UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS - FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL

Yan Barbosa Werneck

Comparing Classical Ordinary Differential Equation and Neural Network Models for Reduced-Order Single-Cell Electrophysiology

Juiz de Fora
2024

Yan Barbosa Werneck

Comparing Classical Ordinary Differential Equation and Neural Network Models for Reduced-Order Single-Cell Electrophysiology

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Mestre em Modelagem Computacional. Área de concentração: Modelagem Computacional.

Orientador: Prof. Dr. Rodrigo Weber dos Santos

Coorientador: Prof. Dr. Bernardo Martins Rocha

Juiz de Fora

2024

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Werneck, Yan Barbosa.

Comparing Classical Ordinary Differential Equation and Neural Network Models for Reduced-Order Single-Cell Electrophysiology / Yan Barbosa Werneck. – 2024.

89 f. : il.

Orientador: Rodrigo Weber dos Santos

Coorientador: Bernardo Martins Rocha

Dissertação (Mestrado) – Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas - Faculdade de Engenharia. Programa de Pós-Graduação em Modelagem Computacional, 2024.

1. Neural Networks. 2. ODEs. 3. Action Potential. 4. PINNs. I.I.Weber dos Santos, Rodrigo, orient. II. Rocha, Bernardo Martins, coorient.

Yan Barbosa Werneck

Comparing Classical Ordinary Differential Equation and Neural Network Models for Reduced-Order Single-Cell Electrophysiology

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Mestre em Modelagem Computacional. Área de concentração: Modelagem Computacional.

Aprovada em 13 de novembro de 2024.

BANCA EXAMINADORA

Prof. Dr. Rodrigo Weber dos Santos - Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. Bernardo Martins Rocha - Coorientador
Universidade Federal de Juiz de Fora

Prof. Dr. Marcelo Lobosco
Universidade Federal de Juiz de Fora

Prof.ª Dr.ª Elizabeth Maura Cherry
Georgia Institute of Technology, Estados Unidos

Juiz de Fora, 05/11/2024.

Documento assinado eletronicamente por **Rodrigo Weber dos Santos**, **Professor(a)**, em 13/11/2024, às 12:55, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020]().

Documento assinado eletronicamente por **Marcelo Lobosco**, **Professor(a)**, em 14/11/2024, às 14:52, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020]().

Documento assinado eletronicamente por **Bernardo Martins Rocha**, **Professor(a)**, em 14/11/2024, às 16:43, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020]().

Documento assinado eletronicamente por **Elizabeth Maura Cherry**, **Usuário Externo**, em 14/11/2024, às 19:42, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do [Decreto nº 10.543, de 13 de novembro de 2020]().

A autenticidade deste documento pode ser conferida no Portal do SEI-Ufjf (www2.ufjf.br/SEI) através do ícone Conferência de Documentos, informando o código verificador **2080407** e o código CRC **0CA171E7**.

AGRADECIMENTOS

I am deeply grateful to those who supported this work and made it possible. My heartfelt thanks go to my advisors, Prof. Rodrigo Weber and Prof. Bernardo Rocha, whose guidance and insights have been invaluable throughout the research process and to my growth as an academic professional.

I would like to extend my gratitude to all involved in the Graduate Program in Computational Modeling (PGMC), whose hard work and collaboration have enriched my academic experience.

As special thank you to my parents, Cíntia and Denilson, for their unwavering love and support, and for inspiring me every step of the way.

Finally, I would like to acknowledge the financial support from CNPq, CAPES, and FAPEMIG through the grants APQ-02445/24 and APQ-02489-21, which made this research possible.

No computer has ever been designed that is ever aware of what it’s doing; but most of the time, we aren’t either.

Marvin Minsky

ABSTRACT

Modeling cardiac electrophysiology plays a crucial role in advancing non-invasive diagnostics and enhancing our understanding of heart function. Historically, models describing excitable cells through systems of Ordinary Differential Equations (ODEs) have been the standard in electrophysiology modeling. These models range from detailed representations of ion channel dynamics to simplified reduced-order models that capture the behavior of excitability phenomenologically. In this work, we compare a fast reduced-order model with data-driven and physics-informed neural networks to assess their effectiveness as efficient replacements for numerical solutions. For this, the FitzHugh-Nagumo model was used, and scenarios with increasing complexity were studied. The networks were trained using numerical data and knowledge of model physics, derived from the ODEs. Additionally, several techniques were employed to improve training, including architecture optimization, increased point density in regions of high error, and time-domain splitting. Inference was conducted using the state-of-the-art TensorRT SDK to speed up model inference, leveraging tensor core matrix-matrix specialization to ensure maximum performance. We observed up to a 1.8x speedup compered to numerical models optimized and implemented in CUDA, with minimal loss in accuracy. These gains highlight valuable use cases for neural network emulators, as faster substitute for numerical methods when complexity can be controlled, while still emphasizing the prominence of equation-based modeling in cardiac electrophysiology in general due to their flexibility.

Contents

1 INTRODUCTION

1.1 Motivation

Cardiac diseases are the leading cause of mortality worldwide, responsible for tens of millions of deaths annually (World Health Organization, 2024). Early detection through preventive exams plays a crucial role in mitigating the impact of these diseases by identifying potential risks before they manifest into severe conditions. However, traditional diagnostic procedures, while effective, are often invasive, carrying associated risks and discomfort for patients. As an alternative, advancements in computational modeling and the development of digital twins offer promising solutions (COOREY GLEN, 2022), (SEL K., 2024) ,(LEPPER; BUCK, 2022). For instance, the Virtual Electrophysiological Study can assess arrhythmia risk without the need for invasive procedures, such as catheter insertion and continuous heart stimulation, required in a conventional Electrophysiological Study. These digital exams provide a non-invasive, risk-free approach to monitoring and predicting cardiac health, potentially revolutionizing the way we approach cardiac disease prevention.

These virtual studies rely on large-scale models of the heart, typically represented by meshes comprising millions of volumetric elements. These models are generally based on partial differential equations (PDEs) and ordinary differential equations (ODEs), which are solved numerically through computationally intensive methods. Furthermore, due to uncertainties in parameter estimation and model analysis (CAMPOS et al., 2020), multiple simulations are often required, further increasing computational costs. To address these challenges, this work explores the use of neural networks to accelerate the computational processes.

The use of neural networks to solve differential equations is an emerging field that benefits a lot from recent advancements in other domains employing neural networks. Techniques such as Physics-Informed Neural Networks (PINNs) (RAISSI; PERDIKARIS; KARNIADAKIS, 2017b) are gaining traction, offering more efficient training methods for these models. Additionally, the development of specialized hardware designed to accelerate neural network inference (QIAN et al., 2022) is further enhancing the feasibility of this approach.

1.2 Objectives

The objective of this work is to investigate how these technological advancements can enable the application of neural networks to accelerate models like those used in the Virtual Electrophysiological Study, thereby making them more efficient and accessible for cardiac disease prevention. In particular we have the following objectives:

O1: Explore the application of artificial neural networks to emulate models such as the FitzHugh-Nagumo (FHN) model used in cardiac electrophysiology simulations.

O2: Investigate the use of techniques like Physics-Informed Neural Networks to enhance the training of these models.

O3: Develop neural network emulators leveraging recent advancements in neural network specialized hardware to be efficient alternatives to numerical solution.

1.3 Literature Review

The use of machine learning techniques to solve systems of differential equations has been gaining increasing attention in recent years (MENG et al., 2022). This approach seeks to leverage the recent developments in machine learning techniques and hardware to find solutions to complex differential equations, which traditionally require intensive numerical methods, in a more efficient manner. In this context, two primary approaches stand out: purely data-driven models and techniques that integrate prior knowledge of the system into the learning process of the models. Of special interest are Physics-Informed Machine Learning (PIML) models, trained by incorporating the known system equations.

