National Technical University of Athens
Master of Science: Computational Mechanics
Fluids Section
School of Chemical Engineering

Application of Physics-Informed Neural Networks (PINNs) for Reaction-Diffusion PDEs modeling an avascular growing tumor

Master Thesis

Ilias Katsifis

Supervisor: Assistant Professor
Mihalis Kavousanakis

Athens, 2024

This thesis is dedicated to the countless non-human animals used in laboratory testing and experimentation. It is my hope that advanced computational tools and the ethical use of artificial intelligence will pave the way to a more compassionate world, fostering a shared environment with all sentient beings.

Acknowledgements

First, I would like to express my deepest gratitude to Assistant Professor Mihalis Kavousanakis for his invaluable guidance throughout my Master's thesis. His enthusiasm for my topic and willingness to help at every stage made this journey both enriching and memorable.

Also, I am sincerely thankful to PhD candidate Ioannis Lampropoulos for his clear explanations of the physics behind the equations in my thesis. His patience and insight were instrumental in deepening my comprehension of complex concepts.

Last but not least, I am grateful to my family and friends for their unwavering emotional support throughout this journey. Their encouragement and understanding were my anchors during the challenging times, and their belief in me kept me motivated.

Abstract

This Master's thesis explores the application of deep learning techniques for reaction-diffusion partial differential equations (PDEs) modeling an avascular growing tumor, specifically utilizing physics-informed neural networks (PINNs). PINNs are an innovative and effective approach for solving partial differential equations (PDEs) and conducting parameter inference. The study focuses on a diffusion-reaction model that simulates the interaction between a cancerous tumor and the nutrient oxygen. To enhance the convergence and effectiveness of the neural network, a novel method known as dynamic weights was employed. More specifically a weight was assigned in each term of the loss function which includes the PDEs, the initial conditions and the boundary conditions. This technique adjusts the weights of each term in the loss function to address potential gradient imbalances during training. Additionally, parameter inference was performed for diffusion coefficients, which vary between patients due to the personalized nature of these values. The results were highly satisfactory, indicating the potential for extending this approach to higher dimensions and more complex geometries, which are common challenges in numerical methods.

Contents

1 Introduction

Numerical methods, such as finite difference, finite element, and finite volume methods, are essential tools for solving partial differential equations (PDEs) that arise in various scientific and engineering applications. These methods discretize the continuous domain into a finite set of points or elements, transforming the PDEs into a system of algebraic equations that can be solved using computational algorithms. By approximating the solutions over discrete points, numerical methods enable the analysis of complex systems that are analytically intractable. However, despite their robustness, these methods can be computationally intensive and may struggle with high-dimensional or highly nonlinear problems.

Artificial Intelligence (AI) has rapidly evolved from its early conceptual stages to becoming integral to modern technology. Initially conceived in the mid-20th century, AI has grown through various phases, including rule-based systems, machine learning, and the current era of deep learning. The progression from simple algorithms to complex neural networks has been driven by advancements in computational power, the availability of large datasets, and innovative algorithmic designs. In the realm of scientific computing, AI has shown significant potential to revolutionize engineering problems. By leveraging machine learning and deep learning techniques, AI can handle complex computations, optimize designs, and predict outcomes with high accuracy.

2 Artificial Neural Networks

2.1 Artificial Intelligence and Machine Learning

An entity engages in learning when it enhances its performance through observations of the environment. In the case of a computer acting as the learning agent, this process is termed **machine learning**. The computer observes data, constructs a model based on that data, and utilizes the model as both a hypothesis about the world and a functional software component capable of problem-solving. Enhancements to any segment of an agent's program can be achieved through machine learning. The specific improvements and the methodologies employed are contingent on several factors, including the targeted component for enhancement, the existing knowledge influencing the model construction by the agent, and the availability of data and feedback for further refinement. [1]

A machine learning algorithm takes data as input and generates an output. The input may be a factor representation, such as a vector containing attribute values. Alternatively, the input can be diverse data structures encompassing atomic and relational formats. [1] More specifically, machine learning methods allow us to (a) understand the cyber processes that created the data we are examining, (b) derive a model that reflects the core principles of these underlying processes, (c) project future trends or values based on this model, and (d) spot unusual or out-of-the-ordinary behavior in the data. [2] Machine learning finds applications in visual perception, speech recognition, gaming, expert systems, decision-making, healthcare, aviation, and language translation. [3]

If the output falls within a finite set of possibilities, such as true or false, the learning task is referred to as **classification**. On the other hand, when the outcome is a numerical value, such as temperature, which can be expressed as an integer or a real number, the learning problem is termed **regression**. [1]

There are three main types of machine learning:

- **Supervised Learning**: The agent observes pairs of input and output to acquire knowledge of a function that establishes a mapping from input to output. Supervised learning involves using various algorithms for classification and regression tasks to create models that can predict outcomes. These algorithms encompass linear and logistic regression and neural networks and extend to Support Vector Machines (SVM), random forests, naive Bayes, and k-nearest neighbors. [4] This thesis focuses on deep learning methods for solving partial differential equations, which are considered supervised learning.

- **Unsupervised Learning**: The agent discerns patterns within the input data without relying on explicit feedback. The primary task in unsupervised learning often involves clustering, where the objective is to identify potentially meaningful groups or clusters among input examples. Cluster analysis has various applications, such as gene sequence analysis, market research, and object recognition. Popular algorithms for unsupervised learning encompass clustering, anomaly detection, neural networks, and techniques for learning latent variable models. [5]

- **Reinforcement Learning**: The agent acquires knowledge through a reinforcement sequence comprising rewards and penalties. The agent is responsible for determining which of the preceding actions led to the received reinforcement, subsequently adjusting its behavior to optimize for increased rewards in future interactions. [1] Reinforcement learning has been instrumental in several significant breakthroughs in machine learning, such as autonomous vehicles, gaming, and data center management. [6]

Machine learning technology has now become a conventional element in software engineering. Whenever a software system is developed, various components within the system can benefit from the integration of machine learning. For instance, a machine-learned model accelerated the analysis of images depicting galaxies under gravitational lensing by a factor of 10 million [7], and another machine-learned model led to a 40% reduction in energy consumption for cooling data centers. [8][1]

2.1.1 Inspiration from biological neural networks

The human brain comprises a substantial number, approximately 10^{11}, of interconnected elements known as neurons, each having around 10^4 connections per element. These neurons, as pictured in Figure 1, comprise three essential components: dendrites, the cell body, and the axon. Dendrites form tree-like receptive networks of nerve fibers responsible for carrying electrical signals into the cell body. The cell body performs the crucial functions of summing and thresholding these incoming signals. Meanwhile, the axon serves as a single, elongated fiber transmitting the signal from the cell body to other neurons. The point where an axon connects with a dendrite is termed a synapse. The overall function of the neural network is determined by the arrangement of neurons and the strengths of individual synapses, which are influenced by a complex chemical process. [9] . These networks are integral to cognitive functions like learning, memory, perception, and decision-making.[10]

An action potential—often called a ”spike”—is a primary activation mechanism in biological neurons. When the total input to a neuron exceeds a certain threshold, the neuron sends an output signal through its axon, allowing the process to continue throughout the network. In this setting, ”training” describes the process of altering synaptic connections, either reinforcing or diminishing them. External stimuli and experiences influence these changes in synaptic strength.[10]

Artificial neural networks fall short of replicating the intricate complexity of the human brain. Nevertheless, they share two fundamental similarities with biological neural networks. Firstly, both networks are constructed from basic

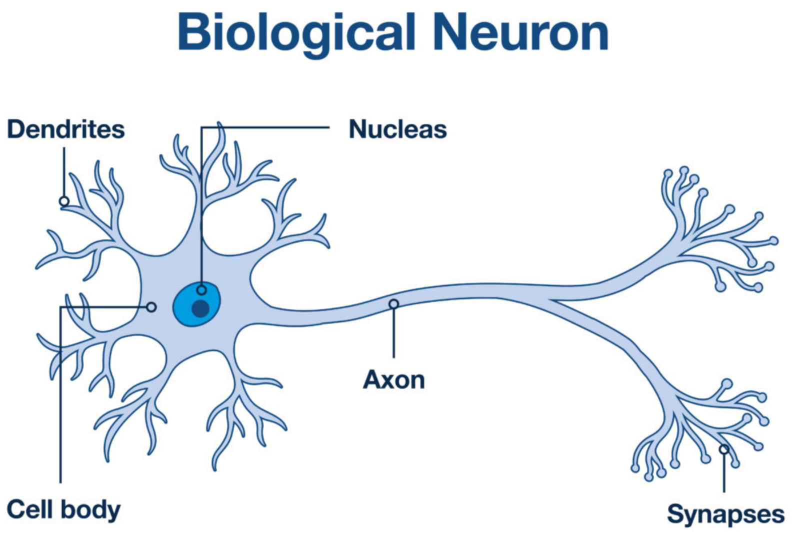


Figure 1: Illustration of a biological neuron [11]

computational units (though artificial neurons are notably simpler than their biological counterparts) that exhibit extensive interconnections. Secondly, the functionality of these networks is contingent upon the connections between neurons. [9]

Notably, while biological neurons exhibit a considerably slower speed (10^{-3} sec compared to 10^{-10} sec) when compared to electrical circuits, the brain can execute tasks at a pace surpassing that of traditional computers. This capability is attributed, in part, to the extensively parallel structure of biological neuronal networks, where all neurons function simultaneously. Although artificial neural networks are presently deployed on conventional digital computers, their inherently parallel structure makes them well-suited for implementation using Very-Large-Scale Integration (VLSI), optical devices, and parallel processors. [9]

2.1.2 Linear Regression

Central to each solution lies a model elucidating the transformation of features into an estimated target. The presumption of linearity posits that the anticipated value of the target can be represented as a weighted sum of the features. In machine learning, engaging with datasets characterized by high dimensions is customary. Compact linear algebra notation proves to be more convenient in such cases. [12][13] In scenarios where our inputs encompass d features, each feature can be assigned an index ranging from 1 to d, allowing us to articulate our prediction $\hat{y}$ as

$$\hat{y} = w_1 x_1 + ... + w_d x_d + b, \tag{2.1}$$

where $w_1 ... w_d$ are called weights and b bias. If the equation provided above is written in vector format, where $\mathbf{x} \in \mathbb{R}^d$ is the feature vector and $\mathbf{w} \in \mathbb{R}^d$ the weights vector then

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{2.2}$$

Certainly, aligning our model with the data necessitates a consensus on a fitness metric. Loss functions serve to measure the discrepancy between actual and predicted target values. Typically, the loss is a non-negative numerical value, with smaller values signifying better alignment and perfect predictions incurring a loss of 0. In the context of regression problems, the squared error stands out as the most prevalent loss function. [12] When our prediction for a given example i is denoted as $\hat{y}^{(i)}$, and the corresponding true label is $y^{(i)}$, the squared error is calculated as

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2. \tag{2.3}$$

Including the constant $\frac{1}{2}$ is inconsequential but serves a notational convenience as it cancels out during the derivative calculation of the loss. As the training dataset is provided and beyond our control, the empirical error solely functions as a variable dependent on the model parameters. [12] For an entire dataset consisting of n examples, the loss function is

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(\mathbf{w}^\top \mathbf{x} + b - y^{(i)})^2. \tag{2.4}$$

2.1.3 Gradient Based Optimization

Many deep learning algorithms typically entail the process of optimization. Optimization involves the endeavor to adjust the variable x in order to either minimize or maximize a given function $f(\mathbf{x})$. [14]

Consider a function $y = f(x)$ where both x and y are real numbers. The derivative of this function is represented as $f'(x)$ or $\frac{dy}{dx}$. The derivative $f'(x)$ signifies the slope of $f(x)$ at the point x, indicating how a small change in the input relates to a corresponding change in the output, expressed as $f(x + \epsilon) = f(x) + \epsilon f'(x)$. This derivative proves instrumental in function minimization as it guides adjustments to x for incremental improvements in y. Thus, to diminish $f(x)$, we iteratively adjust x in small increments opposite to the sign of the derivative. For instance, in the provided illustration (Figure 2) where $x > 0, f'(x) > 0$, suggesting a leftward movement to decrease f, while for $x < 0, f'(x) < 0$, indicating a rightward shift for minimizing f. [14]

In instances where $f'(x) = 0$, the derivative fails to offer guidance on the direction to proceed. Such points, labeled as critical points or stationary points,

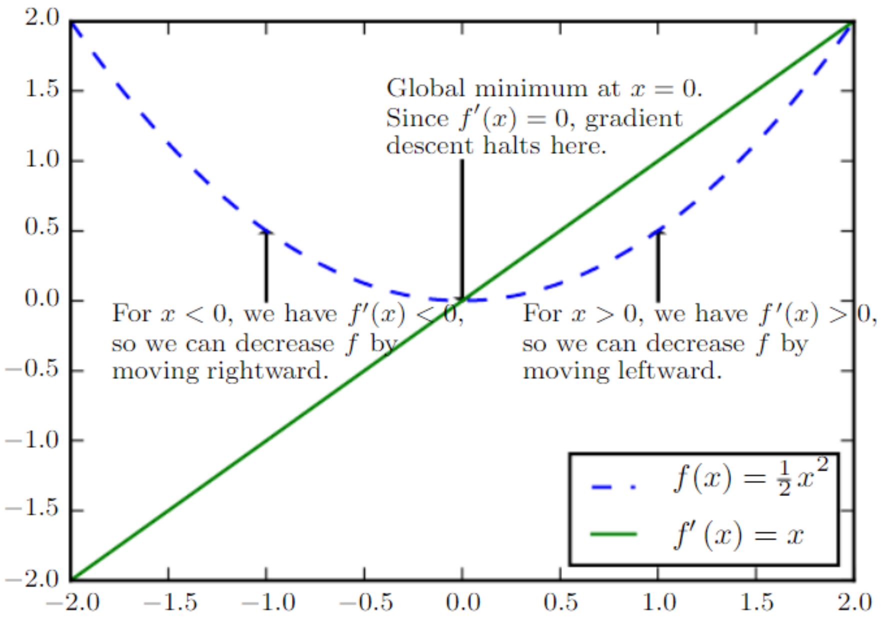


Figure 2: Simple example of gradient descent [14]

denote positions where $f'(x) = 0$. A local minimum occurs when $f(x)$ is lower than its adjacent points, rendering it impossible to further decrease $f(x)$ through infinitesimal steps. Certain critical points do not represent maxima or minima; these are termed saddle points. These three cases can be seen in Figure 3 [14]

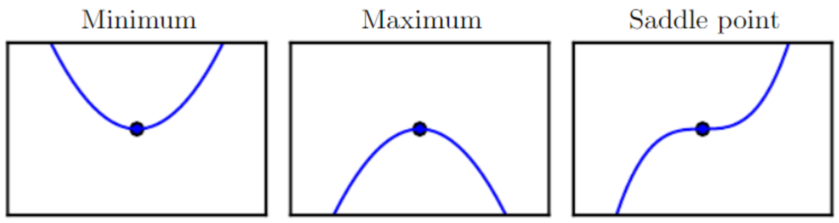


Figure 3: Depiction of minimum, maximum and saddle points of a function [14]

A point achieving the lowest possible value of $f(x)$ is termed a global minimum. In the realm of deep learning, optimizing functions often encounter numerous suboptimal local minima alongside saddle points bordered by extensively flat regions. These complexities render optimization challenging, particularly when dealing with multidimensional input. Consequently, the typical approach involves aiming to identify a value of f that is significantly low, albeit not necessarily meeting formal minimal criteria. An example of approximate function minimization can be seen in Figure 4. [14]

Gradient-based techniques frequently result in reaching a local optimum. On the other hand, non-gradient methods tend to converge to a global optimum, but they usually involve a high number of function evaluations. For large-scale tasks, as often seen in engineering design, non-gradient approaches are generally less efficient. Gradient-based algorithms need both gradient or sensitivity information and function evaluations to establish the best directions for searching, helping to refine designs during optimization. [15]

In the case of functions with multiple inputs, the concept of partial derivatives becomes essential. The partial derivative, denoted by the partial symbol $\frac{\partial f(\mathbf{x})}{\partial x_i}$, quantifies how f alters as only the variable x_i increases at point x. The gradient extends the idea of derivative to scenarios where the derivative pertains to a vector. [14]

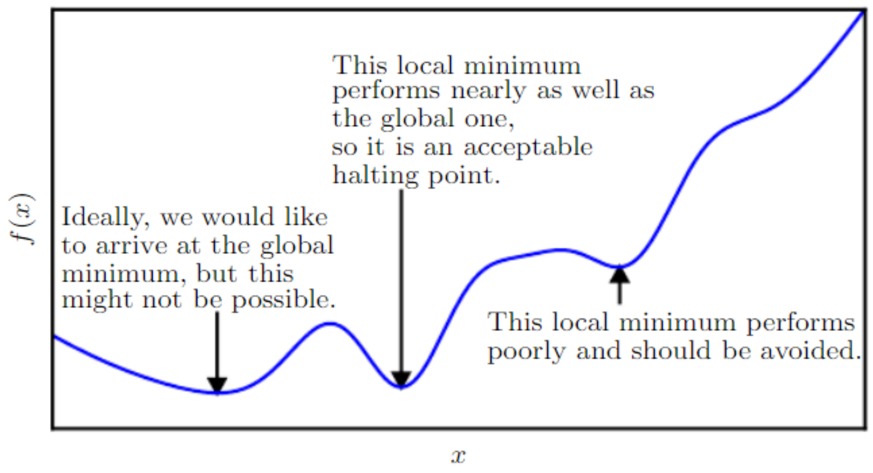


Figure 4: Approximate minimization [14]

