r5_da100_de100.adam_train()
        print(f'Test loss {test_loss(r5_da100_de100,b070_g016_i020_test)}')
```

R5 "Few" datapoints and "many" de-domain points.

```
In [ ]: print('###### r5_da010_de100 10 epochs Run #######')
        r5_da010_de100.adam_train()
        print(f'Test loss {test_loss(r5_da010_de100,b070_g016_i020_test)}')
```

```
In [ ]: print('###### r5_da010_de100 TUS Run #######')
        r5_da010_de100.adam_train()
        print(f'Test loss {test_loss(r5_da010_de100,b070_g016_i020_test)}')
```

R5 "Few" datapoints only

```
In [ ]: print('###### r5_da010_de000 10 epochs Run #######')
        r5_da010_de000.adam_train()
        print(f'Test loss {test_loss(r5_da010_de000,b070_g016_i020_test)}')
```

```
In [ ]: print('###### r5_da010_de000 TUS Run #######')
        r5_da010_de000.adam_train()
        print(f'Test loss {test_loss(r5_da010_de100,b070_g016_i020_test)}')
```

R5 "Many" de-domain points only, and fixed parameters

```
In [ ]: print('###### r5_da000_de100 10 epochs Run #######')
        r5_da000_de100.adam_train()
        print(f'Test loss {test_loss(r5_da000_de100,b070_g016_i020_test)}')
```

```
In [ ]: print('###### r5_da000_de100 TUS Run #######')
        r5_da000_de100.adam_train()
        print(f'Test loss {test_loss(r5_da000_de100,b070_g016_i020_test)}')
```

```
In [ ]:
```

Parameter discovery on noisy data.

```
In [ ]: b020_g005_i020n070 = genfromtxt('Data/SIR_generated/b020_g005_i020n070', delimiter=',').T
        b020_g005_i020n070_train, b020_g005_i020n070_test = sampling(b020_g005_i020n070)
        b020_g005_i020n070_C0 = b020_g005_i020n070[1:][:,0]
```

Round 6

β = 0.20

γ = 0.05

$N = 1$

$S_0 = 0.80$

$I_0 = 0.20$

$noise = 0.7\sigma$

```
In [ ]: r6_da100_de100 = sir_pinn(epochs=10,data= b020_g005_i020n070_train, c0 =b020_g005_i020n070_C0)
        r6_da010_de100 = sir_pinn(epochs=10,data= b020_g005_i020n070_train[:,:10], c0 = b020_g005_i020n070_C0)
```

100 Data-points 100De-points

```
In [ ]: print('###### r6_da100_de100 10 epochs Run #######')
        r6_da100_de100.adam_train()
        print(f'Test loss {test_loss(r6_da100_de100,b020_g005_i020n070_test)}')
```

```
In [ ]: print('###### r6_da100_de100 TUS Run #######')
        r6_da100_de100.adam_train()
        print(f'Test loss {test_loss(r6_da100_de100,b020_g005_i020n070_test)}')
```

10 Data-points 100De-points

```
In [ ]: print('###### r6_da100_de100 10 epochs Run #######')
        r6_da010_de100.adam_train()
        print(f'Test loss {test_loss(r6_da010_de100,b020_g005_i020n070_test)}')
```

```
In [ ]: print('###### r6_da010_de100 TUS Run #######')
        r6_da010_de100.adam_train()
        print(f'Test loss {test_loss(r6_da010_de100,b020_g005_i020n070_test)}')
```

```
In [ ]: r6_da010_de1000 = sir_pinn(epochs=10,data= b020_g005_i020n070_train[:,:10], c0 = b020_g005_i020n070_C0, nde= 1000)
```

```
In [ ]: print('###### r6_da010_de1000 10 epochs Run #######')
        r6_da010_de1000.adam_train()
        print(f'Test loss {test_loss(r6_da010_de1000,b020_g005_i020n070_test)}')
```

```
In [ ]: print('###### r6_da010_de1000 TUS Run #######')
        r6_da010_de1000.adam_train()
        print(f'Test loss {test_loss(r6_da010_de1000,b020_g005_i020n070_test)}')
```

Round 7

β = 0.20

γ = 0.05

$N = 1$

$S_0 = 0.80$

$I_0 = 0.20$

$noise = 1.4\sigma$

```
In [ ]: b020_g005_i020n140 = genfromtxt('Data/SIR_generated/b020_g005_i020n140', delimiter=',').T
        b020_g005_i020n140_train, b020_g005_i020n140_test = sampling(b020_g005_i020n140)
        b020_g005_i020n140_C0 = b020_g005_i020n140[1:][:,0]
```

```
In [ ]: r7_da100_de100 = sir_pinn(epochs=10,data= b020_g005_i020n140_train, c0 =b020_g005_i020n140_C0)
        r7_da010_de100 = sir_pinn(epochs=10,data= b020_g005_i020n140_train[:,:10], c0 = b020_g005_i020n140_C0)
```

100Da-points 100De-points

```
In [ ]: print('###### r7_da100_de100 10 epochs Run #######')
        r7_da100_de100.adam_train()
        print(f'Test loss {test_loss(r7_da100_de100,b020_g005_i020n140_test)}')
```