Pure data-driven models models—spanning from basic regression techniques to more sophisticated algorithms—are trained on large datasets, that may consist of data synthetically generated from computational simulations or from real-world observation. These models learn the relationships between input and output variables solely based on the provided data. This approach has been particularly successful in areas with abundant and reliable empirical data, where advanced machine learning techniques, such as linearly recurrent autoencoder networks (OTTO; ROWLEY, 2019), have proven effective in modeling complex nonlinear dynamics. Additionally, data-driven approaches have demonstrated the capability to recover original differential equations from noisy and incomplete datasets, offering a powerful tool for model discovery and calibration (GLASNER, 2019; MASLYAEVA; HVATOVA, 2019). Specifically in the field of electrophysiology, data-driven neural networks have been successfully used to speed up action potential simulations (GRANDITS et al., 2021).

However, purely data-driven models have significant limitations. The accuracy and generalization capabilities of these models heavily depend on the quality and quantity of the training data. When dealing with observational data, such extensive datasets often do not exist or include significant noise, making it difficult to achieve accurate and generalizable models, often requiring stochastic approaches (RAISSI; PERDIKARIS; KARNIADAKIS, 2017a). In the case of synthetic data, producing these large datasets with numerical models can be prohibitively expensive and computationally intensive. Additionally, in fields where the underlying processes are governed by well-established physical laws, the exclusion of this knowledge from the learning process can result in predictions that are inconsistent with physical realities.

To address these challenges, Physics-Informed Machine Learning has emerged as a promising approach that integrates prior physical knowledge into machine learning models to enhance training. By incorporating physical laws and constraints directly into the learning process, PIML enhances the accuracy and generalization capabilities of models, even when training data is scarce or noisy. In Wu, Sicard e Gadsden (2024) and Meng et al. (2022) a comprehensive review of the applications of PIML is provided, highlighting its effectiveness in areas such as anomaly detection, condition monitoring, and solving complex physical problems, while also discussing some limitations.

A particularly relevant example of Physics-Informed Machine Learning are the Physics-Informed Neural Networks. PINNs extend the concept of integrating physical laws into machine learning by embedding these laws directly into the training loss functions of neural networks. This approach was notably advanced in the work of Raissi, Perdikaris e Karniadakis (2017b), who demonstrated that PINNs could effectively solve partial differential equations (PDEs) with limited data. Since then, PINNs have been successfully applied in diverse fields such as fluid dynamics (RAISSI; PERDIKARIS; KARNIADAKIS, 2019), structural analysis (HAGHIGHAT et al., 2021), and electromagnetic simulations (CHEN et al., 2020), where they offer a powerful alternative to traditional expensive numerical solutions. By incorporating the governing equations as well as known boundary and initial conditions (WANG; YU; PERDIKARIS, 2021), PINNs enable model training even with scarce data.

PINNs are particularly promising in the field of Computational Electrophysiology, where data is often scarce but the underlying system dynamics are well-understood and thoroughly studied (PLANK; MESAROVIC; TRAYANOVA, 2020; PASSERINI, 2020). In this context, PINNs offer a practical solution by integrating prior knowledge of the governing equations, such as those describing cardiac and neuronal electrical activities, directly into the neural network's training process. This capability allows PINNs to efficiently leverage existing physiological models and experimental data to enhance predictive accuracy and reduce the reliance on large datasets. For example, PINNs have been successfully used to characterize cardiac electrophysiology, providing valuable insights into the electrical behavior of the heart and aiding in the study of arrhythmias and other cardiac conditions (SAHLI; RAISSI; KARNIADAKIS, 2020). Furthermore, recent advances in hardware, such as the advent of Tensor Cores, have allowed even faster inference for neural networks (WANG et al., 2021), further highlighting their potential to complement traditional numerical solvers in those extensive scenarios.

2 BASIC CONCEPTS

2.1 Action Potential In Excitable Cells

The action potential is a fundamental phenomenon that occurs in excitable cells such as neurons, muscle fibers, and certain glandular cells. Action potential refers to the rapid and transient change in the membrane potential of a cell in response to an electrical stimulus, allowing cells to transmit signals or initiate specific processes like muscle contraction or hormone secretion.

In its resting state, an excitable cell maintains a stable negative membrane potential, usually between -60 to -90 mV, due to the unequal distribution of ions across the cell membrane. This potential is largely maintained by the sodium-potassium pump (Na^+/K^+ ATPase), which actively transports sodium (Na^+) ions out of the cell and potassium (K^+) ions into the cell. This creates steep concentration gradients of the ionic species between intracellular and extracellular mediums, causing the negative resting potential.

When a stimulus reaches a critical threshold, voltage-gated sodium channels open, leading to a rapid influx of Na^+ ions into the cell. This sudden influx of positive ions causes a sharp depolarization, where the membrane potential becomes positive, typically reaching around $+30$ mV. This depolarization marks the initiation of the action potential. Once the peak is reached, sodium channels close and voltage-gated potassium channels open, allowing K^+ to flow out of the cell. This outward flow of potassium ions returns the membrane potential back to its negative resting state, a process called repolarization. In some cases, the membrane potential may briefly undershoot, becoming more negative than the resting level, a phase called hyperpolarization. This process is shown in Figure 1.

The action potential is an all-or-nothing event, meaning that once the threshold is reached, the full depolarization-repolarization cycle occurs in a predictable, consistent manner. If the stimulus does not reach this threshold, no action potential is triggered. After firing, cells enter a refractory period during which their excitability is significantly reduced. In the absolute refractory period, sodium channels are inactivated, preventing any new action potential, while during the relative refractory period, a stronger stimulus is required to trigger another action potential. Repetitive firing occurs when a sustained stimulus induces multiple action potentials, with the frequency of these spikes encoding information about stimulus intensity.

This excitable dynamics are crucial to allow for the correct function of electrophysiological systems where cells are connected with each other along tissues, and may stimulate each other. In the brain, cells are arranged in a network topology, and process information of input cells to dictate the stimulus given to cells down the line, and the activation or not of each cell along the network is associated with many brain process. Here the all-or-nothing behavior is crucial, being the core component of the information