The directional derivative along the unit vector $\mathbf{u}$ signifies the rate of change of function f in the direction of $\mathbf{u}$. Put differently, it represents the derivative of the function $f(\mathbf{x} + a\mathbf{u})$ with respect to a, assessed at $a = 0$. By employing the chain rule, we observe that $\frac{\partial f(\mathbf{x}+a\mathbf{u})}{\partial a}$ evaluates to $\mathbf{u}^\top \nabla_x f(\mathbf{x})$ when $a = 0$. To minimize f, our aim is to identify the direction that results in the most rapid decrease of f.

$$min_{u,u^\top u=1} \mathbf{u}^\top \nabla_x f(\mathbf{x}) = min_{u,u^\top u=1} ||\mathbf{u}||_2 ||\nabla_x f(\mathbf{x})||_2 cos\theta, \tag{2.5}$$

where θ is the angle between $\mathbf{u}$ and the gradient. If we choose to ignore the factors that are not dependent on $\mathbf{u}$ and consider $||\mathbf{u}||_2 = 1$, then the equation provided above is simplified to $min_u ||\nabla_x f(\mathbf{x})||_2 cos\theta$. This is minimized when $\mathbf{u}$ points in the opposite direction of the gradient. In simpler terms, the gradient indicates the steepest ascent, while the negative gradient indicates the steepest descent. Moving in the direction of the negative gradient allows us to decrease f. [14]

This approach is commonly known as steepest descent or gradient descent. Steepest descent suggests updating the point

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_x f(\mathbf{x}), \tag{2.6}$$

where ϵ represents the learning rate, a positive scalar determining the step size. There are various ways to select ϵ. One popular method is to assign ϵ a small constant value. Alternatively, we may solve for the step size that nullifies the directional derivative. Another approach involves evaluating $f(\mathbf{x}' = \mathbf{x} - \epsilon \nabla_x f(\mathbf{x}))$ for multiple ϵ values and selecting the one that yields the smallest objective function value. This latter strategy is referred to as a line search. [14]

Steepest descent converges when each element of the gradient approaches zero (or is very close to zero in practical terms). While gradient descent is typically applied to optimization in continuous spaces, the fundamental idea of iteratively making incremental moves toward improved configurations can be extended to discrete spaces. [14]

2.1.4 Jacobian and Hessian Matrices

Occasionally, there arises a necessity to compute the partial derivatives of a function that takes vector inputs and yields vector outputs. The compilation of these partial derivatives forms what is termed a Jacobian matrix. More precisely, for a function $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, the Jacobian matrix is defined as

$$J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x}_i). \tag{2.7}$$

The Jacobian matrix can be a vital tool in mathematical analysis. By linearizing a nonlinear system at a particular point, it allows the use of linear system methods to simplify and better understand the nonlinear system. [16]

At times, we may also be concerned with the derivative of a derivative, commonly referred to as a second derivative. For instance, the second derivative of a function f with respect to x_j is denoted as

$$\frac{\partial^2}{\partial x_i \partial x_j} f. \tag{2.8}$$

For a single dimension, it can be denoted as

$$f'' = \frac{\partial^2}{\partial x^2} f. \tag{2.9}$$

The secondary derivative provides insight into how alterations in the input affect the variation of the first derivative. This holds significance as it elucidates whether a gradient step will yield the anticipated enhancement solely based on the gradient. Conceptually, the second derivative gauges curvature. Consider a quadratic function; when its second derivative is zero, it indicates the absence of curvature, rendering it a completely flat line whose value can be determined solely through the gradient. The types of curvature of a function can be seen at Figure 5. [14]

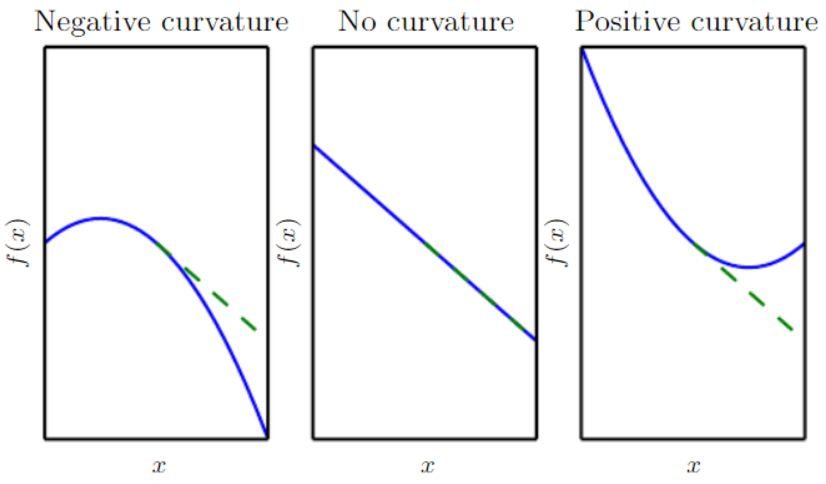


Figure 5: Types of Curvature [14]

When dealing with functions in multiple dimensions, the existence of numerous second derivatives becomes apparent. These derivatives can be consolidated into a matrix known as the Hessian matrix. [17] The Hessian matrix, denoted as $H(f)(\mathbf{x})$, is defined as

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}). \tag{2.10}$$

As the Hessian matrix is both real and symmetric, we decompose it into a set of eigenvalues and an orthogonal basis comprising eigenvectors. The second derivative in a particular direction, represented by a unit vector d, is expressed as

$$\mathbf{d}^\top \mathbf{H} \mathbf{d}. \tag{2.11}$$

When d represents an eigenvector of $\mathbf{H}$, the corresponding eigenvalue governs the second derivative in that direction. The maximum eigenvalue signifies the maximum second derivative, while the minimum eigenvalue denotes the minimum second derivative. [14]

The directional second derivative provides insight into the efficacy of a gradient descent's performance. We can employ a second-order Taylor series approximation to the function $f(\mathbf{x})$ around the current point $\mathbf{x}(0)$:

$$f(\mathbf{x}) = f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H}(\mathbf{x} - \mathbf{x}^{(0)}), \tag{2.12}$$

where $\mathbf{g}$ is the gradient and $\mathbf{H}$ is the Hessian at $\mathbf{x}^{(0)}$. If we utilize a learning rate of ϵ, then the new point $\mathbf{x}$ will be given by $\mathbf{x}^{(0)} - \epsilon\mathbf{g}$. Substituting this into our approximation, we obtain:

$$f(\mathbf{x}^{(0)} - \epsilon\mathbf{g}) = f(\mathbf{x}^{(0)}) - \epsilon\mathbf{g}^\top\mathbf{g} + \frac{1}{2}\epsilon^2\mathbf{g}^\top\mathbf{H}\mathbf{g}. \tag{2.13}$$

This expression comprises three terms: the original value of the function, the anticipated enhancement attributable to the slope of the function, and the adjustment needed to accommodate the curvature of the function. [14]

2.1.5 Second order gradient methods - Newton's method

Unlike first-order methods, second-order methods leverage second derivatives to enhance optimization. Among these, Newton's Method stands out as the most widely utilized second-order method. [14]

Newton's method involves an optimization approach that employs a second-order Taylor series expansion to estimate the loss function L in the vicinity of a point θ_0, while disregarding higher-order derivatives.

$$L(\theta) = L(\theta_{\mathbf{0}}) + (\theta - \theta_{\mathbf{0}})^\top \nabla_\theta L(\theta_0) + \frac{1}{2}(\theta - \theta_{\mathbf{0}})^\top \mathbf{H}(\theta - \theta_{\mathbf{0}}), \tag{2.14}$$

where $\mathbf{H}$ is the Hessian of the loss function L with respect to θ and evaluated at $\theta_{\mathbf{0}}$. Upon determining the critical point of this function, we derive the Newton parameter update rule

$$\theta^* = \theta_{\mathbf{0}} - \mathbf{H}^{\mathbf{-1}}\nabla_\theta L(\theta_{\mathbf{0}}). \tag{2.15}$$

Hence, for a locally quadratic function (with a positive definite $\mathbf{H}$), by adjusting the gradient with $\mathbf{H}^{\mathbf{-1}}$, Newton's method directly converges to the minimum. This updating process can be iterated if the objective function is convex but not strictly quadratic (involving higher-order terms). The algorithm can be seen in Figure 6.

In addition to grappling with challenges arising from particular characteristics of the objective function, such as saddle points, the practical application of Newton's method in training large neural networks is hampered by its considerable computational demands. Given that the size of the Hessian matrix scales quadratically with the number of parameters, even modestly sized neural networks with millions of parameters (denoted as m) would necessitate the inversion of a mXm matrix, incurring a computational complexity of $O(m^3)$.

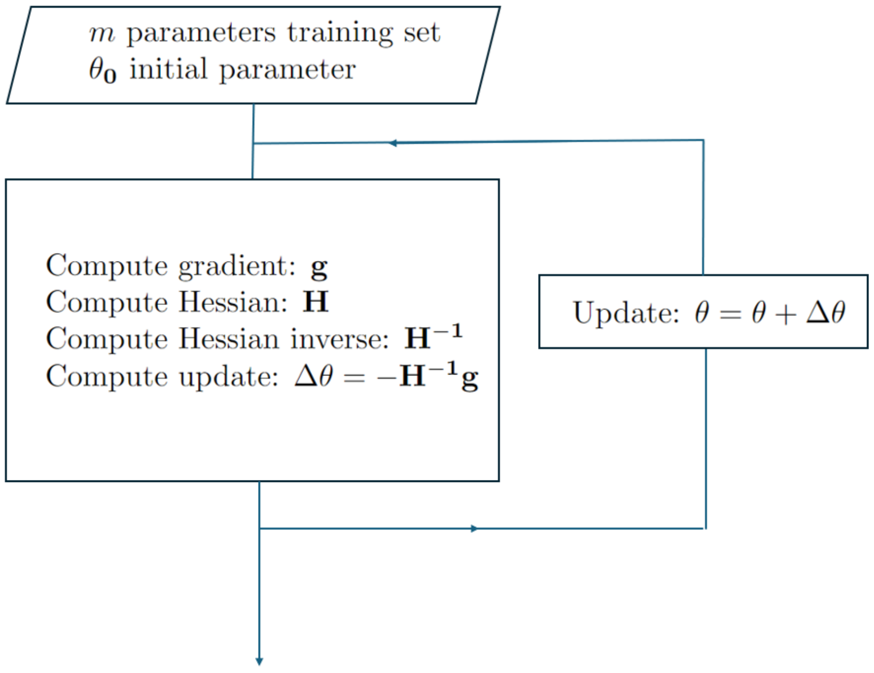

m parameters training set
θ_0 initial parameter
Compute gradient: $\mathbf{g}$
Compute Hessian: $\mathbf{H}$
Compute Hessian inverse: $\mathbf{H^{-1}}$
Compute update: $\Delta\theta = -\mathbf{H^{-1}g}$
Update: $\theta = \theta + \Delta\theta$

Figure 6: Algorithm for Newton's Method

Moreover, since the parameters undergo alterations with each update, the inverse Hessian must be recomputed at every iteration of training. Consequently, Newton's method is only feasible for training networks with a very limited number of parameters. [14]

2.1.6 BFGS/LBFGS method

The Broyden - Fletcher - Goldfarb - Shanno (BFGS) algorithm aims to incorporate some of the benefits of Newton's Method while mitigating its computational demands. Quasi-Newton methods like the BFGS method pursue an approach where they approximate the inverse matrix $\mathbf{H^{-1}}$ with a matrix $\mathbf{M_t}$, which is progressively refined through low-rank updates to enhance its approximation accuracy. [14][18][19]

To take the quasi-Newton step, minimizing $\Delta\theta$ is equivalent to approximating $\mathbf{H} \cong \mathbf{M_t^{-1}}$. The matrix update follows the well-known secant equation::

$$\mathbf{H_{t+1}} = \nabla L(\theta_{\mathbf{t+1}}) - \nabla L(\theta_{\mathbf{t}}). \tag{2.16}$$

If we set

$$\mathbf{s_t} = \theta_{\mathbf{t+1}} - \theta_{\mathbf{t}}, \tag{2.17}$$

and

$$\mathbf{y_t} = \nabla L(\theta_{\mathbf{t+1}}) - \nabla L(\theta_{\mathbf{t}}). \tag{2.18}$$

Then (2.16) becomes:

$$\mathbf{H_{t+1}s_t} = \mathbf{y_t} \Rightarrow \mathbf{M_{t+1}y_t} \cong \mathbf{s_t}. \tag{2.19}$$

The conditions that $\mathbf{M_{t+1}}$ be symmetric and positive definite are necessary but not sufficient to uniquely determine $\mathbf{M_{t+1}}$. To uniquely determine it, we also require:

$$\mathbf{M_{t+1}} = argmin||\mathbf{M} - \mathbf{M_t}||. \tag{2.20}$$

Thus, $\mathbf{M_{t+1}}$ must be the closest symmetric positive definite matrix to $\mathbf{M_t}$ that satisfies the given constraints. In the BFGS method the update formula is:

$$\mathbf{M^{BFGS}_{t+1}} = (\mathbf{I} - \rho_{\mathbf{t}}\mathbf{s_t}(\mathbf{y_t})^\top)\mathbf{M_t}(\mathbf{I} - \rho_{\mathbf{t}}\mathbf{y_t}(\mathbf{s_t})^\top) + \rho_{\mathbf{t}}\mathbf{s_t}(\mathbf{s_t})^\top, \tag{2.21}$$

where

$$\rho_{\mathbf{t}} = ((\mathbf{y_t})^\top \mathbf{s_t})^{\mathbf{-1}}. \tag{2.22}$$

This requires only one multiplication mXm because:

$$\mathbf{M_t}(\mathbf{I} - \rho_{\mathbf{t}}\mathbf{y_t}(\mathbf{s_t})^\top) = \mathbf{M_t} - \rho(\mathbf{M_t y_t})(\mathbf{s_t})^\top. \tag{2.23}$$

Nevertheless, the BFGS algorithm necessitates storing the inverse Hessian matrix $\mathbf{M_t}$, which demands $O(m^2)$ memory. This constraint renders BFGS impractical for the majority of contemporary deep-learning techniques, which commonly involve millions of parameters. [14]

The memory requirements of the BFGS algorithm can be substantially reduced by circumventing the need to store the entire inverse Hessian approximation $\mathbf{M_t}$. The L-BFGS algorithm computes this approximation using a similar approach as BFGS. However, it starts with the assumption that $\mathbf{M_t}$ is the identity matrix instead of storing the approximation from one iteration to the next. Extending this idea, the L-BFGS approach without storage can be expanded to incorporate additional information about the Hessian by retaining some of the vectors employed to update $\mathbf{M_t}$ at each iteration.[14][20]

More specifically if $\mathbf{M_{t(init)}}$ is the identity matrix then let's assume:

$$\mathbf{r_{t(init)}} = \mathbf{M_{t(init)}}\nabla\mathbf{L}(\theta_{\mathbf{t(init)}}), \tag{2.24}$$

and let's assume:

$$\gamma_{\mathbf{t}} = ((\mathbf{s_{t-1}})^\top(\mathbf{y_{t-1}}))((\mathbf{y_{t-1}})^\top \mathbf{y_{j-1}})^{-1}, \tag{2.25}$$

$$\mathbf{q} = \gamma_{\mathbf{t}}\nabla\mathbf{L}(\theta_{\mathbf{t}}). \tag{2.26}$$

For the first n iterations

$$\begin{aligned} &for\ i = 1 : n-1 \\ &\quad \mathbf{a_i} = \rho_{\mathbf{i}}(\mathbf{s_i})^\top \mathbf{q_i} \\ &\quad \mathbf{q_i} = \mathbf{q_i} - \mathbf{a}\mathbf{y_i} \\ &end, \end{aligned} \tag{2.27}$$

and for the rest of m equations:

$$\begin{aligned} &for\ i = n : m \\ &\quad \beta_{\mathbf{i}} = \rho_{\mathbf{i}}(\mathbf{y_i})^\top \mathbf{r_i} \\ &\quad \mathbf{r_i} = \mathbf{r_i} + \mathbf{s_i}(\mathbf{a_i} - \beta_{\mathbf{i}}) \\ &end. \end{aligned} \tag{2.28}$$

If $n << m$, then this incurs only O(n) memory per step making it a suitable optimization option which takes into account the curvature of the loss function. [21]

2.2 Training of Artificial Neural Networks

2.2.1 Deep Feedforward Networks

Deep feedforward networks, also known as feedforward neural networks or multilayer perceptrons (MLPs), represent the cornerstone of deep learning models. Their primary objective is to approximate a function f. Given an input $\mathbf{x}$, a feedforward network establishes a mapping $y = f(\mathbf{x}, \theta)$ and subsequently learns the parameter values θ in order to approximate the desired function. [14]

These models earn the designation "feedforward" due to the unidirectional flow of information from the input, $\mathbf{x}$, through intermediate computations defining f, and ultimately to the output, $\mathbf{y}$. Unlike recurrent neural networks, which incorporate feedback connections allowing outputs to influence subsequent computations within the model, feedforward networks strictly adhere to this unidirectional flow. [14]

Feedforward neural networks derive their name from their structure, which involves the composition of multiple distinct functions. For instance, when three functions—$f^{(1)}$, $f^{(2)}$, and $f^{(3)}$—are linked sequentially to construct $f(\mathbf{x})$, it takes the form $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. These sequential configurations represent the prevailing architecture in neural networks. Here, $f^{(1)}$ assumes the role of the initial layer, designated as the first layer, followed by $f^{(2)}$ as the second layer, and so forth. The cumulative length of these interconnected layers defines the model's depth, hence coining the term "deep learning." The ultimate layer in a feedforward network is referred to as the output layer, with the intermediary layers termed hidden layers. [14][22][23]

In order to gain a better understanding of the architecture of neural networks, it is worth studying a single input neuron as shown in Figure 7:

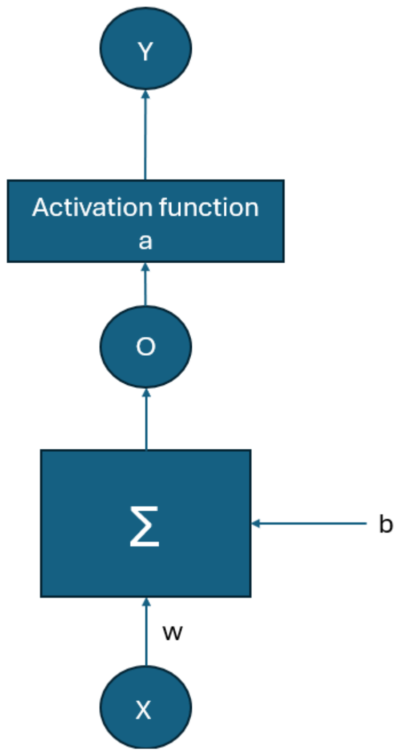


Figure 7: Single Input Neuron

So the output of the single input neuron in Figure 7 is calculated from the equation

$$y = a(wx + b). \tag{2.29}$$

The scalar input x undergoes multiplication with a scalar weight w, resulting in wx, constituting one of the terms transmitted to the summation process. The other input involves the bias b. The summation process then passes through an activation function, denoted as a, yielding a scalar output y for the neuron. [9]

It's important to recognize that both w and b represent adjustable scalar parameters of the neuron. Ordinarily, the designer selects the activation function, while the parameters w and b are modified according to a learning rule, ensuring that the neuron's input/output relationship aligns with a predefined objective. [9]

In Figure 8 the case of multiple input and multiple output single layer neural network is depicted.