```
In [ ]: print('###### r7_da100_de100 TUS Run #######')
        r7_da100_de100.adam_train()
        print(f'Test loss {test_loss(r7_da100_de100,b020_g005_i020n140_test)}')
```

10 Da-points 100De-points

```
In [ ]: print('###### r7_da010_de100 10 epochs Run #######')
        r7_da010_de100.adam_train()
        print(f'Test loss {test_loss(r7_da010_de100,b020_g005_i020n140_test)}')
```

```
In [ ]: print('###### r7_da010_de100 TUS Run #######')
        r7_da010_de100.adam_train()
        print(f'Test loss {test_loss(r7_da010_de100,b020_g005_i020n140_test)}')
```

10 Da-points 1000De-points

```
In [ ]: #r7_da100_de1000 = sir_pinn(epochs=10,data= b020_g005_i020n140_train, c0 =b020_g005_i020n140_C0, nde=1000)
        r7_da010_de1000 = sir_pinn(epochs=10,data= b020_g005_i020n140_train[:,:10], c0 = b020_g005_i020n140_C0, nde= 1000)
```

```
In [ ]: print('###### r7_da010_de1000 10 epochs Run #######')
        r7_da010_de1000.adam_train()
        print(f'Test loss {test_loss(r7_da010_de1000,b020_g005_i020n140_test)}')
```

```
In [ ]: print('###### r7_da010_de1000 TUS Run #######')
        r7_da010_de1000.adam_train()
        print(f'Test loss {test_loss(r7_da010_de1000,b020_g005_i020n140_test)}')
```

```
In [ ]:
```

In [ ]:
```python
import torch
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
from scipy.special import gamma
```

In [ ]:
```python
HU=31.74

patient1 = # Hidden
patient2 = # Hidden
patient3 = # Hidden
patient4 = # Hidden
patient5 = # Hidden
patient6 = # Hidden
```

In [ ]:
```python
p1_sim = np.genfromtxt('Data/blood_generated/p1q4218.69', delimiter=',')
p2_sim = np.genfromtxt('Data/blood_generated/p2q3513.35', delimiter=',')
p3_sim = np.genfromtxt('Data/blood_generated/p3q4310.52', delimiter=',')
p4_sim = np.genfromtxt('Data/blood_generated/p4q4510.85', delimiter=',')
p5_sim = np.genfromtxt('Data/blood_generated/p5q3104.34', delimiter=',')
p6_sim = np.genfromtxt('Data/blood_generated/p6q3905.67', delimiter=',')
```

In [ ]:
```python
vref1 = # Hidden
vref2 = # Hidden
vref3 = # Hidden
vref4 = # Hidden
vref5 = # Hidden
vref6 = # Hidden
```

In [ ]:
```python
qref1 = # Hidden
qref2 = # Hidden
qref3 = # Hidden
qref4 = # Hidden
qref5 = # Hidden
qref6 = # Hidden
```

In [ ]:

In [ ]:
```python
class Neural_net(torch.nn.Module):
    def __init__(self, n_in = 1, n_out =1):
        super(Neural_net, self).__init__()

        self.tanh = torch.nn.Tanh()

        self.layer1 = torch.nn.Linear(n_in,20)
        self.layer2 = torch.nn.Linear(20,20)
        self.layer3 = torch.nn.Linear(20,20)
        self.layer_out = torch.nn.Linear(20,n_out)

    def forward(self, x):
        x = self.layer1(x)
        x = self.tanh(x)
        x = self.layer2(x)
        x = self.tanh(x)
        x = self.layer3(x)
        x = self.tanh(x)
        x = self.layer_out(x)

        return x
```

In [ ]:
```python
class blood_1pinn:
    def __init__(self, epochs, data, nde, params = {}, gamma = None):
        self.epochs = epochs
        self.model = Neural_net(n_out=1)
        self.domain = torch.linspace(0,100, nde, requires_grad=True).reshape(-1,1)

        ###  scaling params
        self.params_unconstr ={}
        for prm, init in params.items():
            self.params_unconstr[prm] = torch.nn.Parameter(torch.tensor([init]))
            self.model.register_parameter(name=prm, param = self.params_unconstr[prm])

        self.const = torch.tensor([1.], dtype = torch.float)


        #optimizers
        self.lbfgs_optimizer = torch.optim.LBFGS(params = self.model.parameters(), lr = 0.001,max_iter = 500)
        self.adam_optimizer = torch.optim.Adam(params = self.model.parameters(), lr = 0.001)


        patient_data, sim_data, vref ,qref  = data

        self.vref = vref
        self.qc =self.m_inj(np.linspace(0,100,nde))
        self.Q_ref = torch.tensor(qref, dtype=torch.float, requires_grad=False).reshape(-1,1)