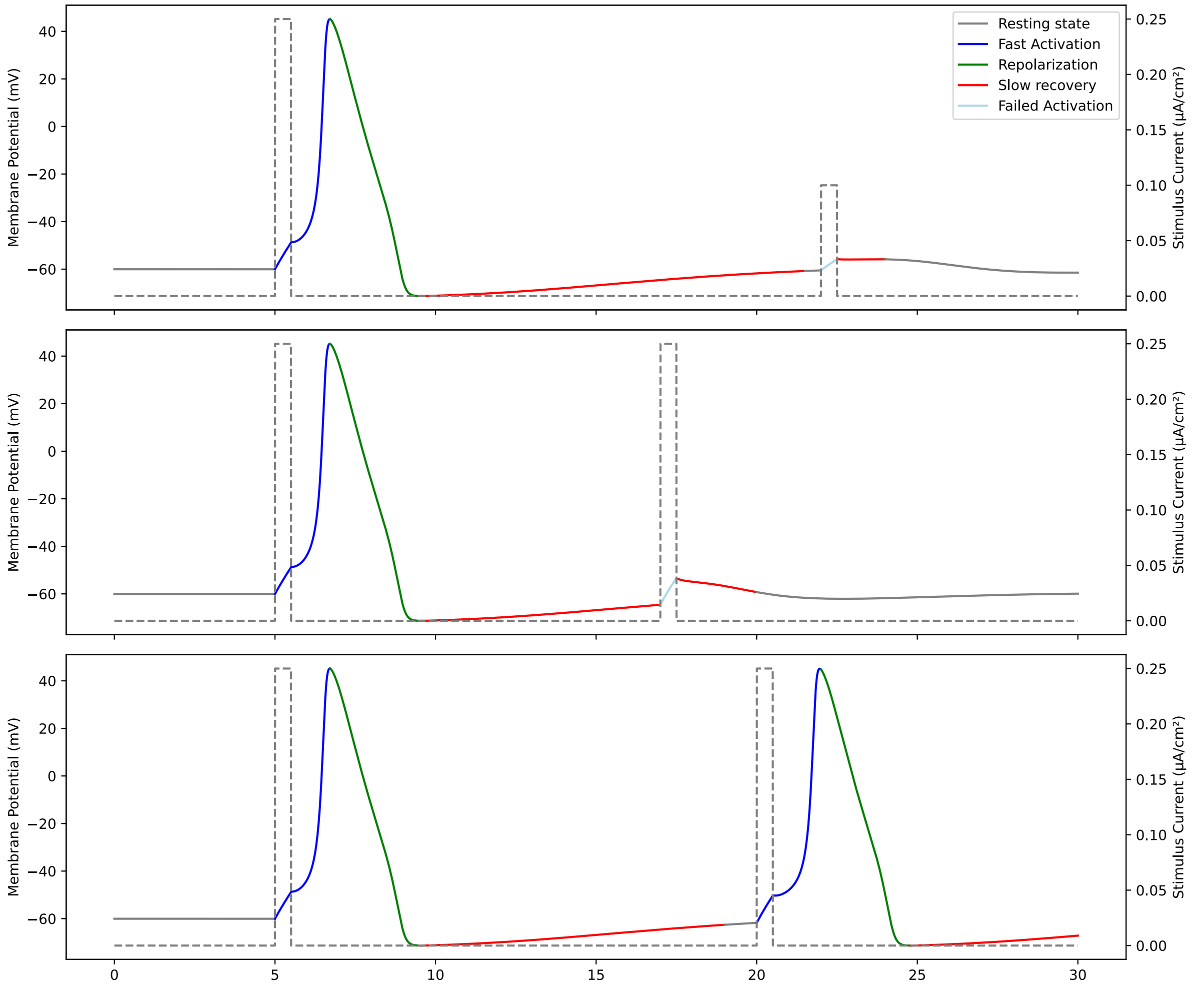

Figure 1 – Representation of an action potential and its principal characteristics. The top figure shows a large stimulus causing an action potential (suprathreshold) followed by a smaller stimulus that doesn't cause an AP (subthreshold). The middle figure illustrates a second stimulus applied while the cell is still in the recovery phase from the first stimulus, preventing the generation of another AP. The bottom figure demonstrates a second stimulus applied after the first one has completely ended, resulting in another action potential. Each phase of the action potential is shown separately, highlighting the different timescales.

processing, aggregating the stimulus received from all connected cells in a 0 or 1 response. In the heart, the action potential causes cells to contract, and the correctly synchronized propagation of an action potential wave along the heart tissue triggers a heartbeat. Here the refractory period is crucial for forcing the propagation to be unidirectional.

Action potentials form the electrical basis of key physiological processes, playing a central role in communication within the nervous system, regulating cardiac rhythms, and driving muscle activity. Thus, the precise coordination of ion channels and their timings is essential for the function of excitable cells, and any disturbance in this balance can lead to abnormal cellular behavior or pathophysiological conditions, such as inducing arrhythmias in the heart or impairing neural signaling.

2.2 Hodgkin-Huxley Model

The study of action potentials in excitable cells can be greatly enhanced by employing detailed biophysical models that capture the underlying ionic mechanisms, facilitating the investigation of the generation and propagation of action potentials. One of the most celebrated Action Potential models in electrophysiology is the Hodgkin-Huxley (HH) model (HODGKIN; HUXLEY, 1952), developed by Alan Hodgkin and Andrew Huxley in 1952. This model was formulated based on experimental data from the squid giant axon and revolutionized our understanding of neuronal activity by providing a quantitative description of the ionic currents that contribute to action potentials.

The Hodgkin-Huxley model is grounded in the biophysical principles of membrane conductance and ion channel dynamics. It describes the behavior of the membrane potential V by accounting for the flow of three main ionic currents: the sodium current (I_{Na}), the potassium current (I_K), and a leakage current (I_L). These currents are modeled as functions of voltage-dependent conductances and driving forces, as shown in the following set of coupled differential equations:

$$C_m \frac{dV}{dt} = -(I_{Na} + I_K + I_L) + I_{ext}, \tag{2.1}$$

$$I_{Na} = g_{Na} m^3 h (V - E_{Na}), \tag{2.2}$$

$$I_K = g_K n^4 (V - E_K), \tag{2.3}$$

$$I_L = g_L (V - E_L), \tag{2.4}$$

where C_m represents the membrane capacitance, and I_{ext} denotes an external stimulus current. The variables m, h, and n are gating variables that govern the opening and closing of ion channels, with their dynamics described by first-order differential equations. g_{Na}, g_K, and g_L represent the maximum conductances of the sodium, potassium, and leakage channels, respectively, while E_{Na}, E_K, and E_L are the corresponding reversal potentials for these ions.

The gating variables (m, h, and n) follow sigmoidal activation and inactivation kinetics, depending on the membrane voltage. Their time evolution is given by equations of the form:

$$\frac{dx}{dt} = \alpha_x(V)(1 - x) - \beta_x(V)x, \tag{2.5}$$

where x represents m, h, or n, and α_x and β_x are voltage-dependent rate constants that determine the transition probabilities between open and closed states of the ion channels.

The Hodgkin-Huxley model captures the full action potential waveform, including the rapid depolarization phase, the overshoot, and the subsequent repolarization and hyperpolarization phases. It is able to describe all the key dynamics of excitability: threshold behavior, refractory periods, and repetitive firing, making it a valuable tool for understanding excitable tissue behavior.

The strength of the Hodgkin-Huxley model lies in its biophysical detail and its ability to provide a mechanistic understanding of action potentials. By specifying the ion conductances and reversal potentials, the model can be adapted to various types of excitable cells. For example, modifications to the channel kinetics and conductances allow the model to be applied to neurons with different firing patterns or to cardiac cells with distinct action potential morphologies, or even to test the effect of drugs that influence the ionic channels. The model's utility in both theoretical and experimental contexts has led to its widespread adoption in neuroscience and cardiac electrophysiology. Figure 2 illustrates the action potential generated by the Hodgkin-Huxley (HH) model under a typical external stimulus, with the separate contributions of the sodium, potassium, and leakage currents depicted.

The Hodgkin-Huxley model is a classical example of a mechanistic model in biological systems. Comprising four coupled ordinary differential equations, it provides a detailed and accurate description of the ionic currents and gating dynamics underlying action potentials.

2.3 FitzHugh-Nagumo: A Surrogate Mathematical Model

While the Hodgkin-Huxley model provides a detailed and mechanistic description of the ionic currents underlying action potentials, its complexity—arising from four coupled nonlinear ordinary differential equations—can make it computationally intensive, particularly when modeling large networks of neurons or cardiac cells. To address this challenge, simplified models like the FitzHugh-Nagumo (FHN) model were developed.

The model developed by Richard FitzHugh and J. Nagumo in the 1960s, serves as a mathematical surrogate, capturing the essential dynamics of excitability and action potentials with reduced complexity. The FHN model reduces the detailed ionic currents and gating variables from the HH model into a simplified framework, focusing on the key features of excitability and action potential generation. The original HH system, described by the membrane potential V and three gating variables m, h, and n, is condensed into a two-dimensional system by retaining the membrane potential V and introducing a recovery variable W. This simplification reduces the dimensionality of the problem, making the system easier to solve numerically and analyze analytically, although it still lacks a complete analytical solution.