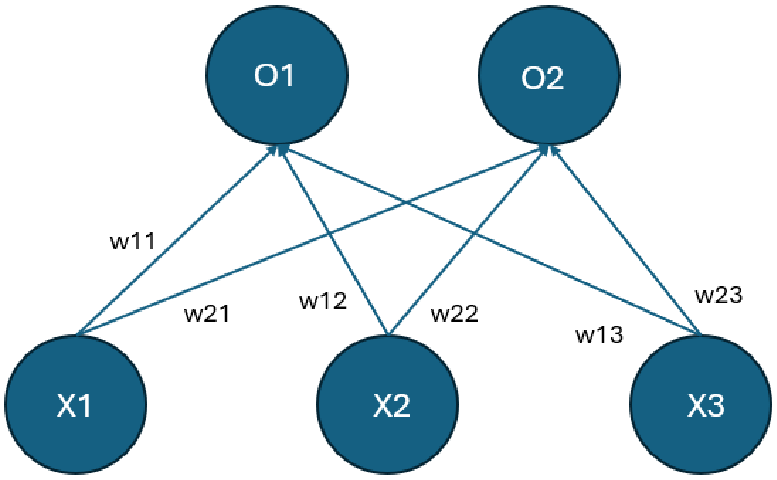


Figure 8: Multiple Input and Multiple Output Single Layer Neuron

The outputs are evaluated:

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + b_1, \tag{2.30}$$

$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + b_2. \tag{2.31}$$

In pursuit of enhanced computational efficiency, it is important to opt for vectorized computations. Consider a scenario where we have a matrix $\mathbf{X} \in \mathbb{R}^{nxd}$ representing n examples with each example possessing a dimensionality (number of outputs) of d. Additionally, let's suppose there are q categories in the output. The corresponding weight matrix is $\mathbf{W} \in \mathbb{R}^{dxq}$ and bias vector $\mathbf{b} \in \mathbb{R}^{1xq}$, the output is [12]

$$\mathbf{O} = \mathbf{XW} + \mathbf{b}, \tag{2.32}$$

and taking into account the activation function a [12]

$$\mathbf{Y} = a(\mathbf{O}). \tag{2.33}$$

2.2.2 Activation functions

An activation function can take the form of either a linear or non-linear function. A specific activation function is selected to fulfill certain specifications inherent to the problem that the neuron aims to address. The most popular activation functions used in deep learning are presented below:

Hard limit activation function
The hard limit activation function assigns a neuron's output to 0 when the function's argument is below zero, and to 1 if the argument is zero or higher.[9]

Sigmoid activation function
The log-sigmoid activation function processes the input, which could range from negative to positive infinity, and compresses the output within a specified range, as dictated by the expression:

$$a = \frac{1}{1 + e^{-x}}. \tag{2.34}$$

The sigmoid function is frequently employed in multilayer networks trained with the backpropagation algorithm, primarily due to its differentiability.[9][24][25]

With the focus turning towards gradient-based learning, the sigmoid function emerged as a logical selection due to its smooth, differentiable approximation to a thresholding unit. Sigmoids remain prevalent as activation functions on output units, particularly when interpreting outputs as probabilities in binary classification tasks. However, a drawback of the sigmoid function lies in optimization challenges, as its gradient diminishes significantly for large positive and negative arguments, potentially resulting in difficult-to-overcome plateaus.[12]

Hyperbolic tangent function
Similar to the sigmoid function, the hyperbolic tangent function (*tanh*) compresses its inputs, converting them into values within the range of -1 to 1: [26]

$$a = tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}. \tag{2.35}$$

While the *tanh* function shares a similar shape with the sigmoid function, it demonstrates point symmetry around the origin of the coordinate system.[12]

<u>ReLU function</u>
The rectified linear unit (ReLU), proposed by Nair and Hinton in 2010 [27], stands out as the preferred option for its straightforward implementation and effective performance across various predictive tasks. ReLU offers a straightforward non-linear transformation, defined as the maximum value between an element x and 0:

$$a = ReLU(x) = max(x, 0). \tag{2.36}$$

In simple terms, the ReLU function keeps only positive elements, disregarding negative ones by assigning their activations to 0. For negative inputs, ReLU has a derivative of 0, while for positive inputs, its derivative is 1.[12][28] The mentioned activation functions are depicted in Figure 9:

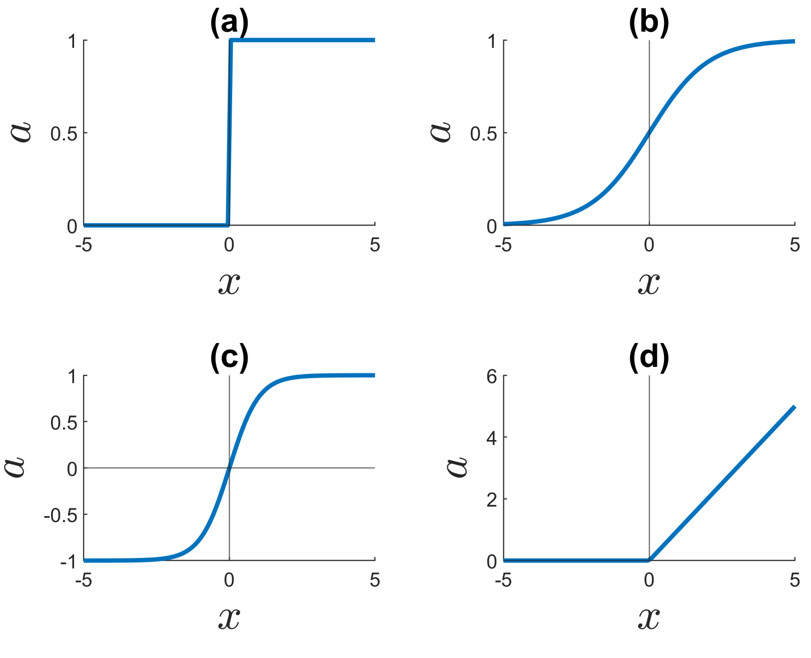


Figure 9: (a) Hard Limit activation function, (b) Sigmoid activation function, (c) Hyperbolic Tangent function, (d) ReLU activation function

2.2.3 Forward Propagation

Forward propagation, also known as the forward pass, involves computing and storing intermediate variables, including outputs, for a neural network from the input layer to the output layer. [29] Let's now proceed step by step to understand the mechanics of a neural network with a single hidden layer shown in Figure 10.

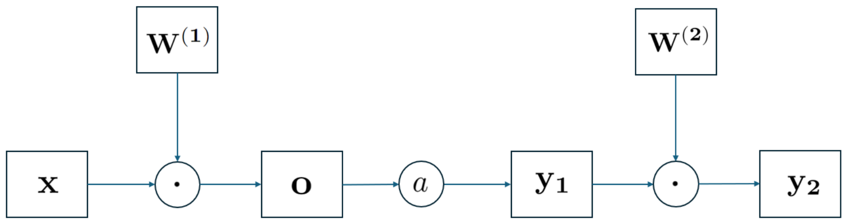


Figure 10: Neural network with a single hidden layer (Forward Propagation)

To simplify matters, let's assume that the input example is denoted as $\mathbf{x} \in \mathbb{R}^{dX1}$, and our hidden layer does not incorporate a bias term. In this scenario, the intermediate variable is

$$\mathbf{o} = \mathbf{W^{(1)}}\mathbf{x}, \tag{2.37}$$

where $\mathbf{W^{(1)}} \in \mathbb{R}^{hXd}$ is the weight parameter of the hidden layer. Following the processing of the intermediate variable through the activation function, we derive our hidden activation vector, which has a length of h:

$$\mathbf{y_1} = a(\mathbf{o}). \tag{2.38}$$

The hidden layer $\mathbf{y_1} \in \mathbb{R}^{hX1}$ serves as another intermediate variable. If we assume that the parameters of the output layers consist solely of a weight, denoted as $\mathbf{W^{(2)}} \in \mathbb{R}^{qXh}$, we can generate an output layer with a vector of length q [12]

$$\mathbf{y_2} = \mathbf{W^{(2)}}\mathbf{y_1}. \tag{2.39}$$

2.2.4 Backpropagation

Backpropagation involves computing the gradients of neural network parameters. This process entails traversing the network in reverse, from the output to the input layer, applying the chain rule from calculus. [30] Intermediate variables (partial derivatives) necessary for gradient computation are stored during this algorithm. For instance, considering functions $\mathbf{Y} = f(\mathbf{X})$ and $\mathbf{Z} = g(\mathbf{Y})$, where $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ represent tensors of varying shapes, we can calculate the derivative of **Z** concerning **X** using the chain rule [12]

$$\frac{\partial \mathbf{Z}}{\partial \mathbf{X}} = prod\left(\frac{\partial \mathbf{Z}}{\partial \mathbf{Y}}, \frac{\partial \mathbf{Y}}{\partial \mathbf{X}}\right). \tag{2.40}$$

In this context, the *prod* operator is utilized to multiply its arguments following operations like transposition and input position swapping. For vectors, this process is straightforward, akin to matrix-matrix multiplication. With higher-dimensional tensors, we employ the corresponding operation accordingly. The *prod* operator simplifies the notation, concealing any complexities. Considering the parameters of a simple network (weights and biases are zero for simplicity) with one hidden layer like the one that was used in forward propagation in Figure 10, represented by $W^{(1)}$ and $W^{(2)}$, shows the aim of backpropagation is to compute the gradients $\frac{\partial L}{\partial W^{(1)}}$ and $\frac{\partial L}{\partial W^{(2)}}$. The chain rule is employed to achieve this, sequentially calculating the gradient of each intermediate variable and parameter. Notably, the order of computation is reversed compared to forward propagation; they start from the output of the computational graph and proceed toward the parameters. So below is the process of backpropagation for the neural network in Figure 10:

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = prod\left(\frac{\partial L}{\partial \mathbf{y_2}}, \frac{\partial \mathbf{y_2}}{\partial \mathbf{W^{(2)}}}\right) = \frac{\partial L}{\partial \mathbf{y_2}}\mathbf{y_1}^\top, \tag{2.41}$$

$$\frac{\partial L}{\partial \mathbf{y_1}} = prod\left(\frac{\partial L}{\partial \mathbf{y_2}}, \frac{\partial \mathbf{y_2}}{\partial \mathbf{y_1}}\right) = \mathbf{W^{(2)}}^\top \frac{\partial L}{\partial \mathbf{y_2}}, \tag{2.42}$$

$$\frac{\partial L}{\partial \mathbf{o}} = prod\left(\frac{\partial L}{\partial \mathbf{y_1}}, \frac{\partial \mathbf{y_1}}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{y_1}} \odot a^{'}(\mathbf{o}), \tag{2.43}$$

$$\frac{\partial L}{\partial \mathbf{W^{(1)}}} = prod\left(\frac{\partial L}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W^{(1)}}}\right) = \frac{\partial L}{\partial \mathbf{o}}\mathbf{x}^\top, \tag{2.44}$$

$\odot$ is the elementwise multiplication operator because the activation function is applied elementwise. This process is depicted in Figure 11.

During the training process of neural networks, forward and backward propagation are interdependent processes. During forward propagation, we navigate the computational graph following dependencies and calculate all variables along the path. These calculated variables are then utilized in backpropagation, where the computation order on the graph is reversed. [12]

Hence, during neural network training, after initializing model parameters, we alternate between forward and backward propagation, updating model parameters using gradients derived from backpropagation. Notably, backpropagation optimizes memory usage by reusing stored intermediate values from forward propagation, thus preventing redundant computations. However, this also means that intermediate values must be retained until backpropagation concludes, contributing to the increased memory demands during training compared to prediction tasks. Additionally, the size of these intermediate values is roughly proportional to the network's depth and the batch size. Consequently, training deeper networks with larger batch sizes can more easily lead to out-of-memory errors. [12]

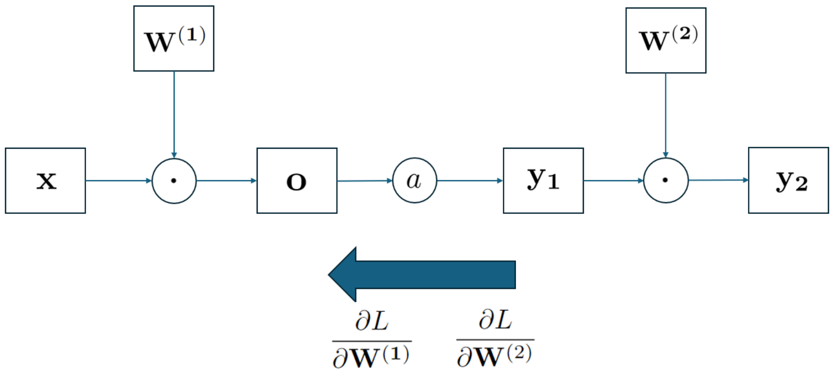


Figure 11: Neural network with a single hidden layer (Backpropagation)

3 Physics Informed Neural Networks

3.1 Scientific Machine Learning

The rapid expansion of accessible data and computing resources has led to significant advancements in machine learning and data analytics, resulting in transformative breakthroughs across various scientific domains such as image recognition [31], cognitive science [32], and genomics [33]. In the field of computational fluid dynamics (CFD), deep learning has been used to solve the Navier-Stokes equations [34] and flows with turbulence. [35] More specifically, in one study, it was discovered that higher Reynolds numbers did not affect the computational cost when deep learning was utilized instead of traditional numerical methods. [36]

Frequently, when analyzing intricate physical, biological, or engineering systems, the expense of acquiring data becomes a barrier, leading us to confront the task of drawing conclusions and making decisions with incomplete information. In such scenarios with limited data, the majority of advanced machine learning methods, including deep, convolutional, or recurrent neural networks, often lack robustness and do not ensure convergence. [37]

In the realm of scientific machine learning, nonlinear problems can be directly addressed without the necessity of committing to any prior assumptions, linearization, or local time stepping. Recent advancements in automatic differentiation, a highly valuable yet often underutilized technique in scientific machine computing, enable the differentiation of neural networks concerning their input coordinates and model parameters. This facilitates the development of **physics-informed neural networks** capable of respecting symmetries, invariances, or conservation principles derived from the physical laws governing observed data, as characterized by general time-dependent and nonlinear partial differential equations. This straightforward yet potent approach enables tackling a broad spectrum of problems in computational science, introducing potentially transformative technology that fosters the creation of new data-efficient and physics-informed learning machines, novel classes of numerical solvers for partial differential equations, and innovative data-driven approaches for model inversion and systems identification. [37] This thesis explores two primary categories of physics-informed neural networks: data-driven solutions for partial differential equations (PDEs) and parameter inference while solving a PDE.

3.1.1 Data-driven solutions of PDEs

A general form of a PDE is:

$$u_t + N[u] = 0, \tag{3.1}$$

where $u(t,x)$ is the hidden solution of the PDE, $x \in \Omega$ and $t \in [0,T]$,$N[\cdot]$ is defined as a non-linear differential operator and Ω is a subset of $\mathbb{R}$. Now let's consider $f(t,x)$ as the left hand side of the equation

$$f := u_t + N[u]. \tag{3.2}$$

A physics-informed neural network is obtained based on this assumption and the equation presented earlier. This network is derived by employing the chain rule for differentiating compositions of functions using automatic differentiation. The parameters within the neural network $u(t,x)$ and $f(t,x)$ can be trained by minimizing the mean squared loss:

$$MSE = MSE_u + MSE_f, \tag{3.3}$$

where

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2, \tag{3.4}$$

and

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2, \tag{3.5}$$

where:

- $(t_u^i, x_u^i, u^i)_{i=1}^{N_u}$ are the initial and boundary data on $u(t,x)$,
- $(t_f^i, x_f^i)_{i=1}^{N_f}$ are the collocation points in which the equation $f = 0$ is solved.

The loss function MSE_u accounts for the consistency with the initial and boundary data, while MSE_f ensures adherence to the structural constraints imposed by the equation at specific collocation points in a finite set.