        #### Data

        if gamma: ## Using a gamma variate fit of data at end-compartment instead of pure data-points
            popt, pcov = curve_fit(self.scaled_gamma, patient_data[0], patient_data[1], bounds = ((1000,1,1),(10000,10,10)))
            self.c_patient = torch.tensor(self.scaled_gamma(np.linspace(0,100,100),popt[0],popt[1], popt[2]), dtype=torch.float).reshape(-1,1)
            self.t_patient = torch.linspace(0,100,100).reshape(-1,1)

        else:
            self.c_patient = torch.tensor(patient_data[1], dtype = torch.float).reshape(-1,1)
            self.t_patient = torch.tensor(patient_data[0], dtype=torch.float).reshape(-1,1)




        sample_ids = np.arange(start=0, stop=len(sim_data[7]), step=len(sim_data[7])//nde)
        self.c7 = torch.tensor(sim_data[7][sample_ids], dtype=torch.float).reshape(-1,1)

        self.patient_data = patient_data

        self.cp_max = max(self.c_patient)
        self.cp_min = min(self.c_patient)
        self.c_patient_norm = (self.c_patient - self.cp_min)/(self.cp_max-self.cp_min)


    #Constrain parameters to be in range

    @property
    def sq_param(self):
        if 'sq' in self.params_unconstr:
            return self.params_unconstr['sq']
        else:
            return self.const

    @property
    def sv_param(self):
        if 'sv' in self.params_unconstr:
            return self.params_unconstr['sv']
        else:
            return self.const

    @property
    def sc_param(self):
        if 'sc' in self.params_unconstr:
            return self.params_unconstr['sc']
        else:
            return self.const
    #------------- Gamma variate
    def scaled_gamma(self, x,c, k, theta):
            frac = 1/(gamma(k)*theta**(k))
            return c*frac*x**(k-1)*np.exp(-(x/theta))

    # ------------
    def m_inj(self,t,ti=[0,3.33,3.33,6.67,6.67],qi=[6,6,6,6,0],ci=[350,350,0,0,0]):
        ti=np.array(ti)
        qi=np.array(qi)
        # ci=np.array(ci)
        # m = qi*ci
        return torch.tensor(np.interp(t,ti,qi),dtype = torch.float)


    def wrap_grad(self, f,x):
        return torch.autograd.grad(f,x,
        grad_outputs=torch.ones_like(x),
        retain_graph=True,
        create_graph=True)[0]


    def de_loss(self):
        c_pred = self.model(self.domain)

        c =c_pred*(self.cp_max - self.cp_min) + self.cp_min
        dup = self.wrap_grad(c_pred, self.domain)

        norm_fac = (self.cp_max -self.cp_min)

        Q = self.qc + self.Q_ref*self.sq_param
        c7 = self.c7*self.sc_param

        z = dup -(Q*c7-Q*c)/(self.vref*self.sv_param*norm_fac)
        return torch.mean(z**2) + c_pred[0]**2


    def data_loss(self):
        u7_pred = self.model(self.t_patient)
        return torch.mean((u7_pred - self.c_patient_norm)**2)


    def combined_loss(self):
        return self.data_loss() #self.de_loss() + self.data_loss()

    def model_call(self):
        return self.model(self.domain)

    def plot_preds(self):
        pred = self.model(self.domain)
        plt.plot(self.domain.detach(),pred.detach()*(self.cp_max -self.cp_min) + self.cp_min)
        plt.scatter(self.patient_data[0], self.patient_data[1])
        plt.grid()
        #plt.legend()


    def lbfgs_train(self):
            self.model.train()
            for epoch in range(self.epochs):
                def closure():
                    self.lbfgs_optimizer.zero_grad()
                    loss = self.combined_loss()
                    loss.backward()
                    return loss
                self.lbfgs_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, training loss: {self.combined_loss()}')
            self.plot_preds()


    def adam_train(self):
            steps = 1000
            for epoch in range(self.epochs):
                for step in range(steps):
                    def closure():
                        self.adam_optimizer.zero_grad()
                        loss = self.combined_loss()
                        loss.backward()
                        return loss
                    self.adam_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, training loss: {self.combined_loss().item()} de_loss: {self.de_loss().item()}, data_loss {self.data_loss()}, sq {self.sq_param.item()} sv: {self.sv_
            self.plot_preds()
```

```
In [ ]:
```

```
In [ ]: test = blood_1pinn(epochs = 20, data = (patient4,p4_sim, vref4[7], qref4),params = {'sv': 1., 'sc':1.}, nde = 100, gamma=False)
```

```
In [ ]: # Model configs:

        qcv ={'sq':1., 'sv':1. , 'sc':1.}
        qv = {'sq':1., 'sv':1. }
        qc = {'sq':1. , 'sc':1.}
        q = {'sq':1.}
        vc = {'sv':1. , 'sc':1.}
        c = { 'sc':1.}
        v = {'sv':1.}
        no_params = {}
```

```
In [ ]: p1_data = (patient1,p1_sim, vref1[7], qref1)

        p1_none = blood_1pinn(epochs = 20, data = p1_data,params =no_params, nde = 100, gamma=False)
```

```
In [ ]: p1_none.adam_train()
        # plt.gca().axes.get_yaxis().set_visible(False)
        # plt.gca().axes.get_xaxis().set_visible(False)
```

Patient 1