The resulting FHN model captures the qualitative dynamics of action potentials,

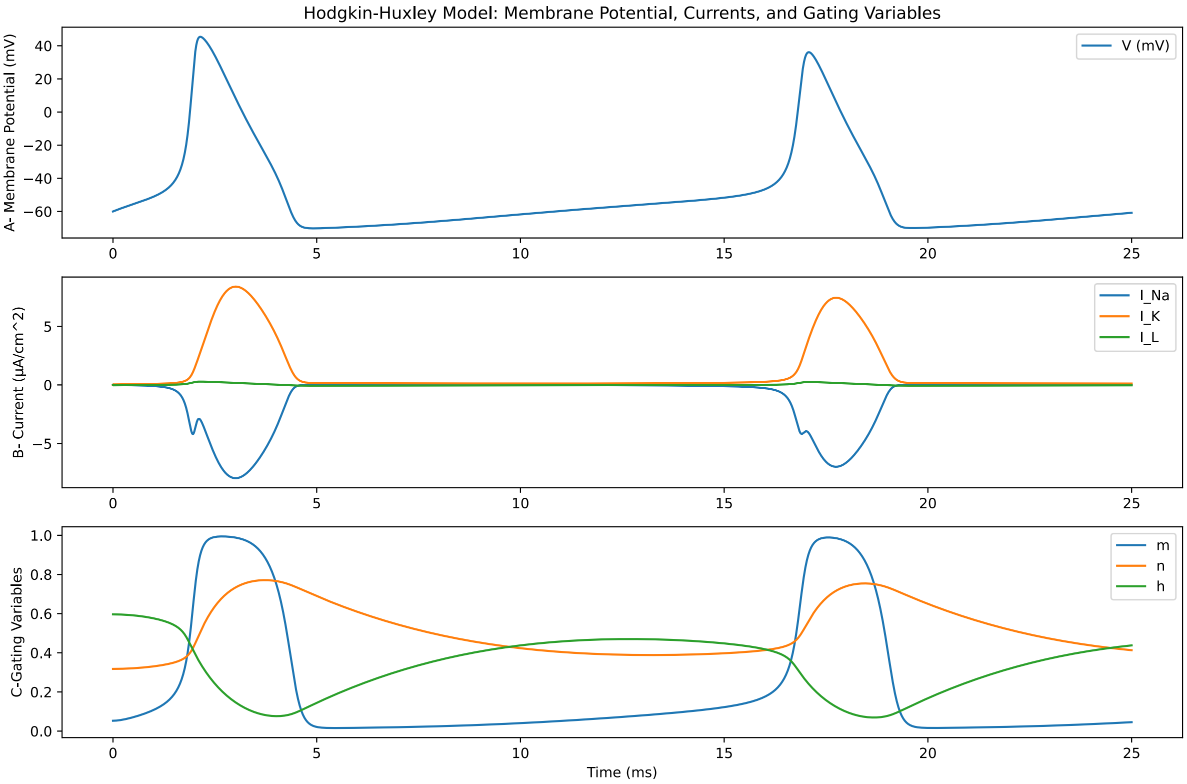

Figure 2 – Hodgkin-Huxley model behavior for commonly used parameters (SICILIANO, 2012), including continuous stimulus. The top figure shows the membrane potential; two action potentials are observed. With continuous stimulus, a new action potential is generated as soon as the refractory period from the last one ends. The middle figure shows the current in each ionic channel, illustrating the contributions of sodium and potassium channels to the overall action potential, and a minor contribution of the leak channel. The bottom figure shows the gating variables, highlighting the dynamic changes in ion channel states that underlie the fast depolarization and slow repolarization phases.

including the all-or-none response, through a set of coupled ordinary differential equations:

$$\frac{dU}{dt} = U(U-\alpha)(1-U) - W + I_e, \tag{2.6}$$

$$\frac{dW}{dt} = \epsilon(\beta U - \gamma W), \tag{2.7}$$

here, U represents the membrane potential, analogous to V in the HH model, while W captures the slower recovery process. The cubic nonlinearity in the equation for U is crucial for reproducing the threshold behavior characteristic of excitable cells. The parameters α, β, γ, and ϵ govern the system's response and can be adjusted to explore different physiological scenarios or simulate the effects of various external stimuli.

While the Hodgkin-Huxley model provides a more detailed representation of the action potential, the FHN model is still capable of reproducing the phases of the action potential waveform—depolarization, re-polarization, and recovery from hypers+hooting —

while retaining the essential excitability dynamics in this simpler form. The model's cubic nonlinearity captures the all-or-none response, while the two-variable system enables a simplified description of the excitation and recovery process, making it computationally efficient without sacrificing the qualitative behavior of excitable cells. The simplicity of the FHN model allows for clear phenomenological interpretations for its parameters in the context of excitable cells, such as neurons and cardiac myocytes. For example, K influences membrane excitability, α and β determine recovery kinetics, and ϵ controls the recovery timescale.

In this work, we explore the parameter space of the FHN model, focusing on how variations in initial conditions U_0 and W_0, as well as the parameter K, influence the system's behavior. Figures 3 showcase model behavior under single and continuous stimulus, Figures 4 and 5 illustrate the model's behavior under different initial conditions and parameter settings used in this work, showcasing its flexibility and utility in simulating excitable dynamics.

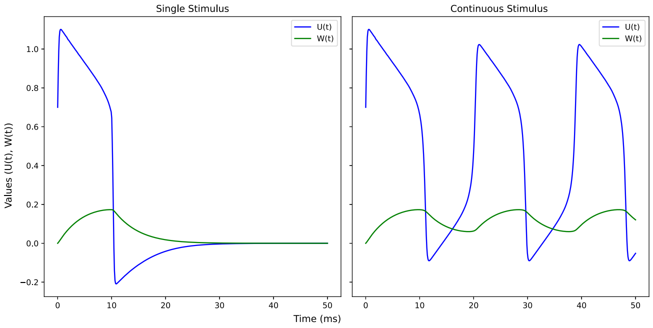

Figure 3 – FHN model behavior under single (left) and continuous stimuli (right).

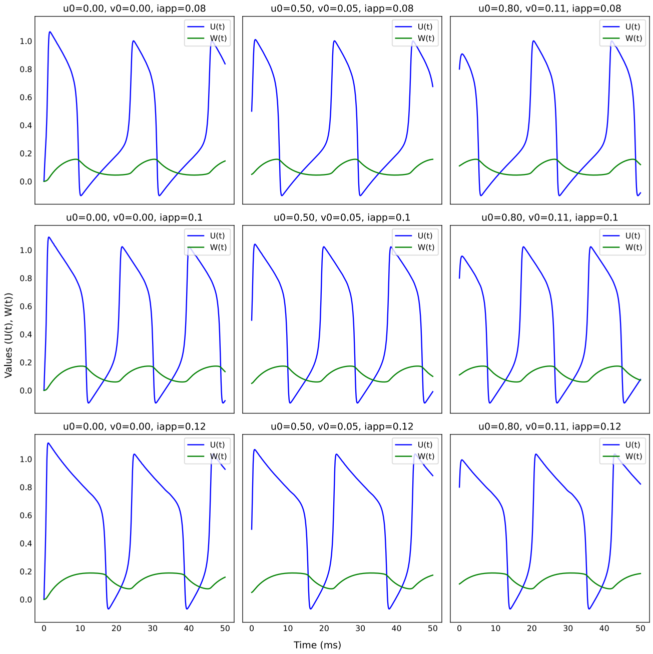

Figure 4 – FHN model behavior across parameter space.