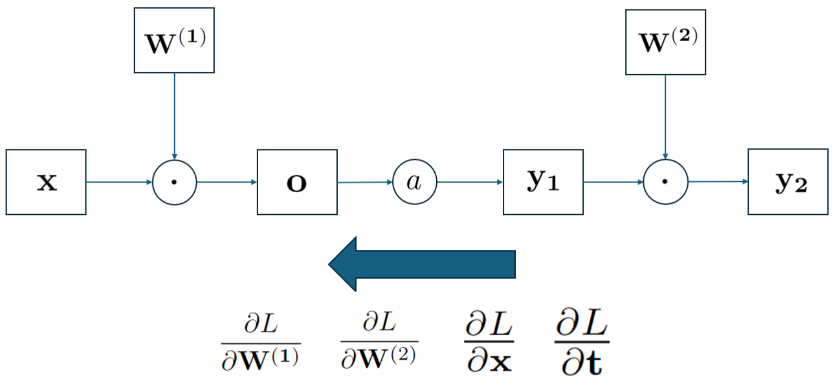
In physics-informed neural networks, an understanding and appreciation of the pivotal role of automatic differentiation in deep learning opens up new avenues. Automatic differentiation, particularly through the back-propagation algorithm [38], is the predominant method for training deep models by computing their derivatives concerning parameters such as weights and biases. Here, the very same techniques of automatic differentiation are harnessed to inform neural networks about physics by computing derivatives concerning their input coordinates (i.e., space and time), where the physics is delineated by partial differential equations. Raissi et al.[37] empirically observed that this structured approach introduces a regularization mechanism, enabling the utilization of relatively compact feed-forward networks. Consequently, these neural networks can be trained effectively even with limited data. This master's thesis delves into the examination of continuous-time models within this framework.

In Figure 15, the example from the subsections explaining forward propagation (1.2.3,1.2.4) is revisited, this time showing that during the process of backpropagation in a physics-informed neural network, the derivatives of the loss function

L are calculated with respect to the inputs. If the inputs are the points regarding space x and points regarding time t, then the extra gradients to be computed are

$$\frac{\partial L}{\partial \mathbf{x}}, \tag{3.6}$$

$$\frac{\partial L}{\partial \mathbf{t}}. \tag{3.7}$$



$\mathbf{W}^{(1)}$
$\mathbf{W}^{(2)}$
$\mathbf{x}$
$\cdot$
$\mathbf{o}$
a
$\mathbf{y_1}$
$\cdot$
$\mathbf{y_2}$
$\frac{\partial L}{\partial \mathbf{W}^{(1)}}$
$\frac{\partial L}{\partial \mathbf{W}^{(2)}}$
$\frac{\partial L}{\partial \mathbf{x}}$
$\frac{\partial L}{\partial \mathbf{t}}$

Figure 12: Physics Informed Neural network with a single hidden layer (Backpropagation)

3.1.2 Parameter Inference

If equation (3.2) had an unknown parameter then its general form would be

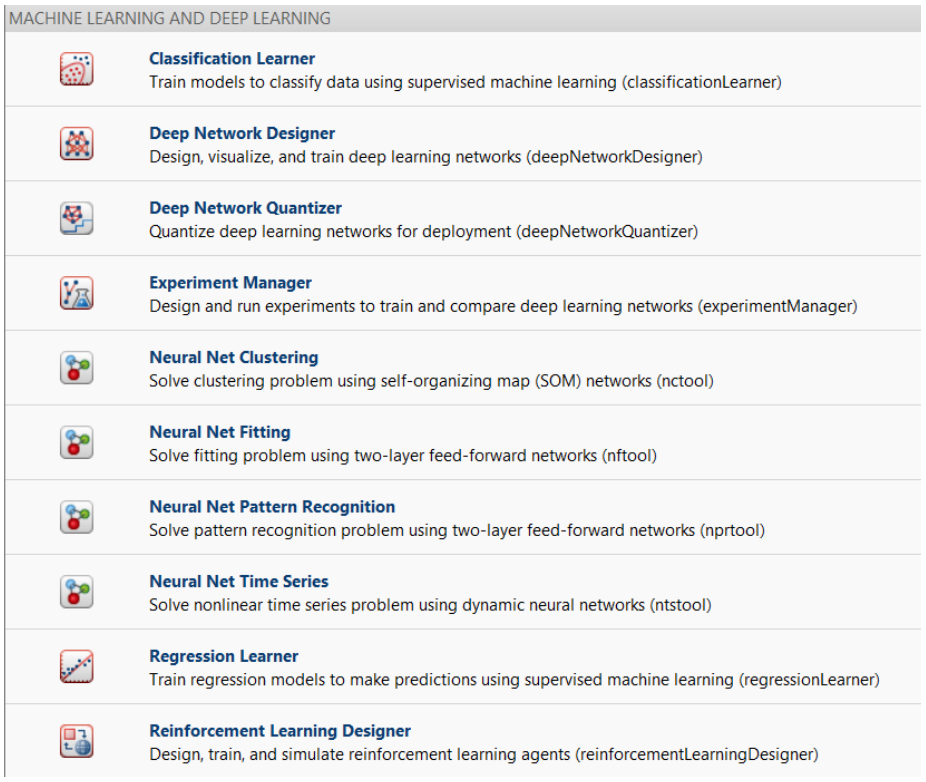
$$f := u_t + N[u; \lambda]. \tag{3.8}$$

The physics-informed neural network can conduct parameter inference for λ during training by providing a small amount of data other than the initial and boundary conditions. In order to gain a better insight the example of Burger's equation is used for both cases.

3.2 Example: Burger's Equation

3.2.1 MATLAB: Deep Learning Toolbox

To utilize Machine Learning and, more specifically, Deep Learning in MATLAB, it is necessary to download the Deep Learning Toolbox. This toolbox provides various methods, ranging from a user-friendly graphical interface (as seen in Figure 13) to constructing networks through coding (which is the approach utilized in this thesis).


MACHINE LEARNING AND DEEP LEARNING

Classification Learner
Train models to classify data using supervised machine learning (classificationLearner)

Deep Network Designer
Design, visualize, and train deep learning networks (deepNetworkDesigner)

Deep Network Quantizer
Quantize deep learning networks for deployment (deepNetworkQuantizer)

Experiment Manager
Design and run experiments to train and compare deep learning networks (experimentManager)

Neural Net Clustering
Solve clustering problem using self-organizing map (SOM) networks (nctool)

Neural Net Fitting
Solve fitting problem using two-layer feed-forward networks (nftool)

Neural Net Pattern Recognition
Solve pattern recognition problem using two-layer feed-forward networks (nprtool)

Neural Net Time Series
Solve nonlinear time series problem using dynamic neural networks (ntstool)

Regression Learner
Train regression models to make predictions using supervised machine learning (regressionLearner)

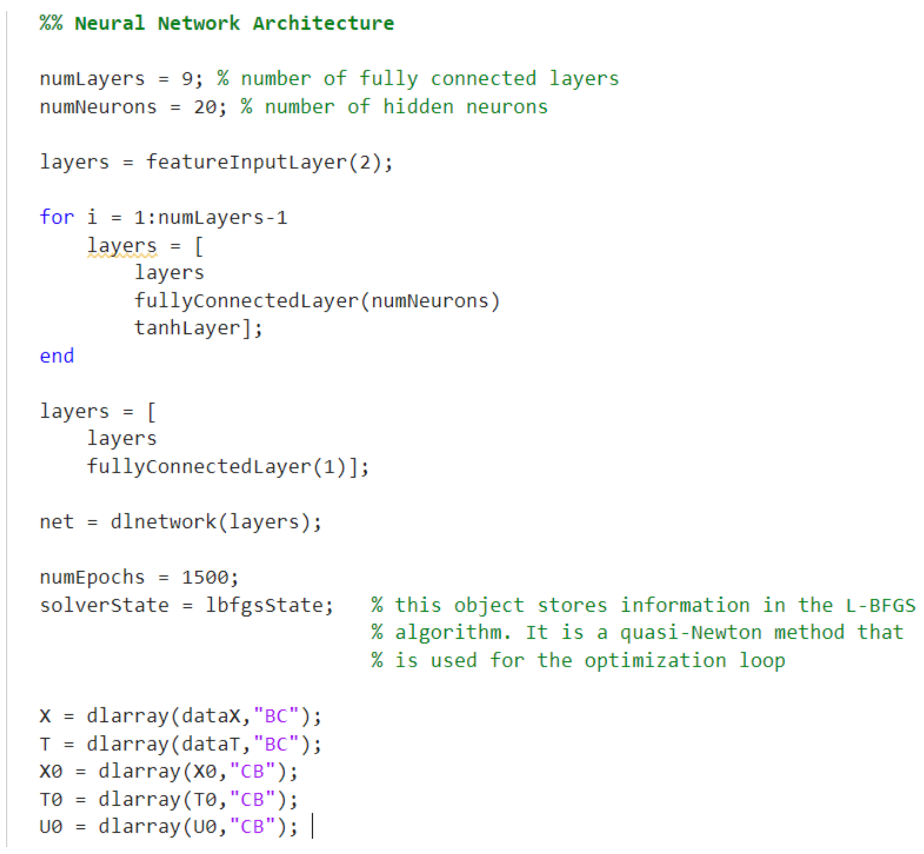
Reinforcement Learning Designer
Design, train, and simulate reinforcement learning agents (reinforcementLearningDesigner)

Figure 13: User-friendly apps from Deep Learning Toolbox MATLAB

Leveraging MATLAB's Deep Learning Toolbox, users can compute gradients effortlessly through automatic differentiation.

Typically, the most straightforward approach to tailoring deep learning training involves constructing a *dlnetwork*. This entails assembling the desired layers within the network structure, followed by executing training within a custom loop employing an optimizer (Gradient Descent, Adam, the LBFGS, etc.).

In Figure 14 is a code snippet for the neural network used for solving Burger's Equation (see subsection 3.2.2). The neural network is built (choosing collocation points, types and number of layers and neurons, number of epochs) and then passed through the function *dlnetwork* to be utilized by the Deep Learning Toolbox with the chosen optimizer LBFGS. Additionally, the data used for training needs to be converted *dlarray* to be understood by the Toolbox, facilitating data structure management and enabling evaluation tracing.

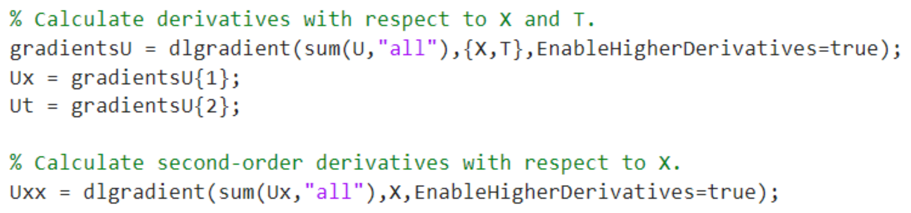


```
%% Neural Network Architecture

numLayers = 9; % number of fully connected layers
numNeurons = 20; % number of hidden neurons

layers = featureInputLayer(2);

for i = 1:numLayers-1
    layers = [
        layers
        fullyConnectedLayer(numNeurons)
        tanhLayer];
end

layers = [
    layers
    fullyConnectedLayer(1)];

net = dlnetwork(layers);

numEpochs = 1500;
solverState = lbfgsState;   % this object stores information in the L-BFGS
                            % algorithm. It is a quasi-Newton method that
                            % is used for the optimization loop

X = dlarray(dataX,"BC");
T = dlarray(dataT,"BC");
X0 = dlarray(X0,"CB");
T0 = dlarray(T0,"CB");
U0 = dlarray(U0,"CB");
```

Figure 14: Code snippet showing the functions *dlnetwork* and *dlarray*

To harness automatic differentiation capabilities, it's essential to employ *dlgradient* within a function and conduct function evaluation using *dlfeval*.

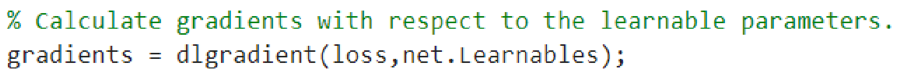
In the following example for Burger's equation, within the *modeloss* function, *dlgradient* is employed to compute first and second-order spatial gradients (U_x,U_{xx}), along with the temporal gradient (U_t). The code snippet in Figure 15 illustrates the computation of these derivatives throughout the training phase:



```
% Calculate derivatives with respect to X and T.
gradientsU = dlgradient(sum(U,"all"),{X,T},EnableHigherDerivatives=true);
Ux = gradientsU{1};
Ut = gradientsU{2};

% Calculate second-order derivatives with respect to X.
Uxx = dlgradient(sum(Ux,"all"),X,EnableHigherDerivatives=true);
```

Figure 15: Code snippet from Matlab for computing the gradients with automatic differentiation utilized by the PINN

As discussed in earlier chapters, the network learnables encompass the weights and biases. These parameters are recalculated utilizing the *dlgradient* and *dleval* pair as depicted in Figure 16. However, it's important to note that while the neural network employs these gradients for optimization, the gradients mentioned in Figure 15 are derived through backpropagation to compute the gradients present in the equation.



```
% Calculate gradients with respect to the learnable parameters.
gradients = dlgradient(loss,net.Learnables);
```

Figure 16: Code snippet from Matlab for computing the gradients with automatic differentiation for optimization

3.2.2 Solution of Burger's Equation

Demonstrated herein is the application of deep learning techniques to address Burger's equation, a partial differential equation (PDE) prevalent across various domains of applied mathematics. Its relevance spans disciplines such as fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flow analysis.

The Burger's equation is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \frac{0.01}{\pi}\frac{\partial^2 u}{\partial x^2} = 0. \tag{3.9}$$

The boundary conditions are:
for $x = -1$:

$$u(x = -1, t) = 0, \tag{3.10}$$

for $x = 1$:

$$u(x = 1, t) = 0, \tag{3.11}$$

and the initial conditions are

$$u(x, t = 0) = -sin(\pi x). \tag{3.12}$$

This approach uses neural networks to eliminate the need for grid generation, making it a meshless method. Instead, a set of collocation points is employed to solve equations within the spatiotemporal domain. These collocation points are also strategically utilized to enforce both the boundary and initial conditions, ensuring their respective values during the training process.

For instance, for $x = -1$ and $x = 1$, 25 equally spaced collocation points each are utilized, while 50 equally spaced collocation points are employed for enforcing the initial conditions. In total, Burger's equation is solved using 10,000 collocation points that entail both time and space directions. No additional data is required, as the neural network demonstrates satisfactory performance with the aforementioned dataset.

The neural network architecture consists of 9 layers, each containing 20 neurons. Training is conducted over 1500 epochs using the LBFGS optimizer. The loss function is formulated by incorporating a function referred to as the *modelloss* function, which comprises two terms derived from equations (3.4) and (3.5).

The reduction of the loss function during the training progress is seen in Figure 17:

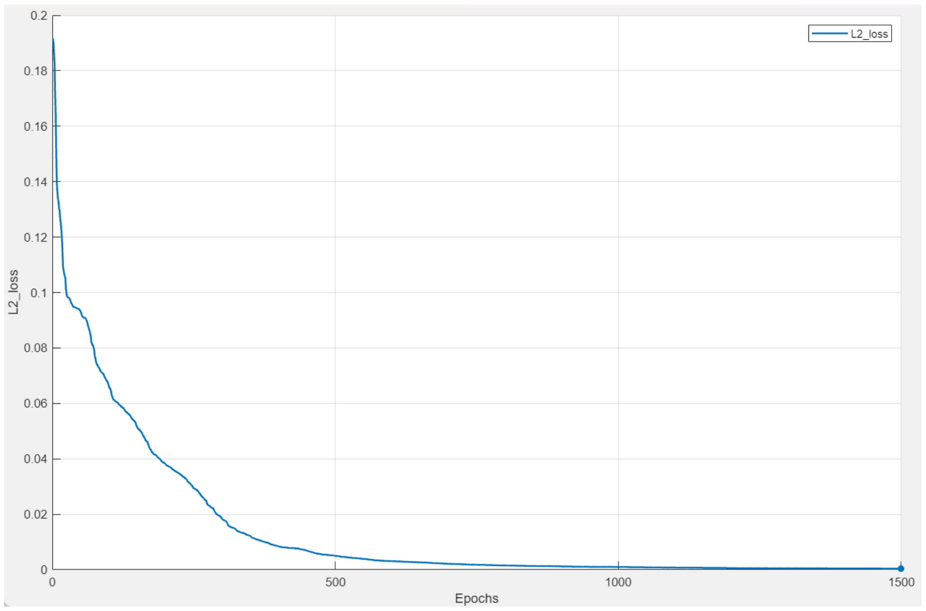


Figure 17: Reduction of loss function during the training process

The solution from the PINN was compared with the solution exported from Comsol simulation software, which utilizes the finite element method to solve the equation numerically. For this specific example, the linear solver MUMPS was chosen, along with the Damped Newton Non-Linear Method and BDF time stepping. Additionally, quadratic basic functions were chosen along with 10000 domain elements. A tolerance factor of 0.1 was chosen. The solution was compared for four different time profiles: 0.25 sec, 0.5 sec, 0.75 sec, and 1 sec, which are shown in Figure 18. It is evident that the neural network was able to solve the equation successfully, like the finite element method.