```
In [ ]: p1_data = (patient1,p1_sim, vref1[7], qref1)

        p1_none = blood_1pinn(epochs = 20, data = p1_data,params =no_params, nde = 100, gamma=False)
        p1_none_gamma = blood_1pinn(epochs = 20, data = p1_data,params =no_params, nde = 100, gamma=True)

        p1_qcv = blood_1pinn(epochs = 20, data = p1_data,params =qcv, nde = 100, gamma=False)
        p1_qcv_gamma = blood_1pinn(epochs = 20, data = p1_data,params =qcv, nde = 100, gamma=True)

        p1_qv = blood_1pinn(epochs = 20, data = p1_data,params =qv, nde = 100, gamma=False)
        p1_qv_gamma = blood_1pinn(epochs = 20, data = p1_data,params =qv, nde = 100, gamma=True)

        p1_qc = blood_1pinn(epochs = 20, data = p1_data,params =qc, nde = 100, gamma=False)
        p1_qc_gamma = blood_1pinn(epochs = 20, data = p1_data,params =qc, nde = 100, gamma=True)

        p1_q = blood_1pinn(epochs = 20, data = p1_data,params =q, nde = 100, gamma=False)
        p1_q_gamma = blood_1pinn(epochs = 20, data = p1_data,params =q, nde = 100, gamma=True)

        p1_vc = blood_1pinn(epochs = 20, data = p1_data,params =vc, nde = 100, gamma=False)
        p1_vc_gamma = blood_1pinn(epochs = 20, data = p1_data,params =vc, nde = 100, gamma=True)

        p1_c = blood_1pinn(epochs = 20, data = p1_data,params =c, nde = 100, gamma=False)
        p1_c_gamma = blood_1pinn(epochs = 20, data = p1_data,params =c, nde = 100, gamma=True)

        p1_v = blood_1pinn(epochs = 20, data = p1_data,params =v, nde = 100, gamma=False)
        p1_v_gamma = blood_1pinn(epochs = 20, data = p1_data,params =v, nde = 100, gamma=True)
```

```
In [ ]: p1_none.adam_train()
In [ ]: p1_none_gamma.adam_train()
```

```
In [ ]: p1_qcv.adam_train()

In [ ]: p1_qcv_gamma.adam_train()

In [ ]: p1_qv.adam_train()

In [ ]: p1_qv_gamma.adam_train()

In [ ]: p1_qc.adam_train()

In [ ]: p1_qc_gamma.adam_train()

In [ ]: p1_q.adam_train()

In [ ]: p1_q_gamma.adam_train()

In [ ]: p1_vc.adam_train()

In [ ]: p1_vc_gamma.adam_train()

In [ ]: p1_c.adam_train()

In [ ]: p1_c_gamma.adam_train()

In [ ]: p1_v.adam_train()

In [ ]: p1_v_gamma.adam_train()

In [ ]:
```

Implementation and test of PiNN over 8 free compartments of the test bolus model

```
In [ ]: import torch
        import matplotlib.pyplot as plt
        import numpy as np
        from scipy.integrate import odeint
        from scipy.optimize import curve_fit
        from scipy.special import gamma
```

```
c:\Users\Lars\anaconda3\envs\torchenv\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/s
table/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
In [ ]: ## patient data:
        HU=31.74

        patient1 = # Hidden
        patient2 = # Hidden
        patient3 = #...
        patient4 =
        patient5 =
```

```
In [ ]: p1_sim = np.genfromtxt('Data/blood_generated/p1q4218.69', delimiter=',')
        p2_sim = np.genfromtxt('Data/blood_generated/p2q3513.35', delimiter=',')
        p3_sim = np.genfromtxt('Data/blood_generated/p3q4310.52', delimiter=',')
        p4_sim = np.genfromtxt('Data/blood_generated/p4q4510.85', delimiter=',')
        p5_sim = np.genfromtxt('Data/blood_generated/p5q3104.34', delimiter=',')
        p6_sim = np.genfromtxt('Data/blood_generated/p6q3905.67', delimiter=',')
```

```
In [ ]: vref1 = # Hidden
        vref2 =
        vref3 =
        vref4 =
        vref5 =
        vref6 =
```

```
In [ ]: qref1 = 4218.69/60
        qref2 = 3513.35/60
        qref3 = 4310.52/60
        qref4 = 4510.85/60
        qref5 = 3104.34/60
        qref6 = 3905.67/60
```

```
In [ ]:
```

```
In [ ]: class Neural_net(torch.nn.Module):
            def __init__(self, n_in = 1, n_out =1):
                super(Neural_net, self).__init__()

                self.tanh = torch.nn.Tanh()

                self.layer1 = torch.nn.Linear(n_in,20)
                self.layer2 = torch.nn.Linear(20,20)
                self.layer3 = torch.nn.Linear(20,20)
                self.layer_out = torch.nn.Linear(20,n_out)

            def forward(self, x):
                x = self.layer1(x)
                x = self.tanh(x)
                x = self.layer2(x)
                x = self.tanh(x)
                x = self.layer3(x)
                x = self.tanh(x)
                x = self.layer_out(x)

                return x
```

```
In [ ]: class blood_8pinn:
            def __init__(self, epochs, data, nde= 100, gamma = False):
                self.epochs = epochs
                self.model = Neural_net(n_out=8)
                self.domain = torch.linspace(0,100, nde, requires_grad=True).reshape(-1,1)

                ###  Variable parameters
                self.s1_unconstr = torch.nn.Parameter(torch.tensor([1.]))
                self.s2_unconstr = torch.nn.Parameter(torch.tensor([1.]))