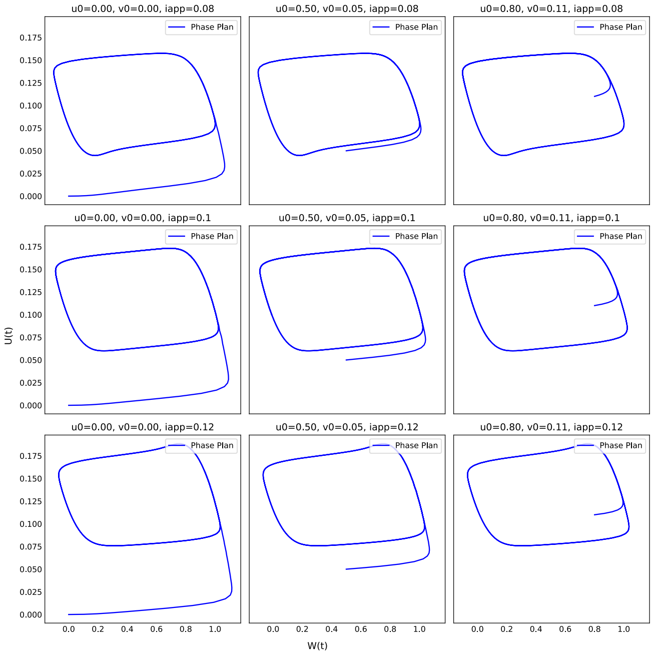

Figure 5 – FHN model phase plane analysis.

2.4 Numerical Methods

Most mechanistic or phenomenological action potential models, such as the HH and FHN, are given by non-linear systems of ODEs that do not permit analytical solutions. A common alternative is to use numerical methods (BUTCHER, 2016), to approximate the solution of these temporal derivatives by evaluating changes over small time intervals. Many such methods incorporate the Taylor series expansion, which approximates functions through their derivatives. Consider a differential equation of the form:

$$\frac{dy}{dt} = f(t, y).$$

Using the approximation obtained by the Taylor series expansion of $y(t + \Delta t)$ around t, truncate after the first term, we substitute the derivative, yielding the Euler Method:

$$y_{n+1} = y_n + \Delta t f(t_n, y_n), \tag{2.8}$$

where y_n represents the approximation of y at time t_n, and Δt is the time step. Being a first-order method, the scale of the local associated error is given by the truncated error term $O(\Delta t^2)$, yielding an global error of $O(\Delta t)$, meaning that for applications needing high accuracy, very fine time steps are required.

2.5 Data-Based Models: Regressors

In addition to conventional differential equation models, data-based models, such as regressors, have emerged as viable alternatives for analyzing and predicting complex systems. The process of obtaining a model that best explains a given set of observed data is known as regression, and the resulting models are termed regressors. Regressors seek to learn patterns directly from the input-output mapping (X → Y) available in the training data.

The regression process involves fitting coefficients to a specific base function to map the observed X → Y relationship:

$$Y = f(X, \beta), \tag{2.9}$$

here, X and Y represent the training data, f is a base regression function, and β is the set of coefficients to be determined during training. The goal of training a regressor is to minimize a loss function that encapsulates the model's ability to fit the data set. L_p error functions are commonly used, with the form:

$$L_p = \frac{1}{n}\left(\sum_{k=1}^{n} |f(x_k) - Y_k|^p\right)^{\frac{1}{p}}. \tag{2.10}$$

Thus, training a regressor model is essentially an optimization problem, with several approaches available. For models with simpler base functions, where the coefficients appear linearly, such as in polynomials and logarithms, linear regressors are frequently employed to take advantage of this simplicity. A common method is least squares minimization, where the loss function, containing the base functions, is easily differentiable. For a polynomial base function of degree n with $n + 1$ coefficients β, the loss function can be expressed as:

$$L = \|\mathbf{Y} - \mathbf{X}\beta\|_2^2, \tag{2.11}$$

which is known as the L_2**-norm squared**. Minimization is performed by finding the point where the gradient of the function is zero, representing the minimum point:

$$\nabla_\beta L = -2\mathbf{X}^T(\mathbf{Y} - \mathbf{X}\beta) = 0, \tag{2.12}$$

solving for β, we find the optimal set of coefficients:

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}. \tag{2.13}$$

However, base functions with linear dependence between coefficients may not be sufficient to explain the complex behavior of certain problems. In such cases, the minimization problem becomes more complex, resulting in a nonlinear system that is difficult to solve. A common strategy is to use the Gradient Descent Method (BOYD; VANDENBERGHE, 2004), where the parameters β are adjusted iteratively in the direction opposite to the gradient of the loss function L. This is expressed by updating the parameters:

$$\beta_i \leftarrow \beta_{i-1} - \eta\nabla_\beta L, \tag{2.14}$$

where η is the learning rate. This method seeks to find the optimal values of β that minimize the loss function and, thus, optimize the model's ability to fit the training data. It is worth noting that the calculation of the gradient of the loss function is still required, involving differentiation of the base function with respect to the coefficients. In some regressor models, this task is trivial, but in other cases, specific numerical differentiation techniques may be necessary.

2.6 Neural Networks

Neural networks, a specific type of regressor model, stand out for their remarkable ability to capture complex patterns in the $X \to Y$ relationship (HAYKIN, 2001). Inspired by the functioning of the biological nervous system, they consist of processing units called neurons, organized in layers, that perform mathematical operations on inputs to produce outputs. The learning capability of neural networks is achieved by adjusting the weights associated with connections between neurons.

Mathematically, a neural network model can be expressed as a composition of chained functions:

$$Y(X) = f^{(L)}\left(f^{(L-1)}\left(\dots f^{(2)}\left(f^{(1)}(X, \mathrm{w}^{(1)}), \mathrm{w}^{(2)}\right), \dots; \mathrm{w}^{(L-1)}\right), \mathrm{w}^{(L)}\right), \tag{2.15}$$

here, each $f^{(i)}$ represents the transformation applied by the i-th layer of the neural network. The first layer $f^{(0)}$ operates on the input values X, and the result is successively operated upon by subsequent layers until the final layer, whose output is Y; this process is called the forward pass. The $\mathrm{w}^{(i)}$ are the weights associated with each layer that need to be adjusted.

In most cases, the applied transformations consist of a linear transformation followed by a nonlinear activation function, that can be described with:

$$f^{(i)} = f(X^{(i)} \cdot \mathrm{w}^{(i)}), \tag{2.16}$$

where $\mathrm{w}^{(i)}$ is the matrix of coefficients for the linear transformation of the i-th layer. The activation function f is nonlinear, and common choices include the hyperbolic tangent $tanh(x)$ and the ReLU function $\max(0, x)$. Figures 6 illustrate a general neural network model.

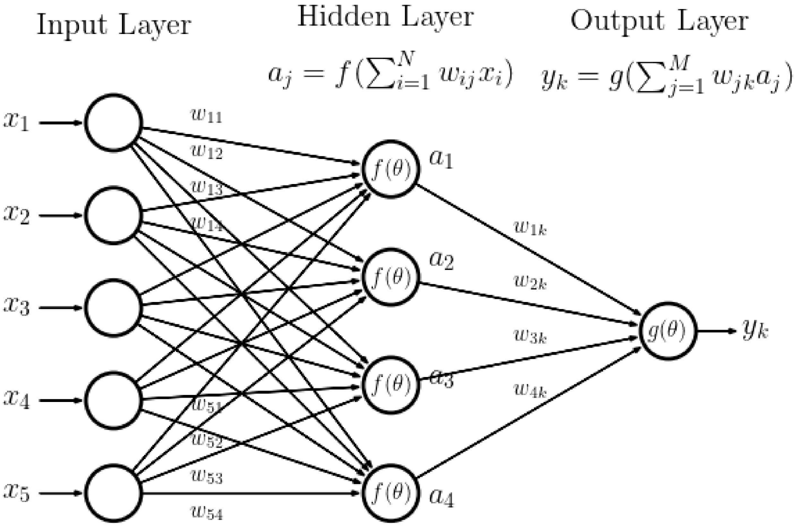

Figure 6 – A common neural network architecture represented as a typical graph scheme. The input x is first linearly transformed with a tensor of weights W, then a nonlinear function f is applied, and finally another linear transformation occurs to get the shape of the output.