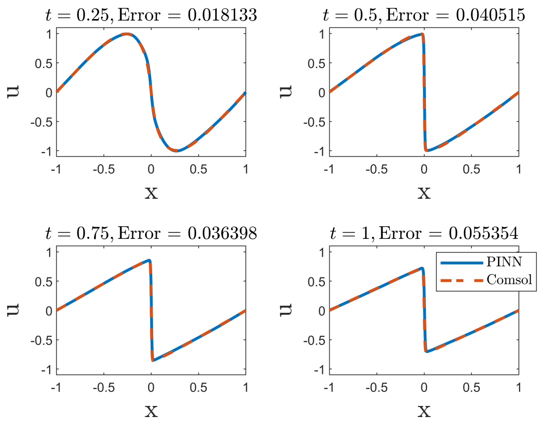


Figure 18: Results for Burger's Equation using a PINN for specific boundary conditions(3.10,3.11) and initial conditions (3.12)

3.2.3 Parameter Inference in the Burger's Equation

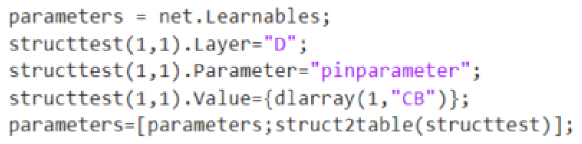
The preceding section demonstrated solving Burger's equation [39] through the provision of initial and boundary conditions and the specific points for solving the equation. One notable advantage of physics-informed neural networks lies in their capacity to deduce unknown parameters within equations. For instance, let's consider the scenario where the parameter D in the following equation is unknown

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - D\frac{\partial^2 u}{\partial x^2} = 0, \tag{3.13}$$

and the real value of D is

$$D_{real} = 0.01/\pi = 0.00318. \tag{3.14}$$

The data provided was the value of u at 0.1, 0.25, 0.5, 0.75, and 0.85 sec for the entire spatial domain. In this case, D is a learnable parameter along with the weights and biases. That means that in the Matlab environment, the parameter D needs to be converted into a structured array and inserted into the network, and its corresponding gradient will be calculated along with the weight and bias gradients. In Figure 19 is the main code snippet, demonstrating the process of converting the parameter that we want to infer into a structured dlarray.



```
parameters = net.Learnables;
structtest(1,1).Layer="D";
structtest(1,1).Parameter="pinparameter";
structtest(1,1).Value={dlarray(1,"CB")};
parameters=[parameters;struct2table(structtest)];
```

Figure 19: MATLAB code snippet for parameter inference

The process of inference is seen in Figure 20. The initial value was $D = 1$. It must be noted that 5000 epochs were used to ensure the best results. It is evident that after a few epochs, the value drops below 0.1:

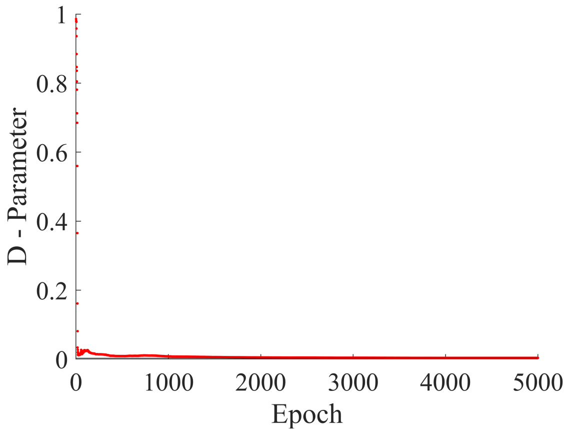


Figure 20: Parameter inference during training for the Burger's Equation

The final value of D from inference is

$$D_{PINN} = 0.003751. \tag{3.15}$$

The relative error is

$$Error_D = \left|\frac{D_{PINN} - D_{real}}{D_{real}}\right| = 0.1784. \tag{3.16}$$

During parameter inference the equation is solved as well and the results can be in Figure 21:

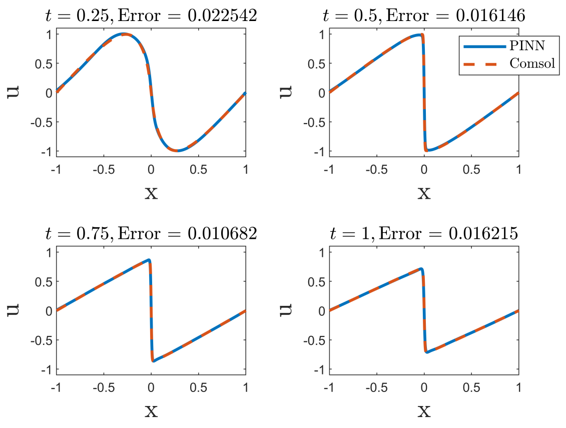


Figure 21: Results for Burger's equation using a PINN with parameter inference

3.3 The advantages and disadvantages of PINNs compared to classical numerical methods

Neural network-based regression techniques have emerged as effective and straightforward solutions. Physics-informed learning (PINN) seamlessly merges principles from physics with machine learning, facilitating the integration of information from both physics laws and scattered noisy data, even under imperfect conditions. A recent study has showcased the ability of PINN to derive meaningful solutions, even in scenarios where the problem lacks perfect, well-posedness due to inherent smoothness or regularity in its formulation. [40]

Furthermore, unlike conventional numerical approaches, physics-informed learning stands out as mesh-free, eliminating the need for computationally intensive mesh generation. This characteristic empowers it to effectively address irregular and moving-domain problems. [41] While deep learning typically requires abundant training data, obtaining such data with high accuracy remains challenging for many physical problems. In such cases, physics-informed learning offers a distinct advantage by demonstrating robust generalization capabilities in scenarios with limited data. [42]

By integrating or embedding physics principles into deep learning models, physics-informed learning effectively restricts the models to a lower-dimensional manifold, enabling training with small datasets. Additionally, physics-informed learning extends its capabilities beyond mere interpolation to encompass extrapolation, facilitating spatial extrapolation in boundary-value problems. [43]

Additionally, the practical limits of numerical computation often stem from the high dimensionality of problems. Termed the **curse of dimensionality**, this phenomenon signifies that the minimal computational cost of approximating a solution increases **exponentially** with the problem's dimensionality. [44] Consequently, many numerical problems characterized by high dimensions become practically insurmountable.

However, deep learning has demonstrated remarkable success in addressing high-dimensional problems, such as fine-resolution image classification, language modeling, and tackling high-dimensional PDEs. [42] One contributing factor to this success lies in the ability of deep neural networks to mitigate the curse of dimensionality, provided that the loss function exhibits a hierarchical composition of local functions. [45] [46]

Nevertheless, the realm of PINNs encounters several obstacles. Firstly, there remains a lack of clarity regarding the precision and convergence concerning adjustable parameters. Furthermore, the optimization strategies necessary to equate PINNs with other computational tools in efficiency remain elusive. [47] One study [48] noted a deficit in PINN accuracy attributable to the absence of established activation functions and specialized architectures, both of which significantly impact final model accuracy. Nonetheless, PINNs demonstrate accuracy in parameter inference. Conversely, another study [49] shed light on the potential limitations of PINNs in solving PDEs characterized by strong non-linearity or high-frequency solutions. To mitigate this, pre-training techniques were suggested to enhance efficiency. Specifically, pre-training involves training a neural network on smaller subdomains to provide initialization and additional supervised learning data for larger subdomains or entire temporal domains, thus enhancing overall performance.

4 Application for Reaction-Diffusion PDEs modeling an avascular growing tumor

4.1 Physical Model

The physical model that was chosen for the application was a reaction-diffusion problem. A simple system was selected that simulated the evolution of cancer cells in a growing tumor at its first stages of evolution, so it is still considered avascular. Vasculature is vital in malignant tumors, as blood vessels formed through angiogenesis induced by tumor cells are a precursor to metastasis. [50] There are two independent variables, space x and time t, and two dependent variables, cancer cells concentration z and oxygen concentration c. Cancer cells consume oxygen for proliferation; therefore, these two variables are interconnected with a system of equations that can be seen in equations 4.1 and 4.2. It must be noted that the growing tumor is spherically symmetrical.

$$\frac{\partial z}{\partial t} - d_z \frac{1}{r^2} \frac{\partial^2 z}{\partial r^2} = k_1 z \frac{c}{c + c_p} - k_2 z \frac{c + c_{c1}}{c + c_{c2}}, \tag{4.1}$$

$$-d_c \frac{1}{r^2} \frac{\partial^2 c}{\partial r^2} = -k_s z c - k_m z \frac{c}{c + c_p}. \tag{4.2}$$

These non-dimensional equations were derived from the work of Lampropoulos et al. [50][51]. On the left-hand side of these equations are terms regarding the transport phenomena, and on the right-hand side are the source terms. In the scope of this thesis, the transport phenomena were limited to diffusion.

The source terms in equation (4.1) for the cancer cells are:

- $k_1 z \frac{c}{c+c_p}$: Michaelis-Menten kinetics for the production (mitosis) of cancer cells. The cancer cell population consumes the substrate, which, in this case, is oxygen. c_p is a constant and has a value which is half of the maximum value of c.
- $k_2 z \frac{c+c_{c1}}{c+c_{c2}}$: This term regards cellular death, and it represents the programmed death of a cell from natural causes. It must be noted that $c_{c1} > c_{c2}$ to ensure that the cellular death rate is increased when c becomes scarce.

The source terms in equation (4.2) for oxygen are:

- $k_s z c$: This term represents sustenance, and it is a second-order reaction between the cancer cells and oxygen.
- $k_m z \frac{c}{c+c_p}$: Oxygen that is consumed from the production (mitosis) of cells.

The values for the constants are provided in Table 1:

Table 1: Parameter names and values

Parameter	Value	Description
d_z	0.01	Cancer cells diffusion coefficient
d_c	1	Oxygen diffusion coefficient
k_1	0.5	Cancer cell mitosis rate
k_2	0.15	Cancer cell death rate
c_{c1}	0.3	Oxygen concentration regulating cellular death rate
c_{c2}	0.2	Threshold oxygen concentration for cellular death rate
c_p	0.5	Threshold oxygen concentration for mitosis
k_s	0.1	Consumption rate constant for sustenance of cancer cells
k_m	0.1	Consumption rate constant of oxygen used for mitosis of healthy cells

The oxygen has no temporal gradient in equation (4.2) since it is considered a quasi-steady state equation because time scales of chemical species are significantly shorter compared to cellular processes. Finally, it must be remarked that this system of equations is based on spherical coordinates.

The chosen radial domain r is

$$r \in [0, 8], \tag{4.3}$$

and the chosen time t interval is

$$t \in [0, 20]. \tag{4.4}$$

The boundary conditions for the cancer cells are:

for r = 0 zero flux, Neumann conditions are chosen, meaning that there is no diffusion of cancer cells:

$$\frac{\partial z}{\partial x}\bigg|_{r=0} = 0, \tag{4.5}$$

for r = 8 Dirichlet conditions are chosen that correspond to the boundary of a tissue within a cancerous tumor grows:

$$z\bigg|_{r=8} = 0. \tag{4.6}$$

The boundary conditions for oxygen are:

for r = 0 zero flux, Neumann conditions are chosen, meaning that there is no diffusion of oxygen:

$$\frac{\partial c}{\partial x}\Big|_{r=0} = 0, \tag{4.7}$$

for r = 8 Dirichlet conditions are chosen that correspond to the boundary of a tissue within a cancer grows and oxygen is in abundance:

$$c\Big|_{r=8} = 1, \tag{4.8}$$

and the initial conditions for the cancer cells (for $r < 5$):

$$z_{initial} = \left(cos(\frac{\pi r}{10})\right)^2. \tag{4.9}$$

The initial conditions for c are calculated from equation (4.2). Both initial conditions are depicted in Figure 22:

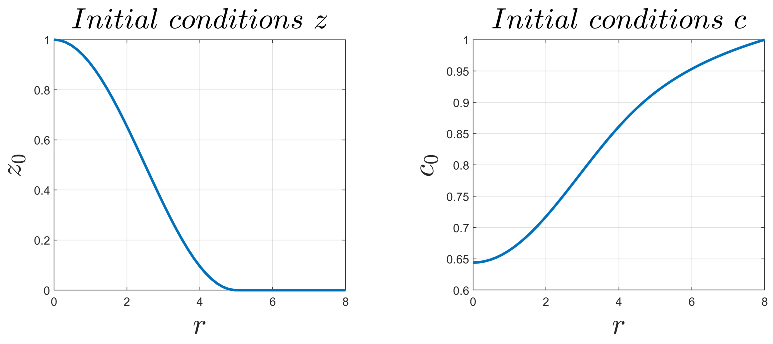


Figure 22: Initial conditions for cancer cells and oxygen

4.2 Loss Function and Dynamic Weights

The loss function for the system of PDEs is

$$L = MSE_{z_0}+MSE_{c_0}+MSE_{f_c}+MSE_{f_z}+MSE_{z_D}+MSE_{c_D}+MSE_{z_N}+MSE_{z_N}, \tag{4.10}$$

the mean squared error for the initial conditions of z with N_{z_0} points which solves the initial condition constraint:

$$MSE_{z_0} = \frac{1}{N_{z_0}} \sum_{i=1}^{N_{z_0}} |z_0^{PINN(i)} - z_0^i|^2, \tag{4.11}$$

the mean squared error for the initial conditions of c with N_{c_0} points which solves the initial condition constraint:

$$MSE_{c_0} = \frac{1}{N_{c_0}} \sum_{i=1}^{N_{c_0}} |c_0^{PINN(i)} - c_0^i|^2, \tag{4.12}$$

the mean squared error for the equation regarding the cancer cells, which solves the PDE f_z

$$f_z = \frac{\partial z}{\partial t} - d_z \frac{1}{r^2}\frac{\partial^2 z}{\partial r^2} - k_1 z \frac{c}{c+c_p} + k_2 z \frac{c+c_{c1}}{c+c_{c2}}, \tag{4.13}$$

is (the number of collocation points is N_{col})

$$MSE_{f_z} = \frac{1}{N_{col}} \sum_{i=1}^{N_{col}} |f_z^i|^2, \tag{4.14}$$

the mean squared error for the equation regarding oxygen, which solves the PDE f_c

$$f_c = -d_c \frac{1}{r^2}\frac{\partial^2 c}{\partial r^2} + k_s z c + k_m z \frac{c}{c+c_p}, \tag{4.15}$$

is (the number of collocation points is N_{col})

$$MSE_{f_c} = \frac{1}{N_{col}} \sum_{i=1}^{N_{col}} |f_c^i|^2, \tag{4.16}$$

the mean squared error for the Dirichlet boundary condition of z with N_{z_D} points which solves the boundary condition constraint:

$$MSE_{z_D} = \frac{1}{N_{z_D}} \sum_{i=1}^{N_{z_D}} |z_D^{PINN(i)} - z_D^i|^2, \tag{4.17}$$

the mean squared error for the Dirichlet boundary condition of c with N_{c_D} points which solves the boundary condition constraint:

$$MSE_{c_D} = \frac{1}{N_{c_D}} \sum_{i=1}^{N_{c_D}} |c_D^{PINN(i)} - c_D^i|^2, \tag{4.18}$$

the mean squared error for the Neumann boundary condition of z with N_{z_N} points which solves the boundary condition constraint:

$$MSE_{z_N} = \frac{1}{N_{z_N}} \sum_{i=1}^{N_{z_N}} |z_N^{PINN(i)} - z_N^i|^2, \tag{4.19}$$

the mean squared error for the Neumann boundary condition of c with N_{c_N} points which solves the boundary condition constraint:

$$MSE_{c_N} = \frac{1}{N_{c_N}} \sum_{i=1}^{N_{c_N}} |c_N^{PINN(i)} - c_N^i|^2. \tag{4.20}$$

This is the traditional approach for the loss function of a PINN. However, during the training of PINNs, an imbalance in the gradients within the loss function can occur. This imbalance can hinder the performance of PINNs. A study [52] introduced a new technique for dynamically adjusting the weights of loss terms, allowing for balanced gradients in each term during training. The neural network identifies the training data that are harder to learn and shifts its focus to them before moving to the next training step. It recalibrates the weights for challenging data to enhance the objective function. These dynamic weights increase steadily and reach a steady point during training. This technique accelerates loss convergence, reduces generalization errors, and enhances computational efficiency.

So, the loss function with the dynamic weights is

$$\begin{aligned} L =& w_1^d MSE_{z_0} + w_2^d MSE_{c_0} + w_3^d MSE_{f_c} + w_4^d MSE_{f_z} \\ &+ w_5^d MSE_{z_D} + w_6^d MSE_{c_D} + w_7^d MSE_{z_N} + w_8^d MSE_{z_N}. \end{aligned} \tag{4.21}$$

The dynamic weights are calculated with the stochastic gradient ascent. They are initialized with a small non-negative value and increased at each iteration until the ideal values are reached:

$$w_1^{d(new)} = w_1^{d(old)} + \eta_d \frac{\partial L}{\partial w_1^{d(old)}}, \tag{4.22}$$

$$w_2^{d(new)} = w_2^{d(old)} + \eta_d \frac{\partial L}{\partial w_2^{d(old)}}, \tag{4.23}$$

$$w_3^{d(new)} = w_3^{d(old)} + \eta_d \frac{\partial L}{\partial w_3^{d(old)}}, \tag{4.24}$$

$$w_4^{d(new)} = w_4^{d(old)} + \eta_d \frac{\partial L}{\partial w_4^{d(old)}}, \tag{4.25}$$

$$w_5^{d(new)} = w_5^{d(old)} + \eta_d \frac{\partial L}{\partial w_5^{d(old)}}, \tag{4.26}$$

$$w_6^{d(new)} = w_6^{d(old)} + \eta_d \frac{\partial L}{\partial w_6^{d(old)}}, \tag{4.27}$$

$$w_7^{d(new)} = w_7^{d(old)} + \eta_d \frac{\partial L}{\partial w_7^{d(old)}}, \tag{4.28}$$

$$w_8^{d(new)} = w_8^{d(old)} + \eta_d \frac{\partial L}{\partial w_8^{d(old)}}. \tag{4.29}$$

In this thesis, every weight is initialized at 0.1 and the learning rate is the same in each epoch $\eta_d = 1$.

4.3 Neural Network Architecture

The neural network for solving this system of equations had these characteristics:

- number of layers: 10
- number of neurons in each layer: 20
- number of internal collocation points: 500 (logarithmically spaced)
- number of points for initial conditions: 51
- number of points for boundary conditions for z and c (Neumann): 25
- number of points for boundary conditions for z and c (Dirichlet): 25
- number of epochs: 6500.
- optimizer: LBFGS

The chosen neural network was based on the network from Burger's equation example in section 3.2. The main objective was to minimize the number of collocation points while maintaining a simple architecture. Consequently, the parameters were continually adjusted to find the optimal balance between satisfactory results and a straightforward neural network with a minimal number of layers and neurons.