                # Adding parameters to model trainable variables
                self.model.register_parameter(name='s1_param', param = self.s1_unconstr)
                self.model.register_parameter(name='s2_param', param =self.s2_unconstr)

                ### Optimizers
                self.lbfgs_optimizer = torch.optim.LBFGS(params = self.model.parameters(), lr = 0.001,max_iter = 500)
                self.adam_optimizer = torch.optim.Adam(params = self.model.parameters(), lr = 0.001)

                #### Simulation Fixed parameters
                patient_data, sim_data, vref , qref  = data
                self.pdat = patient_data

                self.Q_ref = qref
                self.Qinj_ref = 162/6500*self.Q_ref

                self.V_ref = torch.tensor(vref, dtype= torch.float)
                self.mc, self.qc =self.m_inj(np.linspace(0,100,nde))

                # self.Q = Q_ref + qc
                # self.Q_inj = Qinj + qc
                self.PS = 10


                ### Initialize data


                d_ids = np.arange(start=0, stop=len(sim_data[0]), step=len(sim_data[0])//nde)

                s_data ={}
                for i,d in enumerate(sim_data[1:]/HU):
                    s_data[i] = torch.tensor(d[d_ids], dtype=torch.float).reshape(-1,1)

                self.maxes ={i:max(d) for i,d in s_data.items()}
                self.mins = {i:min(d) for i,d in s_data.items()}

                if gamma: ## Using a gamma variate fit of data at end-compartment instead of pure data-points
                    popt, pcov = curve_fit(self.scaled_gamma, patient_data[0], patient_data[1], bounds = ((1000,1,1),(10000,10,10)))
                    self.c_patient = torch.tensor(self.scaled_gamma(np.linspace(0,100,100),popt[0],popt[1], popt[2]), dtype=torch.float).reshape(-1,1)
                    self.t_patient = torch.linspace(0,100,100).reshape(-1,1)

                else:
                    self.c_patient = torch.tensor(patient_data[1], dtype = torch.float).reshape(-1,1)
                    self.t_patient = torch.tensor(patient_data[0], dtype=torch.float).reshape(-1,1)

                self.c_patient_max = max(self.c_patient)
                self.c_patient_min = min(self.c_patient)
                self.c_patient_norm = (self.c_patient - self.c_patient_min)/(self.c_patient_max-self.c_patient_min)


            #Constrain parameters to be in range
            @property
            def s1_param(self):
                return self.s1_unconstr
```

```python
    @property
    def s2_param(self):
        return self.s2_unconstr

    def model_call(self,x):
        #return torch.abs(self.model(x))
        return self.model(x)

    #------------- Gamma variate
    def scaled_gamma(self, x,c, k, theta):
            frac = 1/(gamma(k)*theta**(k))
            return c*frac*x**(k-1)*np.exp(-(x/theta))


    # ------------
    def m_inj(self, t,ti=[0,3.33,3.33,6.67,6.67],qi=[6,6,6,6,0],ci=[350,350,0,0,0]):
        ti=np.array(ti)
        qi=np.array(qi)
        ci=np.array(ci)
        m = qi*ci
        return torch.tensor(np.interp(t,ti,m), dtype=torch.float).reshape(-1,1), torch.tensor(np.interp(t,ti,qi), dtype = torch.float).reshape(-1,1)

    def normalize(self, id, unnormed):
        return (unnormed - self.mins[id])/(self.maxes[id]- self.mins[id])

    def un_normalize(self, id, normed):
        return normed*(self.maxes[id] -self.mins[id])+ self.mins[id]


    def wrap_grad(self, f,x):
        return torch.autograd.grad(f,x,
        grad_outputs=torch.ones_like(x),
        retain_graph=True,
        create_graph=True)[0]


    def de_loss(self):
        pred = self.model_call(self.domain)
        cs = [x.reshape(-1,1) for x in torch.unbind(pred, dim =1)]
        dcs =[self.wrap_grad(c, self.domain) for c in cs]
        cs_unnormed = [self.un_normalize(i,d) for i,d in enumerate(cs)]

        c0,c1,c2,c3,c4,c5,c6,c7 = cs_unnormed
        dc0,dc1,dc2,dc3,dc4,dc5,dc6,dc7 = dcs