The activations functions are predefined, and the training process involves determining the coefficients $\mathrm{w}^{(i)}$ for each layer. Therefore, training a neural network is an optimization process where the goal is to determine the linear transformation coefficients in each layer that minimize a loss function, for example:

$$L(X, Y, \mathrm{w}) = ||Y - f^{(L)}\left(f^{(L-1)}\left(\dots f^{(2)}\left(f^{(1)}(X, \mathrm{w}^{(1)})), \mathrm{w}^{(2)}\right), \dots, \mathrm{w}^{(L-1)}\right); \mathrm{w}^{(L)}\right) ||_2. \tag{2.17}$$

In the case of neural networks, the optimization process is particularly complex due to the large number of coefficients to be adjusted for each layer, which implies an N × N matrix of coefficients to be optimized. Another challenge is the difficulty in differentiating the loss function with respect to each component, due to the nonlinear relationships introduced by activation functions and the chained nature of the model. Consequently, the optimization process for neural networks typically employs a gradient descent method paired with a differentiation technique called backpropagation, for calculating the partial derivatives of the chained function.

The partial derivative of the loss function with respect to the output of the last layer, denoted as $\delta^{(L)}$, can be computed as:

$$\delta^{(L)} = \frac{\partial L}{\partial f^{(L)}}.$$

Error propagation to previous layers is performed iteratively. For the i-th layer, the partial derivative of the error with respect to the output of that layer, denoted as $\delta^{(i)}$, can be expressed as:

$$\delta^{(i)} = \left(\mathrm{w}^{(i+1)}\right)^T \delta^{(i+1)} \cdot f'^{(i)} \left(X^{(i)}\mathrm{w}^{(i)}\right. ,$$

where $f'^{(i)}$ represents the derivative of the activation function applied in the i-th layer.

Gradients with respect to the parameters $\mathrm{w}^{(i)}$ are then calculated for each layer using the accumulated partial derivatives. For the i-th layer, the gradient is given by:

$$\frac{\partial L}{\partial \mathrm{w}^{(i)}} = \delta^{(i)} \cdot \left(f^{(i-1)}\right)^T. \tag{2.18}$$

These partial derivatives are used to update the weights during the optimization process with the gradient descent method (SGD), adjusting the model parameters in the direction of the negative gradient of the loss function:

$$\mathrm{w}_i \leftarrow \mathrm{w}_{i-1} - \eta \frac{\partial L}{\partial \mathrm{w}}. \tag{2.19}$$

However, in neural networks, SGD is often not enough (BOTTOU; CURTIS; NOCEDAL, 2018). SGD updates the weights by following the direction of the negative gradient, but it relies on a fixed learning rate and may struggle with issues like slow convergence, getting stuck in local minima, or oscillating around saddle points, especially in complex, high-dimensional spaces typical of deep learning models.

To address these issues, more sophisticated algorithms like Adam (Adaptive Moment Estimation) (KINGMA; BA, 2015) are often preferred. Adam is a combination of two other optimization methods: RMSProp, which adjusts the learning rate using a moving average of squared gradients to prevent updates from becoming excessively large or small, and SGD with momentum, which introduces a momentum term through a moving average of gradients to smooth updates and enhance convergence.

For the Adam method, in a given iteration i, the gradient of the loss function $L(\mathrm{w})$ with respect to the parameters w is first computed:

$$g_i \leftarrow \frac{\partial L}{\partial \mathrm{w}}(\mathrm{w}_i).$$

Then the first moment estimate m_i, a moving average of the gradients, is updated using the equation:

$$m_i \leftarrow \beta_1 m_{i-1} + (1 - \beta_1) g_i,$$

where β_1 is commonly set to 0.9 and regulates the decay rate of this average. This is the same term employed in SGD with momentum to smooth learning by leveraging past gradients.

Similarly, the second moment estimate v_i, a moving average of the squared gradients, is updated as follows:

$$v_i \leftarrow \beta_2 v_{i-1} + (1 - \beta_2) g_i^2,$$

where β_2 is typically set to 0.999, also controlling its decay rate. This is the same term present in RMSProp to control the magnitude of the updates.

Finally, the parameters w are updated using:

$$\mathrm{w}_{i+1} \leftarrow \mathrm{w}_i - \frac{\eta m_i}{\sqrt{v_i} + \epsilon},$$

where η denotes the learning rate, and ϵ (a small constant, typically 10^{-8}) ensures numerical stability. This sophisticated adaptive mechanism allows Adam to efficiently navigate complex loss landscapes typical in neural network training, dynamically accelerating convergence in flatter regions of the loss surface and slowing down in steeper areas, leading to more efficient optimization.

The universal approximation theorem (HORNIK; STINCHCOMBE; WHITE, 1989) asserts that a feedforward neural network with a single hidden layer, containing a finite number of neurons, can approximate any continuous function on compact subsets of $\mathbb{R}^n$, given a non-linear activation function such as the sigmoid or ReLU. This theorem highlights the remarkable capacity of neural networks to model a wide range of functions, applicable to both regression and classification tasks. Although a single hidden layer is theoretically sufficient for function approximation, any practical use would require an impractically tall single layer, therefore increasing the number of hidden layers is often preferred.

Large models with more hidden layers have more expressive power and can learn more intricate patterns, however they are harder to train and more expensive to use once trained. Various different kinds of network topologies have been developed to fine-tune the neural network's capabilities for specific applications. Notable developments include, Large Language Models (LLMs) (BROWN et al., 2020), very deep networks with transformer units (VASWANI et al., 2017), which have revolutionized natural language processing by efficiently handling long-range dependencies and enabling parallel processing, and convolutional neural networks (CNNs) (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), which excel in image recognition tasks due to their ability to learn spatial hierarchies.

2.7 Physics-Informed Neural Networks (PINNs)

Physics-Informed Neural Networks are a class of neural networks designed to solve problems where the governing equations of a system are known. Most dynamic systems of interest can be described by a set of differential equations, derived from conservation laws, known physical phenomena, or empirical observation. Traditionally, neural networks rely purely on data for training, but PINNs leverage this prior system knowledge, incorporating the governing equations directly into the training process. This allows the network to learn solutions that respect both the observed data and the underlying physical laws, making them highly effective for modeling physical systems.

Mathematically, PINNs enforce the system governing equations, ODEs or PDEs, with a new term in the loss function that accounts for the residual of the network's prediction in relation to the prescribed equations. Consider a system described by the PDE:

$$\mathcal{N}(u) = 0, \quad u = u(x,t), \tag{2.20}$$

where $\mathcal{N}(u)$ represents a differential operator applied to the unknown solution $u(x,t)$. A new term can be introduced to the loss function, to minimize the residuals between the network's prediction u_{w} and the governing differential equations, over a solution space ω:

$$L_{\mathrm{physics}} = \int_{\omega} \mathcal{N}(u_{\mathrm{w}}) d\omega. \tag{2.21}$$

This new term enforces the system's physical constraints by penalizing deviations from the expected behavior dictated by the governing differential equation. It is important to note that $\mathcal{N}$ is a differential operator, that is, it is described by a function of one (ODEs) or more (PDEs) derivatives of the solution function $u(x,t)$. Therefore, the minimizing term is also a differential function, requiring the derivatives of the network's prediction in relation to its arguments, and this is computed using automatic differentiation (see section 2.8).