4.4 Results: PDE system solution

4.4.1 PINN solution of the system of PDEs

First, the solution of the system of equations is presented. The data provided are the dependent variables z and c at the initial and boundary conditions and the system of equations (Equations 4.1 and 4.2). Apart from that, no other data is required (for example, at another time step). The results are examined at t = 0, 10, and 20 to evaluate the neural network's performance. Furthermore, the results are compared with the results of the same problem exported from COMSOL, which uses the finite element method to solve the system of PDEs. In COMSOL the linear solver MUMPS was chosen, along with the Damped Newton Non-Linear Method and BDF time stepping. Additionally, quadratic basic functions were chosen along with 500 domain elements. A tolerance factor of 1 was chosen.

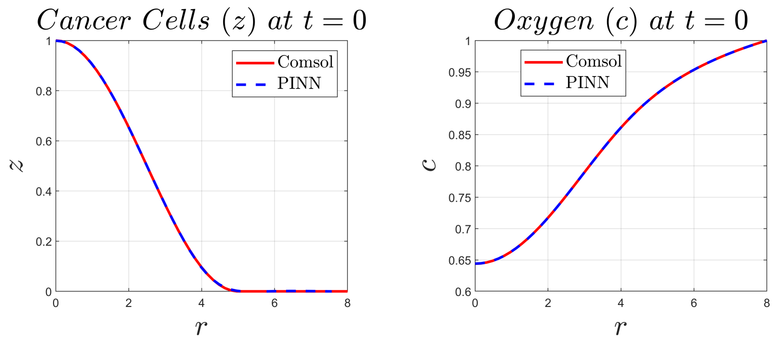


Figure 23: Comparison of initial conditions of the PINN and COMSOL solution

As expected, the neural network was able to fit the data remarkably well when predicting the initial conditions (depicted in Figure 23). The relative errors can be seen in Table 2:

Table 2: Initial Conditions at t = 0 for the entire spatial domain

Independent Variable	Relative Error
z	0.00197
c	0.0001

The solution for the system of PDEs at t = 10 (Figure 24 and 25):

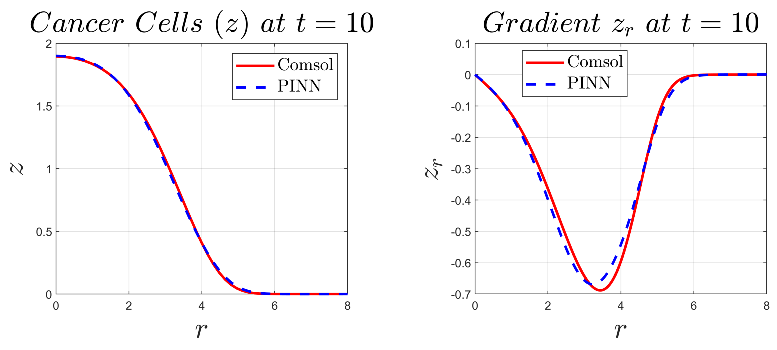


Figure 24: Comparison of cancer cells at t = 10 of the PINN and COMSOL solution

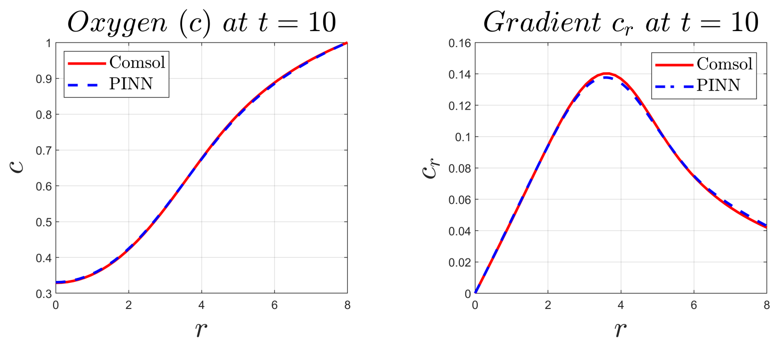


Figure 25: Comparison of oxygen at t = 10 of the PINN and COMSOL solution

and the relative errors can be seen in Table 3:

Table 3: z and c at t = 10 for the entire spatial domain

Independent Variable	Relative Error
z	0.01233
c	0.00276

The solution for the system of PDEs at t = 20 (Figure 26 and 27):

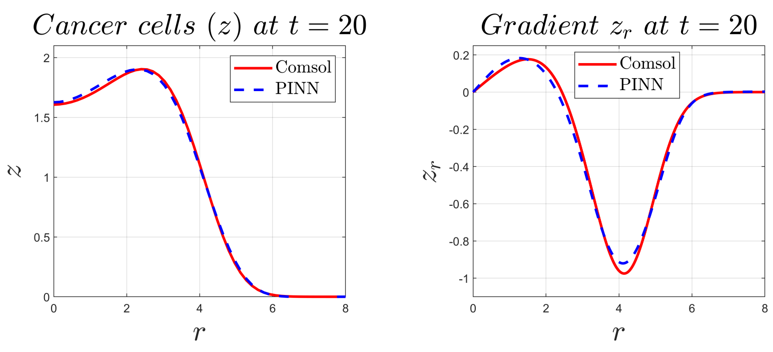


Figure 26: Comparison of cancer cells at t = 20 of the PINN and COMSOL solution

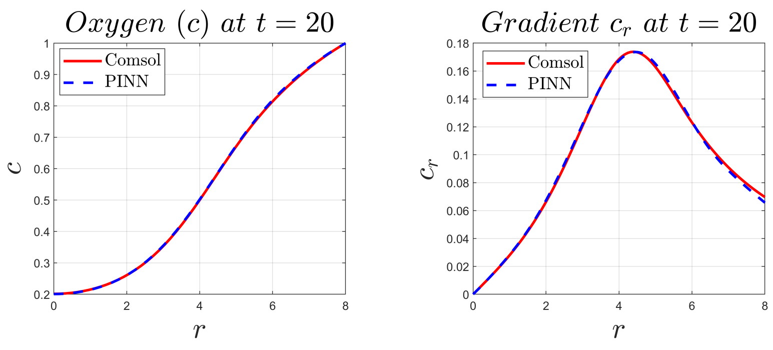


Figure 27: Comparison of oxygen at t = 20 of the PINN and COMSOL solution

and the relative errors can be seen in Table 4:

Table 4: z and c at t = 20 for the entire spatial domain

Independent Variable	Relative Error
z	0.01725
c	0.00326

Compared to the t = 10 results, the relative error seems to be slightly bigger at t = 20. However, the PINN seems to accurately predict the area in which cellular death is observed (r between 0 and 2).

The temporal gradients for t = 10 and t = 20 are shown in Figures 28 and 29:

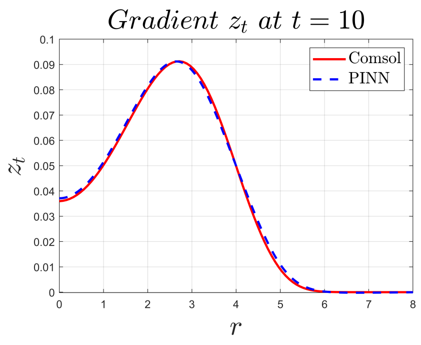


Figure 28: Temporal Gradient at t = 10

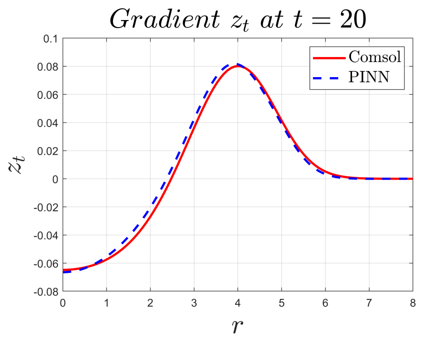


Figure 29: Temporal Gradient at t = 20

It is evident that the network can calculate the gradient z_t effectively.

Figure 30 depicts the spatiotemporal evolution of cancer cells concentration, z:

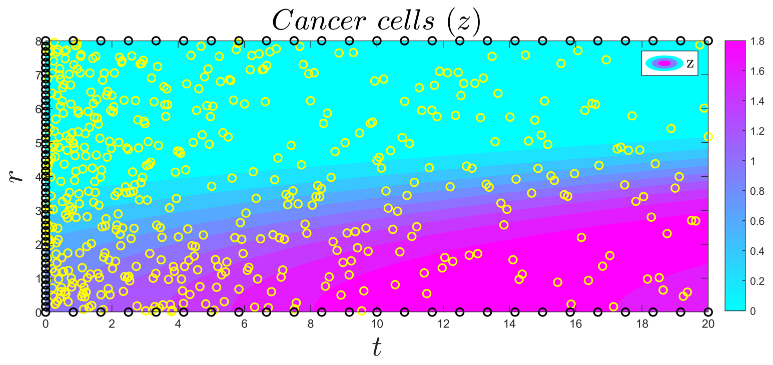


Figure 30: Contour graph of cancer cells dimensionless concentration. The black circles are the initial and boundary conditions, and the yellow circles are the collocation points at which the equations are solved.

It is evident from Figure 30 that the chosen collocation points are much more dense at the beginning of the phenomenon and become sparse towards the end at t = 20. This is important because it was observed that more collocation points were needed in the beginning to compute the gradient z_t. Figure 31 shows the corresponding spatiotemporal evolution of oxygen, c:

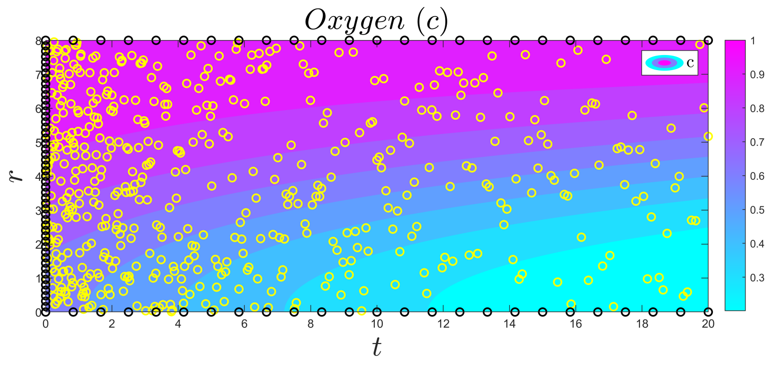


Figure 31: Contour graph of oxygen dimensionless concentration. The black circles are the initial and boundary conditions, and the yellow circles are the collocation points at which the equations are solved.

4.4.2 Extrapolation

It was mentioned that the PINN was trained within a certain spatial domain ($r \in [0, 8]$) and time interval ($t \in [0, 20]$). An extrapolation was done to examine if the PINN can predict evolution at larger times and if it can produce more reasonable results than other neural networks that might predict unphysical values. The neural network was tested at the extrapolated time of $t = 30$, and the results are provided in Figures 32 and 33:

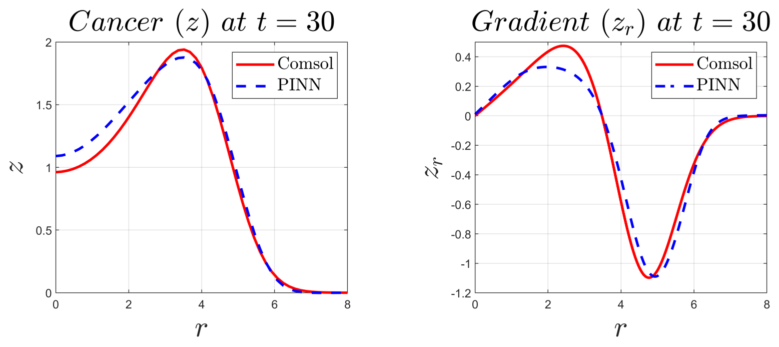


Figure 32: Comparison of cancer cells at t = 30 of the extrapolated PINN and COMSOL solution

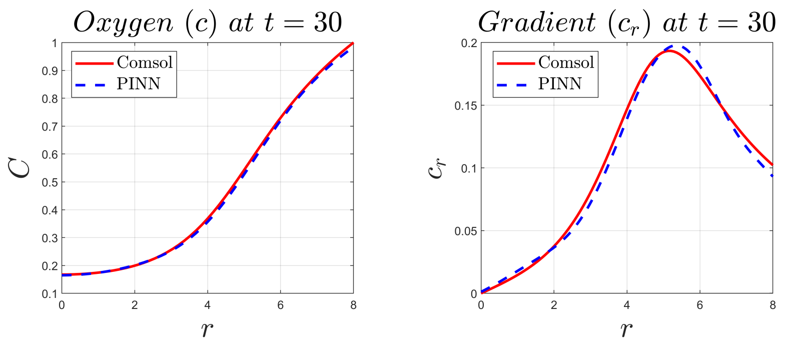


Figure 33: Comparison of oxygen at t = 30 of the extrapolated PINN and COMSOL solution

Table 5: Extrapolated z and c at t = 30 for the entire spatial domain

Independent Variable	Relative Error
z	0.07021
c	0.01694

The results (as seen in Table 5 from the relative errors) seem to be lacking when compared to the numerical results provided by COMSOL. It must be noted that the COMSOL data at $t = 30$ was not used during training, and Figures 32 and 33 are merely underlining the discrepancy in the results by comparing them. At $t = 30$ cellular death is observed in the range $r \in [0, 3.5]$ as seen in Figure 32. This is the area that the PINN mostly failed to predict compared to the numerical solution provided by COMSOL. However, this allows leeway for optimizing the neural network architecture with hyperparameter tuning or with a different neural network altogether.

4.5 Parameter Inference

For parameter inference, different cases were examined. These parameters are of great importance because they are usually **patient-specific** and are diffusion coefficients, which determine the transport properties. Unfortunately, they are usually unavailable, and we want to test PINNs' capability to predict their values using only a limited amount of data.

The PINN architecture remained the same; however, new terms had to be inserted in the loss function. These terms were the mean squared errors (MSE) that enforced the constraint of the extra provided data in the spatiotemporal domain during training. Moreover, the method of dynamic weights was implemented during inference as well, adding new dynamic weights for every new term in the loss function.

4.5.1 Case 1: Cancer cell and oxygen diffusion coefficient inference using cancer cell and oxygen data measurements

The initial and boundary conditions for z and c were provided in this first case. Additionally, the data from t = 5 for both dependent values (z and c) were provided in the entire spatial domain, and the parameters d_z and d_c were inferred. The spatiotemporal evolution of cancer cells and oxygen concentrations are depicted in Figures 34 and 35:

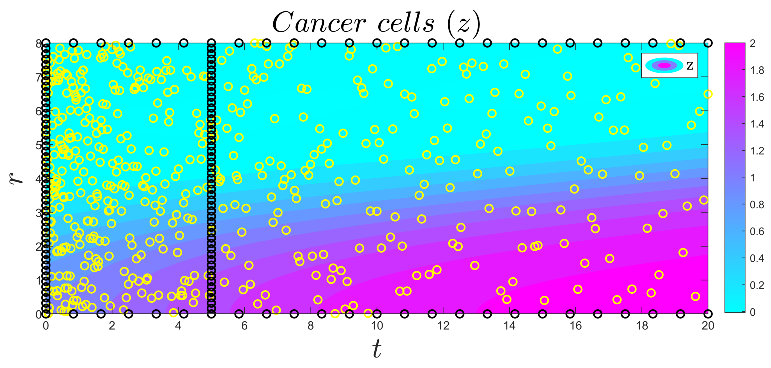


Figure 34: Contour graph of cancer cells dimensionless concentration. The black circles are the initial and boundary conditions and the distribution of z at t = 5. The yellow circles are the collocation points at which the equations are solved.

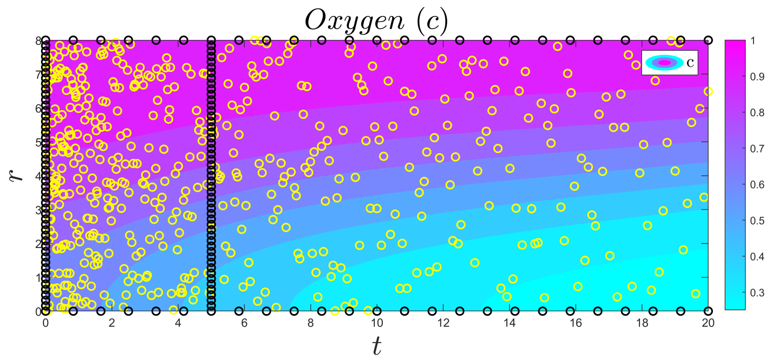


Figure 35: Contour graph of dimensionless oxygen concentration. The black circles are the initial and boundary conditions and the distribution of c at t = 5. The yellow circles are the collocation points at which the equations are solved.

Table 6: Parameter Inference for Case 1

Parameter	Actual Value	Inferred Value	Relative Error
d_z	0.01	0.0207	1.0740
d_c	1	1.0444	0.0444

It is evident from Table 6 that the relative value for d_c is satisfactory, whereas the relative value of d_z can be minimized further. This might be achieved by increasing the collocation points since they are relatively small and by increasing the epochs as well. In addition, the contribution of d_z in the $L2 - loss$ function is smaller because it's two orders of magnitude smaller compared to d_c.

Table 7: z and c at t = 20 for Case 1 for the entire spatial domain

Independent Variable	Relative Error
z	0.1672
c	0.059

The relative error for z (as depicted in Table 7) seems to be higher than the PINN solution of the system in section 4.4.1. Figure 34 and Figure 35 also show that, in this case, the necrosis of the cancer cells was not predicted. On the other hand, the error of c is low, but this is due to the fact that the inferred value of d_c is closer to its corresponding actual value.