        Q = self.Q_ref*self.s1_param+ self.qc
        Q_inj = 162/6500*self.Q_ref*self.s1_param + self.qc
        V = self.V_ref*self.s2_param

        z0 = dc0 - (self.mc -(Q_inj*c0))/(V[0]*(self.maxes[0]- self.mins[0]))
        z1 = dc1 - ((Q_inj)*c0-Q*c1)/(V[1]*(self.maxes[1]- self.mins[1]))
        z2 = dc2 - (Q*c1-Q*c2)/(V[2]*(self.maxes[2]- self.mins[2]))
        z3 = dc3 - (Q*c2-Q*c3-self.PS*Q*(c3-c4))/(V[3]*(self.maxes[3]- self.mins[3]))
        z4 = dc4 - (self.PS*Q*(c3-c4))/(V[4]*(self.maxes[4]- self.mins[4]))
        z5 = dc5 - (Q*c3-Q*c5)/(V[5]*(self.maxes[5]- self.mins[5]))
        z6 = dc6 - (Q*c5-Q*c6)/(V[6]*(self.maxes[6]- self.mins[6]))
        z7 = dc7 - (Q*c6-Q*c7)/(V[7]*(self.maxes[7]- self.mins[7]))

        # z = (1/11111111)*(1e7*torch.mean(z0**2)+1e6*torch.mean(z1**2)+ 1e5*torch.mean(z2**2) + 1e4*torch.mean(z3**2) /
        #     1e3*torch.mean(z4**2) + 1e2*torch.mean(z5**2) + 1e1*torch.mean(z6**2) + 1e0*torch.mean(z7**2))

        weights = [1,1,1,1,1,1,1,1]

        zs = [torch.mean(z0**2), torch.mean(z1**2), torch.mean(z2**2) , torch.mean(z3**2) /
            torch.mean(z4**2) , torch.mean(z5**2) , torch.mean(z6**2) , torch.mean(z7**2)]

        z = sum([w*l for w,l in zip(zs,weights)])*(1/(sum(weights)))

        init_cond =  torch.mean(pred[0]**2)

        return init_cond + z


    def data_loss(self):
        c1,c2,c3,c4,c5,c6,c7,c8 = torch.unbind(self.model_call(self.t_patient), dim =1)
        return torch.mean((c8.reshape(-1,1) - self.c_patient_norm)**2)


    def combined_loss(self):
        return self.de_loss() + self.data_loss()


    def plot_preds(self):
        pred = self.model_call(self.domain)
        cs = [x for x in torch.unbind(pred, dim =1)]
        for i,c in enumerate(cs[1:]):
            plt.plot(self.domain.detach(),self.un_normalize(i+1,c.detach())*HU, label = f'c {i+2}')

        #plt.scatter(self.pdat[0], self.pdat[1])
        plt.grid()
        plt.legend()


    def lbfgs_train(self):
            self.model.train()
            for epoch in range(self.epochs):
                def closure():
                    self.lbfgs_optimizer.zero_grad()
                    loss = self.combined_loss()
                    loss.backward()
                    return loss
                self.lbfgs_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, training loss: {self.combined_loss()}')
            self.plot_preds()


    def adam_train(self):
            steps = 1000
            for epoch in range(self.epochs):
                for step in range(steps):
                    def closure():
                        self.adam_optimizer.zero_grad()
                        loss = self.combined_loss()
                        loss.backward()
                        return loss
                    self.adam_optimizer.step(closure=closure)
                print(f'Epoch {epoch}, training loss: {self.combined_loss()}, s1 {self.s1_param} S2: {self.s2_param}  Q: {self.Q_ref*self.s1_param*60}')
            self.plot_preds()
```

```python
In [ ]: class b8_pinn_noparams(blood_8pinn):
            def __init__(self, epochs, data, nde= 100, gamma = False):
                super().__init__(epochs, data, nde= 100, gamma= gamma)
                self.const = torch.tensor([1.], dtype=torch.float)

            @property
            def s2_param(self):
                return self.const

            @property
            def s1_param(self):
                return self.const

        class b8_pinn_sQ(blood_8pinn):
            def __init__(self, epochs, data, nde= 100, gamma = False):
                super().__init__(epochs, data, nde= 100, gamma = gamma)
                self.const = torch.tensor([1.], dtype=torch.float)

            @property
            def s1_param(self):
                return self.const

        class b8_pinn_sV(blood_8pinn):
            def __init__(self, epochs, data, nde= 100, gamma = False):
                super().__init__(epochs, data, nde= 100, gamma = gamma)
                self.const = torch.tensor([1.], dtype=torch.float)

            @property
            def s2_param(self):
                return self.const
```

In [ ]:
```
testdata = (patient4 ,p4_sim, vref4, qref4)

inst = blood_8pinn(epochs=20, data=testdata, gamma=False)
```

Patient 1

In [ ]:
```
p1data = (patient1, p1_sim, vref1, qref1)