For instance, consider this form for the operator:

$$\mathcal{N}(u) = \frac{\partial u}{\partial t} - \nu \frac{\partial^2 u}{\partial x^2} - f(x,t) = 0, \tag{2.22}$$

where $u(x,t)$ is the unknown solution, ν is a physical parameter (e.g., diffusion coefficient), and $f(x,t)$ is a source term. The goal is to minimize the residual of this PDE:

$$L_{\text{physics}} = \int_0^T \int_0^X \left(\frac{\partial u_{\text{w}}(x,t)}{\partial t} - \nu \frac{\partial^2 u_{\text{w}}(x,t)}{\partial x^2} - f(x,t) \right)^2 dx\, dt. \tag{2.23}$$

Of course, this residual function is too hard to track analytically, so a numerical approximation is made, estimating the error over the domain by sampling a set of points from the parameter space:

$$L_{\text{physics}} = \frac{1}{N_r} \sum_{i=1}^{N_r} \left(\frac{\partial u_{\text{w}}(x_i,t_i)}{\partial t} - \nu \frac{\partial^2 u_{\text{w}}(x_i,t_i)}{\partial x^2} - f(x_i,t_i) \right)^2, \tag{2.24}$$

where N_r is the batch size, a training meta-parameter, and at each sample i, the residual function is evaluated. The sampling process is arbitrary but generally seeks to produce homogeneous samples, and more sophisticated methods can dynamically sample more from higher error regions. Other kinds of residual functions can also be used, penalizing more or less sharp deviations.

In order to perform that evaluation, the derivative of the neural network's prediction in relation to its parameters $\frac{\partial u_{\text{w}}(x_i,t_i)}{\partial t}$ and $\frac{\partial^2 u_{\text{w}}(x_i,t_i)}{\partial x^2}$ must be known, which also requires an approximation. This is done by assuming the predicted solution $u_{\text{w}}(x,t)$ to be differentiable — a reasonable assumption, since all the activation functions are differentiable — and using automatic differentiation (AD) to get approximations of the derivatives required for the computation.

To analytically and numerically fix a solution among a family of solution functions in a differential equation, an initial condition or boundary condition is required. With PINNs, it is no different; the boundary conditions are also enforced in the solution. Each boundary condition enforces another governing equation but applied only to a certain part of the parameter space. These equations may also include functions of the input parameters, allowing them to be parameterized, or derivatives of the output, requiring AD as well. For the example system, the boundary condition constraints enforce the initial condition of the solution at $t = 0$ and a Neumann boundary at the borders Ω:

$$L_{\text{IC}} = \int_0^X \left(u_{\text{w}}(x, t=0) - f_1(x, \gamma_1)\right)^2 dx, \tag{2.25}$$

$$L_{\text{B}} = \int_0^T \left(\frac{\partial u_{\text{w}}(x=\Omega, t)}{\partial x} - f_2(t, \gamma_2) \right)^2 dt, \tag{2.26}$$

where f_1 and f_2 are arbitrary functions, and γ_1 and γ_2 are coefficients that can be fixed or an input of the neural network, allowing for boundary condition parametrization. Similar to the main physics constraint, the boundary terms are also approximated using a finite number of points:

$$L_{\mathrm{B1}} = \frac{1}{N_r} \sum_{i=1}^{N_r} \left(u_{\mathrm{w}}(x_i, t=0) - f_1(x_i, \gamma_1)\right)^2 , \tag{2.27}$$

$$L_{\mathrm{B2}} = \frac{1}{N_r} \sum_{i=1}^{N_r} \left(\frac{\partial u_{\mathrm{w}}(x=\Omega, t)}{\partial x} - f_2(t, \gamma_2)\right)^2 . \tag{2.28}$$

The final training loss function includes the main physical constraint, the boundary constraint, and may or may not also include a traditional data constraint:

$$L = \phi_1 L_{\mathrm{Physics}} + \phi_2 L_{B1} + \phi_3 L_{B2} + \phi_4 L_{\mathrm{Data}}, \tag{2.29}$$

where vector ϕ is a set of weights, with one for each term of the loss function. These weights are training meta-parameters, they can be fixed or dynamically adjusted during the training process. However, it is common to have the boundary terms be much more weighted than the others, to force an early fixation of the solution.

In summary, PINNs provide a powerful framework that blends data-driven machine learning with the underlying physics of a system, allowing for the efficient solution of differential equations while respecting physical laws. By incorporating both empirical data and known governing equations, PINNs significantly reduce the need for extensive datasets, often required by purely data-driven models. Their flexibility and ability to generalize across various physical systems make them a promising tool for advancing the fields of scientific computing, engineering, and medicine, where accurately modeling complex, dynamic processes is crucial.

2.8 Automatic Differentiation

Automatic differentiation (AD) (BAYDIN et al., 2018) is a computational technique for calculating the derivatives of functions defined by computer programs. It is one of the core components of neural network training and physics-informed learning. AD is neither symbolic differentiation nor numerical differentiation (like finite differences); instead, it systematically applies the chain rule to elementary operations of a function to compute derivatives efficiently and accurately. It takes advantage of the fact that any computer function can be broken down into a series of elementary operations and propagates derivatives to machine precision, avoiding truncation errors common in numerical methods, and introduces minimal computational overhead. This makes AD an essential tool in machine learning algorithms and scientific computing applications, where complex derivatives must be computed at each training iteration.

There are two primary modes of automatic differentiation. Forward mode propagates the derivatives with respect to one independent variable forward in a chain to the output function, calculating its derivative with respect to that one variable. Take

$Y = f(X)$ where $f : \mathbb{R}^n \to \mathbb{R}^m$. Forward AD calculates $\frac{\partial f}{\partial x_a} = [\frac{\partial f_1}{\partial x_a}, \frac{\partial f_2}{\partial x_a}, \frac{\partial f_3}{\partial x_a}, \dots, \frac{\partial f_n}{\partial x_a}]$ in one pass of the chain, where x_a is an arbitrary component of input vector X.

To build the chain, f is broken down into elementary operations, and their derivatives are added according to the chain rule. For example, if f involves intermediate calculations z, then:

$$\frac{\partial f}{\partial x_a} = \sum_{j=1}^{k} \frac{\partial f}{\partial z_j} \frac{\partial z_j}{\partial x_a}, \tag{2.30}$$

where z_j are the intermediate operations. The resulting chain has the form:

$$\frac{\partial f}{\partial x_a} = \frac{\partial x_a}{\partial x_a} \cdot \frac{\partial z_1}{\partial x_a} \cdot \frac{\partial z_2}{\partial z_1} \cdot \dots \cdot \frac{\partial z_{k-1}}{\partial z_k} \cdot \frac{\partial f}{\partial z_k}. \tag{2.31}$$

An initial value for $\frac{\partial x_a}{\partial x_a} = 1$ is known as a seed. The function is evaluated one elementary operation at a time, with each derivative being accumulated. After propagating the derivatives through the entire function, one obtains:

$$\frac{\partial Y}{\partial x_a} = \begin{bmatrix} \frac{\partial f_1}{\partial x_a} \\ \frac{\partial f_2}{\partial x_a} \\ \vdots \\ \frac{\partial f_m}{\partial x_a} \end{bmatrix}, \tag{2.32}$$

which represents the derivatives of all the components of the function with respect to a single independent variable. If derivatives of more independent variables must be known, more passes are required. Forward AD is therefore particularly efficient when the number of inputs is smaller than the number of outputs. It is useful in some cases of PINNs where a constraint is enforced on multiple derivatives of some input variable, $L_{\text{physics}} = F(\frac{\partial f_1}{\partial x_a}, \frac{\partial f_2}{\partial x_a}, \frac{\partial f_3}{\partial x_a}, \dots, \frac{\partial f_n}{\partial x_a})$, but it is not generally used for neural networks.