4.5.2 Case 2: Cancer cell diffusion coefficient inference using cancer cell measurements

In the second case, a more realistic scenario is examined. Usually, in an experimental setup, collecting data for the cancer cells is much easier than it is for oxygen. Therefore, in this case study only the boundary conditions are known for oxygen (c), whereas for the cancer cells (z) the available data are the initial and boundary conditions and the distribution at t = 5. First, d_z is considered unknown and is the only parameter inferred in this case. In case 3, d_c will be inferred as well. The spatiotemporal evolution of cancer cells and oxygen concentrations are depicted in Figures 36 and 37

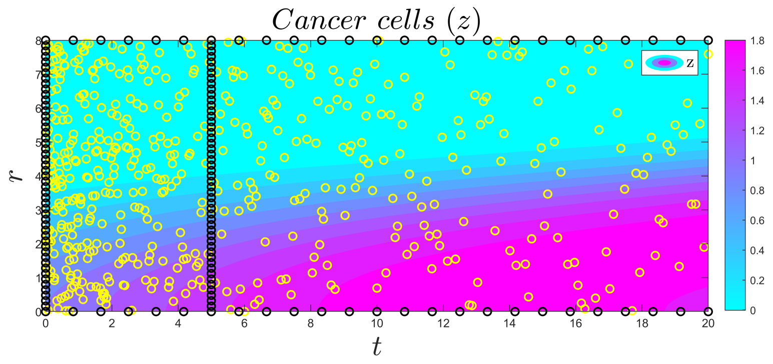


Figure 36: Contour graph of cancer cell dimensionless concentration. The black circles are the initial and boundary conditions and the distribution of z at t = 5. The yellow circles are the collocation points at which the equations are solved.

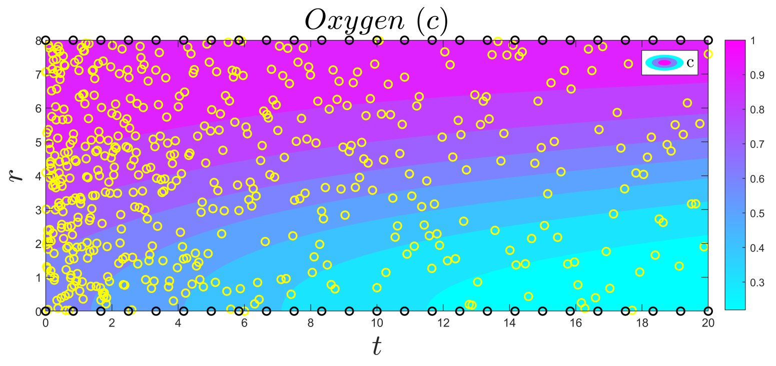


Figure 37: Contour graph of oxygen dimensionless concentration. The black circles are the boundary conditions for the dependent variable c. The yellow circles are the collocation points at which the equations are solved.

The relative error for the parameter inference after training is depicted in Table 8:

Table 8: Parameter Inference (Case 2)

Parameter	Actual Value	Inferred Value	Relative Error
d_z	0.01	0.0141	0.4107

Compared to Case 1, the relative error of the inferred d_z is lower. However, this is due to the fact that only one parameter was inferred. In Table 9 are the relative errors at t = 20 for Case 2, which are lower than Case 1:

Table 9: z and c at t = 20 for Case 2 for the entire spatial domain

Independent Variable	Relative Error
z	0.05889
c	0.02363

4.5.3 Case 3: Cancer cell and oxygen diffusion coefficient inference using cancer cell measurements

Case 3a)
Similar to case 2, only the boundary conditions were provided for oxygen, and for the cancer cells (z), the available data are the initial and boundary conditions and the distribution at t = 5. In this case, however, both d_z and d_c were inferred. The spatiotemporal evolution of cancer cells and oxygen concentrations is depicted in Figure 38

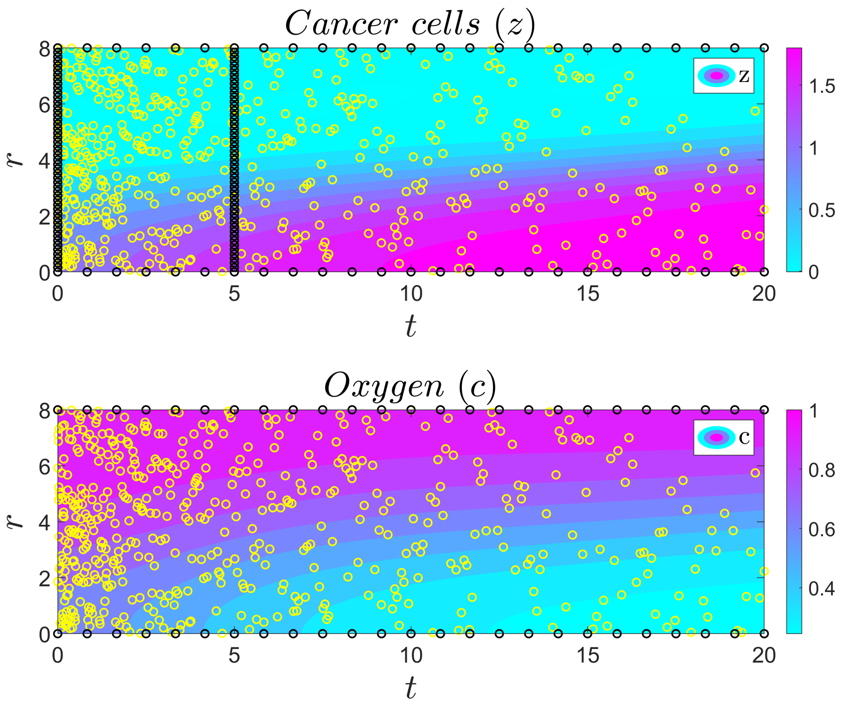


Figure 38: Contour graphs of cancer cells and oxygen dimensionless concentrations. The black circles are the initial and boundary conditions for z and the boundary conditions for c, as well as the distribution of z at t = 5. The yellow circles are the collocation points at which the equations are solved.

Table 10: Parameter Inference (Case 3a), z at t=5)

Parameter	Actual Value	Inferred Value	Relative Error
d_z	0.01	0.0199	0.99
d_c	1	1.0202	0.0202

Table 11: z and c at t = 20 for Case 3a) for the entire spatial domain

Independent Variable	Relative Error
z	0.10272
c	0.05617

In Table 10 are the relative errors for the parameter inference and in Table 11 are the relative errors for the dimensionless concentrations of cancer cells and oxygen at t=20 for the entire spatial domain. The results are similar to that of Case 1, which means that additional data for oxygen concentration might be unnecessary. Again, the area of necrosis was not predicted, and more data is needed.

Case 3b)
The distribution of the cancer cells dimensionless concentration at t=16 was added, which at a later stage of the tumor growth evolution compared to t=5. The spatiotemporal evolution of cancer cells and oxygen concentrations is depicted in Figure 39. The respective relative errors are shown Tables 12 and 13:

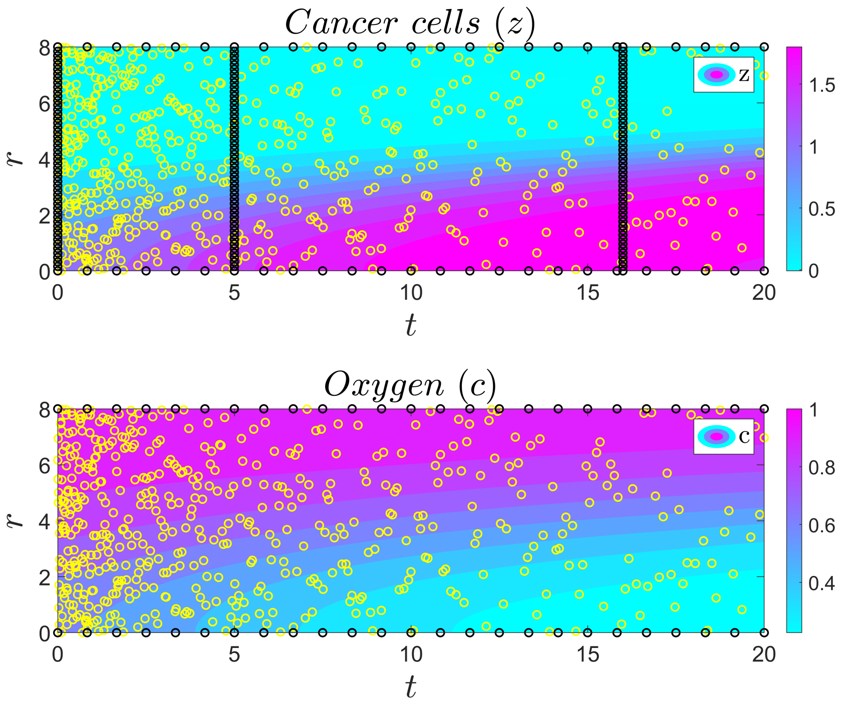


Figure 39: Contour graphs of cancer cells and oxygen dimensionless concentrations. The black circles are the initial and boundary conditions for z and the boundary conditions for c, as well as the distributions of z at t = 5,16 sec. The yellow circles are the collocation points at which the equations are solved.

Table 12: Parameter Inference (Case 3b), z at t=5,16)

Parameter	Actual Value	Inferred Value	Relative Error
d_z	0.01	0.0102	0.02
d_c	1	0.9718	0.0282

Table 13: z and c at t = 20 for Case 3b) for the entire spatial domain

Independent Variable	Relative Error
z	0.05859
c	0.02858

Case 3c)
Finally, the distribution of cancer cells at t = 10 was added as the final step of the parametric analysis, which is in the middle of the temporal domain. The spatiotemporal evolution of cancer cells and oxygen concentrations is depicted in Figure 40:

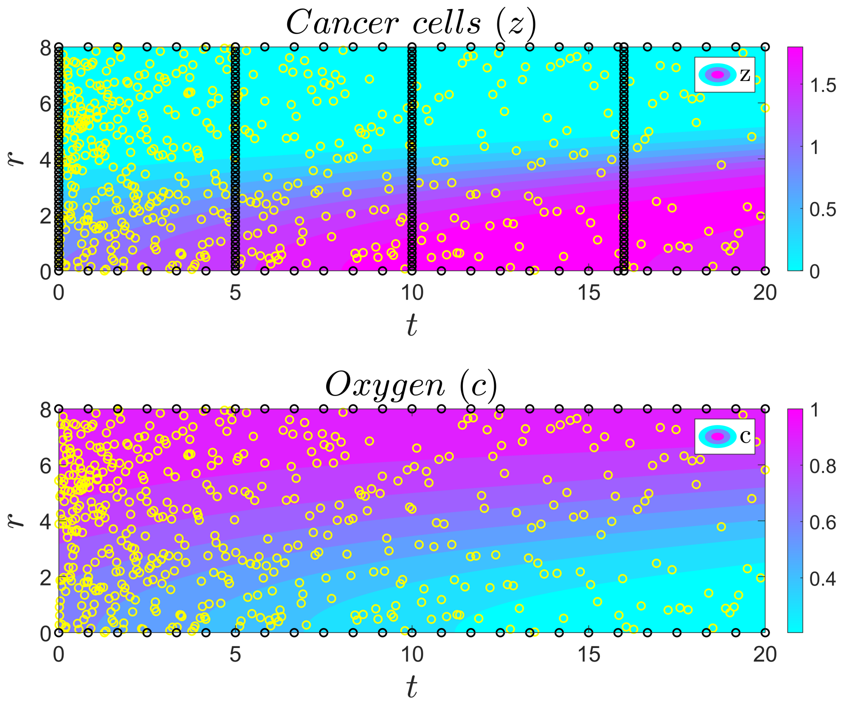


Figure 40: Contour graphs of cancer cells and oxygen. The black circles are the initial and boundary conditions for z and the boundary conditions for c, as well as the distributions of z at t = 5,10 and 16. The yellow circles are the collocation points at which the equations are solved.

and the errors are depicted in Table 14 and Table 15:

Table 14: Parameter Inference (Case 3c), z at t=5,10,16

Parameter	Actual Value	Inferred Value	Relative Error
d_z	0.01	0.0089	0.1236
d_c	1	0.9850	0.015

Table 15: z and c at t = 20 for Case 3c) for the entire spatial domain

Independent Variable	Relative Error
z	0.01542
c	0.00552

The trend of the relative error from the parametric analysis is:

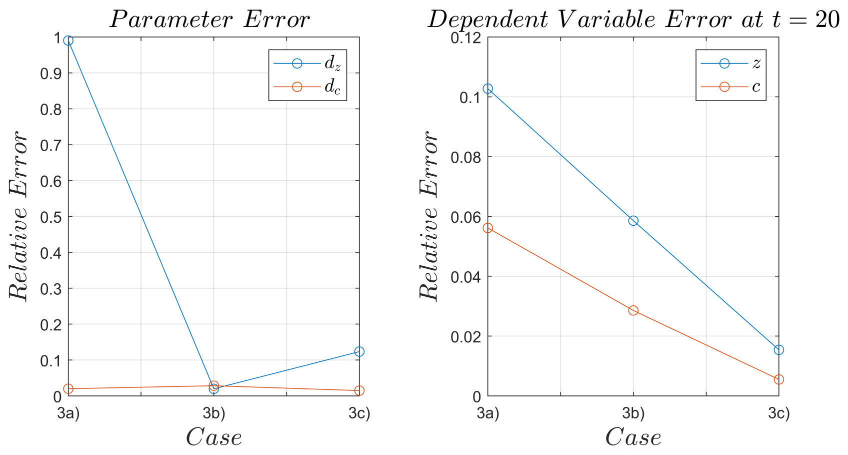


Figure 41: Parametric analysis for the relative error of Case 3

As was expected, adding more data for the cancer cells during training decreases the relative error in both dependent variables. However, in all cases, the error of d_c remained stagnant, whereas the error d_z showed a sharp decrease from Case 3a) to 3b) and a slight increase from 3b) to 3c). All of this should be considered, and one must prioritize their end goal and the available data at hand. If it is parameter inference, adding more data does not affect the error drastically. On the other hand, with a closer look at the contour graphs, the death of cancer cells is more accurately depicted in Case 3c) and overlooked in Case 3a), and this should be considered if one wants to accurately depict the solution in the spatio-temporal domain.

4.5.4 Case 4: Parameter inference with noisy data

Finally, for the final case, noisy data was introduced for case 3a). Usually, in real-case scenarios, experimental data tends to be noisy rather than smooth. More specifically, a noise ±5% of the initial conditions and the distribution at t = 5 of cancer cells z were inserted into the training data of case 3a) (shown in Figure 42).

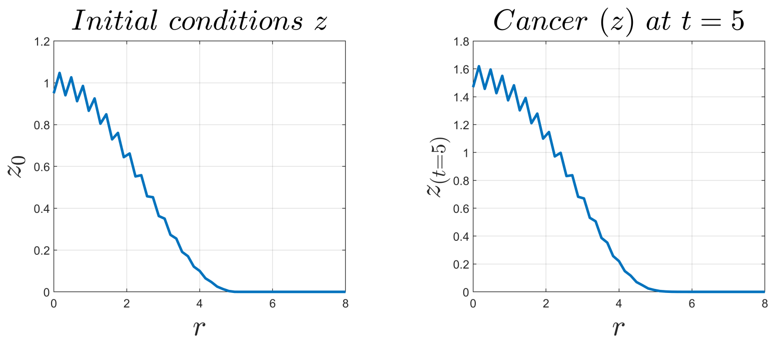


Figure 42: Noisy data of Case 4

The results are provided in Table 16 and 17:

Table 16: Parameter Inference (Case 4), z at t=5)

Parameter	Actual Value	Inferred Value	Relative Error
d_z	0.01	0.0339	2.3899
d_c	1	1.1956	0.1956

Table 17: z and c at t = 20 for Case 4 for the entire spatial domain

Independent Variable	Relative Error
z	0.2227
c	0.09106

Even though the results do seem to provide a good estimate of the solution, the error for both dependent variables at t=20 and for the parameters d_z and d_c are higher than in the previous cases. This could be fixed by prioritizing the most important end goal: the correct predicted solution or a better estimate of parameters with the available data.

5 Conclusions and Future Considerations

In this thesis, physics-informed neural networks (PINNs) were utilized to address a system of partial differential equations (PDEs) that simulate the growth of an avascular tumor. Specifically, the focus was on modeling the growth of a cancerous tumor from a nutrient source, oxygen. PINNs, characterized by their ability to function effectively with minimal data, employed automatic differentiation to compute spatial and temporal gradients inherent in the equations, thereby facilitating the solution process.

The study comprised two main components. Initially, a neural network was employed to tackle the system of partial differential equations (PDEs), where both initial and boundary conditions were provided as input data for the two dependent variables: cancer cells and oxygen. Additionally, an extrapolation case study was conducted. The second segment concentrated on parameter inference, incorporating various case studies with diverse sets of available data. To enhance accuracy, a novel approach involving dynamic weights was implemented. This method addresses the imbalance of gradients within the loss function terms by adjusting weights for each loss term during training. This facilitated the adoption of a relatively straightforward feedforward method consisting of 10 layers, each comprising 20 neurons and employing 500 logarithmically spaced internal collocation points within the spatiotemporal domain.

The evaluation of the system of PDEs involved analyzing the relative error across three time steps: t=0, 10, and 20. For cancer cells, the relative error ranged from 0.00197 to 0.01725, while for oxygen, it ranged from 0.0001 to 0.00326. Notably, at t=20, the depiction of cellular death was precise. Furthermore, an extrapolation to t=30 was conducted, revealing relative errors of 0.07021 for cancer cells and 0.01694 for oxygen, which were deemed satisfactory. However, when compared to the numerical results provided by COMSOL, the prediction accuracy for the area of cellular death was not optimal.