print('Patient 1 s1, and s2')
p1_all_params= blood_8pinn(epochs=20, data = p1data)
p1_all_params.adam_train()
```

In [ ]:
```
print('Patient 1 s1, and s2, gamma fit')
p1_all_params_gamma = blood_8pinn(epochs=20, data = p1data, gamma=True)
p1_all_params_gamma.adam_train()
```

In [ ]:
```
print('Patient 1 s1')
p1_sq = b8_pinn_sQ(epochs=20, data = p1data)
p1_sq.adam_train()
```

In [ ]:
```
print('Patient 1 s1, gamma fit')
p1_sq_gamma = b8_pinn_sQ(epochs=20, data = p1data, gamma = True)
p1_sq_gamma.adam_train()
```

In [ ]:
```
print('Patient 1, s2')
p1_sv = b8_pinn_sV(epochs=20, data = p1data)
p1_sv.adam_train()
```

In [ ]:
```
print('patient 1, s2, gamma fit')
p1_sv_gamma = b8_pinn_sV(epochs=20, data = p1data, gamma = True)
p1_sv_gamma.adam_train()
```

In [ ]:
```
print('patient 1, no params')
# p1_none =b8_pinn_noparams(epochs=20, data = p1data)
#p1_none.adam_train()

p1_none.plot_preds()
t_ms = [p1_sim[0][np.argmax(c)] for c in p1_sim[1:]]
maxes =[max(c) for c in p1_sim[1:]]

d = 2
for i,p in zip(t_ms[1:], maxes[1:]):
    plt.scatter(i,p, marker='x', label =f'c{d} true peak')
    d+=1
plt.legend()
```

In [ ]:
```
preds = p1_none.model_call(p1_none.domain).detach().numpy()


t_mspred = [preds.T[0][np.argmax(c)] for c in preds.T[1:]]
pred_maxes =[max(c) for c in preds.T[1:]]
```

In [ ]:
```
print('patient 1 , no params, gamma fit')
# p1_none_gamma = b8_pinn_noparams(epochs=20, data = p1data, gamma=True)
p1_none_gamma.adam_train()

#p1_none_gamma.plot_preds()

d = 2
for i,p in zip(t_ms[1:], maxes[1:]):
    plt.scatter(i,p, marker='x', label =f'c{d} true peak')
    d+=1
plt.legend()
```

Patient 2

In [ ]:
```
p2data = (patient2, p2_sim, vref2, qref2)

#1
print('Patient 2 s1, and s2')
p2_all_params= blood_8pinn(epochs=20, data = p2data)
p2_all_params.adam_train()
```

In [ ]:
```
#2
print('Patient 2 s1, and s2, gamma fit')
p2_all_params_gamma = blood_8pinn(epochs=20, data = p2data, gamma=True)
p2_all_params_gamma.adam_train()
```

In [ ]:
```
#3
print('Patient 2 s1')
p2_sq = b8_pinn_sQ(epochs=20, data = p2data)
p2_sq.adam_train()
```

In [ ]:
```
#4
print('Patient 2 s1, gamma fit')
p2_sq_gamma = b8_pinn_sQ(epochs=20, data = p2data, gamma = True)
p2_sq_gamma.adam_train()
```

In [ ]:
```
#5
print('Patient 2, s2')
p2_sv = b8_pinn_sV(epochs=20, data = p2data)
p2_sv.adam_train()
```

In [ ]:
```
#6
print('patient 2, s2, gamma fit')
p2_sv_gamma = b8_pinn_sV(epochs=20, data = p2data, gamma = True)
p2_sv_gamma.adam_train()
```

In [ ]:
```
#7
print('patient 2, no params')
p2_none =b8_pinn_noparams(epochs=20, data = p2data)
p2_none.adam_train()
```

In [ ]:
```
#8
print('patient 2 , no params, gamma fit')
p2_none_gamma = b8_pinn_noparams(epochs=20, data = p2data, gamma=True)
p2_none_gamma.adam_train()
```

In [ ]:
```

```

Patient 3

```
In [ ]: p3data = (patient3, p3_sim, vref3, qref3)

        #1
        print('Patient 3 s1, and s2')
        p3_all_params= blood_8pinn(epochs=20, data = p3data)
        p3_all_params.adam_train()
```

```
In [ ]: #2
        print('Patient 3 s1, and s2, gamma fit')
        p3_all_params_gamma = blood_8pinn(epochs=20, data = p3data, gamma=True)
        p3_all_params_gamma.adam_train()
```

```
In [ ]: #3
        print('Patient 3 s1')
        p3_sq = b8_pinn_sQ(epochs=20, data = p3data)
        p3_sq.adam_train()
```

```
In [ ]: #4
        print('Patient 3 s1, gamma fit')
        p3_sq_gamma = b8_pinn_sQ(epochs=20, data = p3data, gamma = True)
        p3_sq_gamma.adam_train()
```

```
In [ ]: #5
        print('Patient 3, s2')
        p3_sv = b8_pinn_sV(epochs=20, data = p3data)
        p3_sv.adam_train()
```

```
In [ ]: #6
        print('patient 3, s2, gamma fit')
        p3_sv_gamma = b8_pinn_sV(epochs=20, data = p3data, gamma = True)
        p3_sv_gamma.adam_train()
```

```
In [ ]: #7
        print('patient 3, no params')
        p3_none =b8_pinn_noparams(epochs=20, data = p3data)
        p3_none.adam_train()
```

```
In [ ]: #8
        print('patient 3 , no params, gamma fit')
        p3_none_gamma = b8_pinn_noparams(epochs=20, data = p3data, gamma=True)
        p3_none_gamma.adam_train()
```

```
In [ ]:
```

Patient 4

```
In [ ]: p4data = (patient4, p4_sim, vref4, qref4)

        #1
        print('Patient 4 s1, and s2')
        p4_all_params= blood_8pinn(epochs=20, data = p4data)
        p4_all_params.adam_train()
```

```
In [ ]: #2
        print('Patient 4 s1, and s2, gamma fit')
        p4_all_params_gamma = blood_8pinn(epochs=20, data = p4data, gamma=True)
        p4_all_params_gamma.adam_train()
```

```
In [ ]: #3
        print('Patient 4 s1')
        p4_sq = b8_pinn_sQ(epochs=20, data = p4data)
        p4_sq.adam_train()
```

```
In [ ]: #4
        print('Patient 4 s1, gamma fit')
        p4_sq_gamma = b8_pinn_sQ(epochs=20, data = p4data, gamma = True)
        p4_sq_gamma.adam_train()
```

```
In [ ]: #5
        print('Patient 4, s2')
        p4_sv = b8_pinn_sV(epochs=20, data = p4data)
        p4_sv.adam_train()
```

```
In [ ]: #6
        print('patient 4, s2, gamma fit')
        p4_sv_gamma = b8_pinn_sV(epochs=20, data = p4data, gamma = True)
        p4_sv_gamma.adam_train()
```

```
In [ ]: #7
        print('patient 4, no params')
        p4_none =b8_pinn_noparams(epochs=20, data = p4data)
        p4_none.adam_train()
```

```
In [ ]: #8
        print('patient 4 , no params, gamma fit')
        p4_none_gamma = b8_pinn_noparams(epochs=20, data = p4data, gamma=True)
        p4_none_gamma.adam_train()
```

```
In [ ]:
```

Patient 5

```
In [ ]: p5data = (patient5, p5_sim, vref5, qref5)

        #1
        print('Patient 5 s1, and s2')
        p5_all_params= blood_8pinn(epochs=20, data = p5data)
        p5_all_params.adam_train()
```

```
In [ ]: #2
        print('Patient 5 s1, and s2, gamma fit')
        p5_all_params_gamma = blood_8pinn(epochs=20, data = p5data, gamma=True)
        p5_all_params_gamma.adam_train()
```

```
In [ ]: #3
        print('Patient 5 s1')
        p5_sq = b8_pinn_sQ(epochs=20, data = p5data)
        p5_sq.adam_train()
```

```
In [ ]: #4
        print('Patient 5 s1, gamma fit')
        p5_sq_gamma = b8_pinn_sQ(epochs=20, data = p5data, gamma = True)
        p5_sq_gamma.adam_train()
```

```
In [ ]: #5
        print('Patient 5, s2')
        p5_sv = b8_pinn_sV(epochs=20, data = p5data)
        p5_sv.adam_train()
```

```
In [ ]: #6
        print('patient 5, s2, gamma fit')
        p5_sv_gamma = b8_pinn_sV(epochs=20, data = p5data, gamma = True)
        p5_sv_gamma.adam_train()
```

```
In [ ]: #7
```

```
print('patient 5, no params')
p5_none =b8_pinn_noparams(epochs=20, data = p5data)
p5_none.adam_train()
```

```
In [ ]: #8
print('patient 5 , no params, gamma fit')
p5_none_gamma = b8_pinn_noparams(epochs=20, data = p5data, gamma=True)
p5_none_gamma.adam_train()
```

```
In [ ]:
```

Patient 6

```
In [ ]: p6data = (patient6, p6_sim, vref6, qref6)

#1
print('Patient 6 s1, and s2')
p6_all_params= blood_8pinn(epochs=20, data = p6data)
p6_all_params.adam_train()
```

```
In [ ]: #2
print('Patient 6 s1, and s2, gamma fit')
p6_all_params_gamma = blood_8pinn(epochs=20, data = p6data, gamma=True)
p6_all_params_gamma.adam_train()
```

```
In [ ]: #3
print('Patient 6 s1')
p6_sq = b8_pinn_sQ(epochs=20, data = p6data)
p6_sq.adam_train()
```

```
In [ ]: #4
print('Patient 6 s1, gamma fit')
p6_sq_gamma = b8_pinn_sQ(epochs=20, data = p6data, gamma = True)
p6_sq_gamma.adam_train()
```

```
In [ ]: #5
print('Patient 6, s2')
p6_sv = b8_pinn_sV(epochs=20, data = p6data)
p6_sv.adam_train()
```

```
In [ ]: #6
print('patient 6, s2, gamma fit')
p6_sv_gamma = b8_pinn_sV(epochs=20, data = p6data, gamma = True)
p6_sv_gamma.adam_train()
```

```
In [ ]: #7
print('patient 6, no params')
p6_none =b8_pinn_noparams(epochs=20, data = p6data)
p6_none.adam_train()
```

```
In [ ]: #8
print('patient 6 , no params, gamma fit')
p6_none_gamma = b8_pinn_noparams(epochs=20, data = p6data, gamma=True)
p6_none_gamma.adam_train()
```

```
In [ ]:
```