Parameter inference involved examining various cases, some solely focusing on the diffusion coefficient (d_z) of cancer cells, while others considered both d_z and d_c (the diffusion coefficient of oxygen), with different datasets provided for each case. The third case, which embraced a more realistic approach, solely provided distributions for cancer cells at different time steps, while for oxygen, only boundary conditions were available. The most promising results in the cases of inference were obtained in a scenario where three different distributions for cancer cells were given (at t=5, 10, and 16), yielding relative errors of 0.1236 for d_z and 0.015 for d_c. In this case, the relative error for the cancer cells was 0.01542, and for the oxygen, 0.00552 at $t = 20$ for the entire spatial domain. Additionally, it was noted that augmenting the training data led to a decrease in the relative error of the solution, as anticipated, albeit without guaranteeing more accurate parameter inference. Furthermore, a case involving noisy input data was examined to demonstrate the neural network's capability to deliver satisfactory results even when the provided data is not smooth.

The contribution presented in this thesis lays a solid groundwork for future advancements. For instance, extending the problem to higher dimensions poses a challenge for numerical methods due to the increased computational costs associated with the curse of dimensionality. However, this relatively straightforward problem addressed in the thesis can serve as a pre-trained network for more complex problems, such as those in 2D and 3D, providing a valuable starting point for training by initializing the weights and biases of the neural network. Moreover, exploring the implementation of the problem in more intricate geometries holds promise. The meshless nature of the PINN method circumvents issues related to grid generation in spatial domains with complex configurations.

Furthermore, parameter inference in the context of cancerous tumor growth is of paramount importance, considering that many parameters, such as diffusion coefficients, are patient-specific and vary from individual to individual. Consequently, this study's findings underscore the effectiveness of employing neural networks for parameter inference, yielding highly satisfactory results.

In conclusion, it is essential to recognize that artificial intelligence in scientific computing is not intended to supplant classical numerical methods, which have been established over many years. Rather, it serves as a complementary tool, augmenting these methods to tackle challenging physical phenomena encountered in engineering applications.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN: 9780134610993. URL: http://aima.cs.berkeley.edu/.

[2] Thomas W. Edgar and David O. Manz. "Chapter 6 - Machine Learning". In: *Research Methods for Cyber Security*. Ed. by Thomas W. Edgar and David O. Manz. Syngress, 2017, pp. 153–173. ISBN: 978-0-12-805349-2. DOI: https://doi.org/10.1016/B978-0-12-805349-2.00006-6. URL: https://www.sciencedirect.com/science/article/pii/B9780128053492000066.

[3] Annina Simon et al. "An Overview of Machine Learning and its Applications". In: *International Journal of Electrical Sciences Engineering* Volume (Jan. 2016), pp. 22–24.

[4] Gangadhar Shobha and Shanta Rangaswamy. "Chapter 8 - Machine Learning". In: *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications*. Ed. by Venkat N. Gudivada and C.R. Rao. Vol. 38. Handbook of Statistics. Elsevier, 2018, pp. 197–228. DOI: https://doi.org/10.1016/bs.host.2018.07.004. URL: https://www.sciencedirect.com/science/article/pii/S0169716118300191.

[5] Khadija El Bouchefry and Rafael S. de Souza. "Chapter 12 - Learning in Big Data: Introduction to Machine Learning". In: *Knowledge Discovery in Big Data from Astronomy and Earth Observation*. Ed. by Petr Škoda and Fathalrahman Adam. Elsevier, 2020, pp. 225–249. ISBN: 978-0-12-819154-5. DOI: https://doi.org/10.1016/B978-0-12-819154-5.00023-0. URL: https://www.sciencedirect.com/science/article/pii/B9780128191545000230.

[6] Radu-Alexandru Burtea and Calvin Tsay. "Safe deployment of reinforcement learning using deterministic optimization over neural networks". In: *33rd European Symposium on Computer Aided Process Engineering*. Ed. by Antonios C. Kokossis, Michael C. Georgiadis, and Efstratios Pistikopoulos. Vol. 52. Computer Aided Chemical Engineering. Elsevier, 2023, pp. 1643–1648. DOI: https://doi.org/10.1016/B978-0-443-15274-0.50261-4. URL: https://www.sciencedirect.com/science/article/pii/B9780443152740502614.

[7] Yashar Hezaveh, Laurence Levasseur, and Philip Marshall. "Fast Automated Analysis of Strong Gravitational Lenses with Convolutional Neural Networks". In: *Nature* 548 (Aug. 2017). DOI: 10.1038/nature23463.

[8] Jim Gao. "Machine Learning Applications for Data Center Optimization". In: 2014. URL: https://api.semanticscholar.org/CorpusID:64625439.

[9] M.T. Hagan et al. *Neural Network Design*. Martin Hagan, 2014. ISBN: 9780971732117. URL: https://books.google.gr/books?id=4EW9oQEACAAJ.

[10] Abdullatif Baba. "Neural networks from biological to artificial and vice versa". In: *Biosystems* 235 (2024), p. 105110. ISSN: 0303-2647. DOI: https://doi.org/10.1016/j.biosystems.2023.105110. URL: https://www.sciencedirect.com/science/article/pii/S030326472300285X.

[11] *Introduction to Neural Networks*. Accessed: March 28, 2024. 2024. URL: https://www.electronicsworld.co.uk/introduction-to-neural-networks/12544/.

[12] Aston Zhang et al. *Dive into Deep Learning*. https://D2L.ai. Cambridge University Press, 2023.

[13] Gangadhar Shobha and Shanta Rangaswamy. "Chapter 8 - Machine Learning". In: *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications*. Ed. by Venkat N. Gudivada and C.R. Rao. Vol. 38. Handbook of Statistics. Elsevier, 2018, pp. 197–228. DOI: https://doi.org/10.1016/bs.host.2018.07.004. URL: https://www.sciencedirect.com/science/article/pii/S0169716118300191.

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

[15] Kuang-Hua Chang. "Chapter 4 - Structural Design Sensitivity Analysis". In: *Design Theory and Methods Using CAD/CAE*. Ed. by Kuang-Hua Chang. Boston: Academic Press, 2015, pp. 211–323. ISBN: 978-0-12-398512-5. DOI: https://doi.org/10.1016/B978-0-12-398512-5.00004-9. URL: https://www.sciencedirect.com/science/article/pii/B9780123985125000049.

[16] Md Rejwanur Rashid Mojumdar and Meskat Hossain. "Partial derivatives and jacobian matrix". In: *Encyclopedia of Electrical and Electronic Power Engineering*. Ed. by Jorge García. Oxford: Elsevier, 2023, pp. 610–613. ISBN: 978-0-12-823211-8. DOI: https://doi.org/10.1016/B978-0-12-821204-2.00083-0. URL: https://www.sciencedirect.com/science/article/pii/B9780128212042000830.

[17] Mouhacine Benosman. "Chapter 1 - Some Mathematical Tools". In: *Learning-Based Adaptive Control*. Ed. by Mouhacine Benosman. Butterworth-Heinemann, 2016, pp. 1–17. ISBN: 978-0-12-803136-0. DOI: https://doi.org/10.1016/B978-0-12-803136-0.00001-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128031360000014.

[18] Jasbir S. Arora. "11 - More on Numerical Methods for Constrained Optimum Design". In: *Introduction to Optimum Design (Second Edition)*. Ed. by Jasbir S. Arora. Second Edition. San Diego: Academic Press, 2004, pp. 379–412. ISBN: 978-0-12-064155-0. DOI: https://doi.org/10.1016/B978-012064155-0/50011-2. URL: https://www.sciencedirect.com/science/article/pii/B9780120641550500112.

[19] Ranjana Dwivedi and Vinay Kumar Srivastava. “12 - Fundamental optimization methods for machine learning”. In: *Statistical Modeling in Machine Learning*. Ed. by Tilottama Goswami and G.R. Sinha. Academic Press, 2023, pp. 227–247. ISBN: 978-0-323-91776-6. DOI: https://doi.org/10.1016/B978-0-323-91776-6.00005-1. URL: https://www.sciencedirect.com/science/article/pii/B9780323917766000051.

[20] M. M. Najafabadi, T. M. Khoshgoftaar, F. Villanustre, et al. “Large-scale distributed L-BFGS”. In: *Journal of Big Data* 4.1 (2017), p. 22. DOI: 10.1186/s40537-017-0084-5.

[21] Anders Skajaa. “Limited Memory BFGS for Nonsmooth Optimization”. Additional notes. MA thesis. New York University, Jan. 2010.

[22] N.J. Sairamya et al. “Chapter 12 - Hybrid Approach for Classification of Electroencephalographic Signals Using Time–Frequency Images With Wavelets and Texture Features”. In: *Intelligent Data Analysis for Biomedical Applications*. Ed. by D. Jude Hemanth, Deepak Gupta, and Valentina Emilia Balas. Intelligent Data-Centric Systems. Academic Press, 2019, pp. 253–273. ISBN: 978-0-12-815553-0. DOI: https://doi.org/10.1016/B978-0-12-815553-0.00013-6. URL: https://www.sciencedirect.com/science/article/pii/B9780128155530000136.

[23] Hannes Kisner, Yitao Ding, and Ulrike Thomas. “Chapter 4 - Capacitive material detection with machine learning for robotic grasping applications”. In: *Tactile Sensing, Skill Learning, and Robotic Dexterous Manipulation*. Ed. by Qiang Li et al. Academic Press, 2022, pp. 59–79. ISBN: 978-0-323-90445-2. DOI: https://doi.org/10.1016/B978-0-32-390445-2.00011-8. URL: https://www.sciencedirect.com/science/article/pii/B9780323904452000118.

[24] Yunji Chen et al. “Chapter 2 - Fundamentals of neural networks”. In: *AI Computing Systems*. Ed. by Yunji Chen et al. Morgan Kaufmann, 2024, pp. 17–51. ISBN: 978-0-323-95399-3. DOI: https://doi.org/10.1016/B978-0-32-395399-3.00008-1. URL: https://www.sciencedirect.com/science/article/pii/B9780323953993000081.

[25] Hui Liu. “Chapter 7 - Rail transit channel robot systems”. In: *Robot Systems for Rail Transit Applications*. Ed. by Hui Liu. Elsevier, 2020, pp. 283–328. ISBN: 978-0-12-822968-2. DOI: https://doi.org/10.1016/B978-0-12-822968-2.00007-3. URL: https://www.sciencedirect.com/science/article/pii/B9780128229682000073.

[26] Won-Kee Hong. “2 - Understanding artificial neural networks: analogy to the biological neuron model”. In: *Artificial Intelligence-Based Design of Reinforced Concrete Structures*. Ed. by Won-Kee Hong. Woodhead Publishing Series in Civil and Structural Engineering. Woodhead Publishing, 2023, pp. 7–13. ISBN: 978-0-443-15252-8. DOI: https://doi.org/10.1016/B978-0-443-15252-8.00003-0. URL: https://www.sciencedirect.com/science/article/pii/B9780443152528000030.

[27] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *ICML 2010*. 2010, pp. 807–814.

[28] K. Balaji and K. Lavanya. "Chapter 5 - Medical Image Analysis With Deep Neural Networks". In: *Deep Learning and Parallel Computing Environment for Bioengineering Systems*. Ed. by Arun Kumar Sangaiah. Academic Press, 2019, pp. 75–97. ISBN: 978-0-12-816718-2. DOI: https://doi.org/10.1016/B978-0-12-816718-2.00012-9. URL: https://www.sciencedirect.com/science/article/pii/B9780128167182000129.

[29] Stephanie Kay Ashenden et al. "Chapter 2 - Introduction to artificial intelligence and machine learning". In: *The Era of Artificial Intelligence, Machine Learning, and Data Science in the Pharmaceutical Industry*. Ed. by Stephanie Kay Ashenden. Academic Press, 2021, pp. 15–26. ISBN: 978-0-12-820045-2. DOI: https://doi.org/10.1016/B978-0-12-820045-2.00003-9. URL: https://www.sciencedirect.com/science/article/pii/B9780128200452000039.

[30] Sarvesh PS Rajput. "6 - Applying artificial intelligence to predict green concrete compressive strength". In: *Artificial Intelligence for Renewable Energy Systems*. Ed. by Ashutosh Kumar Dubey et al. Woodhead Publishing Series in Energy. Woodhead Publishing, 2022, pp. 131–149. ISBN: 978-0-323-90396-7. DOI: https://doi.org/10.1016/B978-0-323-90396-7.00003-1. URL: https://www.sciencedirect.com/science/article/pii/B9780323903967000031.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[32] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. "Human-level concept learning through probabilistic program induction". In: *Science* 350.6266 (2015), pp. 1332–1338. DOI: 10.1126/science.aab3050. eprint: https://www.science.org/doi/pdf/10.1126/science.aab3050. URL: https://www.science.org/doi/abs/10.1126/science.aab3050.

[33] Babak Alipanahi et al. "Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning". In: *Nature biotechnology* 33.8 (2015), pp. 831–838.

[34] Shuang Hu et al. "Physics-informed neural network combined with characteristic-based split for solving forward and inverse problems involving Navier–Stokes equations". In: *Neurocomputing* 573 (2024), p. 127240. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2024.127240. URL: https://www.sciencedirect.com/science/article/pii/S0925231224000110.

[35] Meng-Juan Xiao et al. "Physics-informed neural networks for the Reynolds-Averaged Navier–Stokes modeling of Rayleigh–Taylor turbulent mixing". In: *Computers Fluids* 266 (2023), p. 106025. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2023.106025. URL: https://www.sciencedirect.com/science/article/pii/S0045793023002505.

[36] S. Hanrahan, M. Kozul, and R.D. Sandberg. "Studying turbulent flows with physics-informed neural networks and sparse data". In: *International Journal of Heat and Fluid Flow* 104 (2023), p. 109232. ISSN: 0142-727X. DOI: https://doi.org/10.1016/j.ijheatfluidflow.2023.109232. URL: https://www.sciencedirect.com/science/article/pii/S0142727X23001315.

[37] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), 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.

[38] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767 [cs.SC].

[39] MathWorks. *Solve Partial Differential Equations with L-BFGS Method and Deep Learning*. https://www.mathworks.com/help/deeplearning/ug/solve-partial-differential-equations-with-lbfgs-method-and-deep-learning.html. Accessed: 2024-06-25. 2024.

[40] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. "Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations". In: *Science* 367.6481 (2020), pp. 1026–1030. DOI: 10.1126/science.aaw4741. eprint: https://www.science.org/doi/pdf/10.1126/science.aaw4741. URL: https://www.science.org/doi/abs/10.1126/science.aaw4741.

[41] Sifan Wang and Paris Perdikaris. "Deep learning of free boundary and Stefan problems". In: *Journal of Computational Physics* 428 (Mar. 2021), p. 109914. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.109914. URL: http://dx.doi.org/10.1016/j.jcp.2020.109914.

[42] George Em Karniadakis et al. "Physics-informed machine learning". In: *Nature Reviews Physics* 3.6 (May 2021). ISSN: 2522-5820. DOI: 10.1038/s42254-021-00314-5. URL: https://www.osti.gov/biblio/1852843.

[43] Liu Yang, Xuhui Meng, and George Em Karniadakis. "B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data". In: *Journal of Computational Physics* 425 (Jan. 2021), p. 109913. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.109913. URL: http://dx.doi.org/10.1016/j.jcp.2020.109913.

[44] Erich Novak and Klaus Ritter. “The Curse of Dimension and a Universal Method For Numerical Integration”. In: *Multivariate Approximation and Splines*. Ed. by Günther Nürnberger, Jochen W. Schmidt, and Guido Walz. Basel: Birkhäuser Basel, 1997, pp. 177–187. ISBN: 978-3-0348-8871-4.

[45] Tomaso Poggio et al. *Why and When Can Deep – but Not Shallow – Networks Avoid the Curse of Dimensionality: a Review*. 2017. arXiv: `1611.00740 [cs.LG]`.

[46] Philipp Grohs et al. “A Proof that Artificial Neural Networks Overcome the Curse of Dimensionality in the Numerical Approximation of Black–Scholes Partial Differential Equations”. In: *Memoirs of the American Mathematical Society* 284.1410 (Apr. 2023). ISSN: 1947-6221. DOI: `10.1090/memo/1410`. URL: `http://dx.doi.org/10.1090/memo/1410`.

[47] Michael Penwarden et al. “A metalearning approach for Physics-Informed Neural Networks (PINNs): Application to parameterized PDEs”. In: *Journal of Computational Physics* 477 (2023), p. 111912. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2023.111912`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999123000074`.

[48] Félix Fernández de la Mata et al. “Physics-informed neural networks for data-driven simulation: Advantages, limitations, and opportunities”. In: *Physica A: Statistical Mechanics and its Applications* 610 (2023), p. 128415. ISSN: 0378-4371. DOI: `https://doi.org/10.1016/j.physa.2022.128415`. URL: `https://www.sciencedirect.com/science/article/pii/S0378437122009736`.

[49] Jiawei Guo et al. “Pre-training strategy for solving evolution equations based on physics-informed neural networks”. In: *Journal of Computational Physics* 489 (2023), p. 112258. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2023.112258`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999123003534`.

[50] I. Lampropoulos, M. Charoupa, and M. Kavousanakis. “Intra-tumor heterogeneity and its impact on cytotoxic therapy in a two-dimensional vascular tumor growth model”. In: *Chemical Engineering Science* 259 (2022), p. 117792. ISSN: 0009-2509. DOI: `https://doi.org/10.1016/j.ces.2022.117792`. URL: `https://www.sciencedirect.com/science/article/pii/S0009250922003761`.

[51] I. Lampropoulos and M. Kavousanakis. “Application of combination chemotherapy in two dimensional tumor growth model with heterogeneous vasculature”. In: *Chemical Engineering Science* 280 (2023). Cited by: 0. DOI: `10.1016/j.ces.2023.118965`. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161651040&doi=10.1016%2fj.ces.2023.118965&partnerID=40&md5=4db90a5bd0e5cfa42acd9bc3f2083d37`.

[52] Shirong Li and Xinlong Feng. “Dynamic Weight Strategy of Physics-Informed Neural Networks for the 2D Navier–Stokes Equations”. In: *Entropy* 24.9 (2022). ISSN: 1099-4300. DOI: `10.3390/e24091254`. URL: `https://www.mdpi.com/1099-4300/24/9/1254`.