Reverse-mode AD, on the other hand, propagates the derivatives backward from the outputs to the inputs. By tracking a single component of the output f over the chain all the way to the input vector X, its gradient with respect to the inputs, $\frac{\partial f_a}{\partial X} = [\frac{\partial f_a}{\partial x_1}, \frac{\partial f_a}{\partial x_2}, \frac{\partial f_a}{\partial x_3}, \dots, \frac{\partial f_a}{\partial x_n}]$, is calculated, for an arbitrary component f_a at each pass.

Consider a function $Y = f(X)$, where $f : \mathbb{R}^n \to \mathbb{R}^m$. Reverse-mode AD calculates the gradient of a particular output component f_a with respect to the entire input vector X:

$$\frac{\partial f_a}{\partial X} = \begin{bmatrix} \frac{\partial f_a}{\partial x_1} \\ \frac{\partial f_a}{\partial x_2} \\ \vdots \\ \frac{\partial f_a}{\partial x_n} \end{bmatrix}. \tag{2.33}$$

To build the chain, the function f is broken down into elementary operations, and their derivatives are added according to the chain rule. For example, if f involves intermediate calculations through variables z_j, the chain rule applied in reverse-mode is:

$$\frac{\partial f_a}{\partial X} = \sum_{j=1}^{k} \frac{\partial f_a}{\partial z_j} \frac{\partial z_j}{\partial X}. \tag{2.34}$$

In reverse-mode AD, the derivatives are accumulated by traversing the chain backward, starting from the output. Each intermediate derivative is stored and reused, significantly reducing the computational cost when calculating the gradient of f_a with respect to all input variables. This method is especially efficient when the number of inputs n is much larger than the number of outputs m.

Backpropagation is a specific application of reverse-mode AD used for training neural networks. It calculates the gradient of the scalar loss function L with respect to the network's parameters. The loss function is a single scalar output, and its gradient with respect to all parameters must be computed for optimization. During backpropagation, the computational graph is traversed backward, applying the chain rule to propagate the error from the output layer to the input layer.

For a neural network, the chain rule for backpropagation can be expressed as:

$$\frac{\partial L}{\partial \mathrm{w}_i} = \frac{\partial L}{\partial z_k} \cdot \frac{\partial z_k}{\partial z_{k-1}} \cdot \ldots \cdot \frac{\partial z_1}{\partial \mathrm{w}_i}, \tag{2.35}$$

where w_i are the parameters of the network, and the intermediate variables z_j represent the activations at each layer of the network. The gradients are accumulated layer by layer, starting from the output and moving toward the input, which efficiently computes the full gradient for the network parameters, used in each training iteration:

$$\frac{\partial L}{\partial \mathrm{w}} = \begin{bmatrix} \frac{\partial L}{\partial \mathrm{w}_1} \\ \frac{\partial L}{\partial \mathrm{w}_2} \\ \vdots \\ \frac{\partial L}{\partial \mathrm{w}_n} \end{bmatrix}. \tag{2.36}$$

Backpropagation allows for efficient gradient calculations by reusing intermediate values, making it essential for training large neural networks. Reverse-mode AD is also widely used in PINNs, where it is necessary to compute derivatives of the output with respect to multiple input variables (e.g., space and time).

3 METHODS

In this section, the tools used to produce neural network and evaluate their accuracy and performance in relation to numerical methods are detailed.

3.1 CUDA Optimized Numerical Solutions and Training Sets

To generate training data for the regression model and to establish a baseline for comparison with the trained networks - both in terms of accuracy and computational cost - the Fitzhugh-Nagumo model was numerically solved using the Euler method. The resulting scheme for the FHN model with the Euler method is:

$$U_{n+1} = U_n + \Delta t \left[K U_n (U_n - \alpha)(1 - U_n) - W_n + I_{app} \right], \tag{3.1}$$

$$W_{n+1} = W_n + \Delta t \left[\epsilon (\beta U_n - 0.8 W_n) \right]. \tag{3.2}$$

The error associated with this numerical scheme is proportional to the time discretization Δt, while the computational cost is inversely proportional. To balance these aspects, the neural networks were compared with the Euler method using $\Delta t = 0.1$, for stability reasons. The Euler Method, despite being a first-order method and with many more sophisticated options available, was chosen for this comparison because its simplicity allows for a clear assessment of how neural networks can potentially offer advantages in solving differential equations. It follows that if neural networks cannot surpass this baseline efficiency of the Euler method, they would not be better than any other option.

To achieve optimal computational performance with the Euler method, an optimized implementation was developed in CUDA/C++. The CUDA code is designed to accept an input file where each line contains a set of parameters and initial conditions to be solved, and to execute the numerical scheme in parallel using the GPU. Optimization techniques, such as those in the CUDA guide (NVIDIA, 2023), were employed, focusing in efficient flow control and memory access, in addition to performance focused targeted compilation with the O3 and architecture flags, thus providing extremely efficient computation.

To replace numerical ODE solvers with neural networks, the training sets used in this work consist of numerical solutions of the FHN model obtained via the Euler method. These sets are generated by uniformly sampling the parameter space and computing approximate solutions at specific time intervals. Each row in the set corresponds to a combination of inputs (model parameters and time points) and outputs (model solutions). The dataset comprises solutions for multiple parameter sets, sampled at different time points, and is randomly reshuffled during training. The data is split into training and validation sets, ensuring no overlap between them.

3.2 Neural Network and Training

In this work, multiple problems are handled with the neural networks. In each case, many different architectures were used to find optimal solutions. These model architectures share the same overall design with some parameterized options that allow generating multiple models. The loss functions also vary for each model and problem with the Adams minimization method employed in every case.

The models have the general form of an MLP, or multi-layer perceptron, which is a fully connected neural network architecture. This architecture consists of multiple layers of neurons: an input layer, one or more hidden layers, and an output layer, where each neuron in each layer is connected to every neuron in the following layer. The neurons in each hidden layer apply nonlinear activation functions. In Figure 7 we have a scheme representing the model topology employed. In this work multiple problems in the form f(X) = Y are tackled, but the same base neural network structure is used in all of them.

The architecture was parameterized with respect to the number of hidden layers, the size of each hidden layer and the nonlinear activation function in each hidden layer's neurons. Therefore the resulting MLP model's architecture can be described by a sequence of layers each with two attributes: size and activation function. This allows models with heterogeneous architecture, where the size of the hidden layers and the activate functions is not the same for every layer, as well as homogeneous architecture to be deployed. This is important in fine tuning the models, in order to produce the smaller (number of neurons) possible model to satisfy certain criteria.

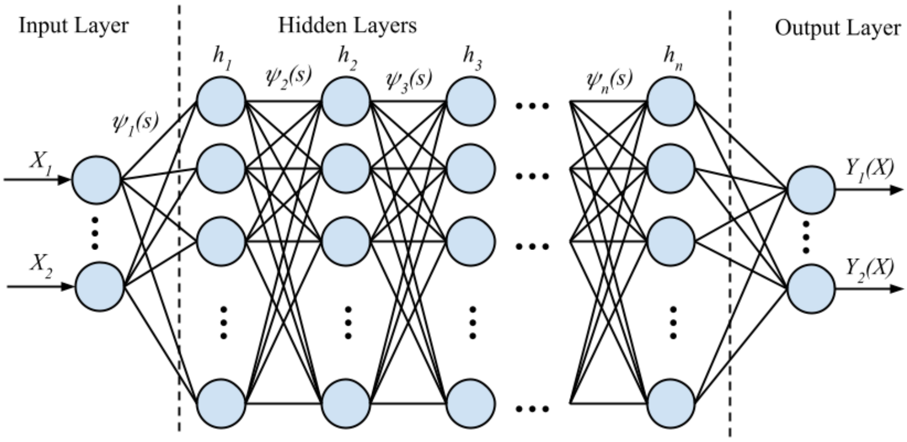

Figure 7 – MLP (Multi-Layer Percepton) base architecture used in this work. The model learns the relation between inputs X and outputs Y. The model may have multiple hidden layer h_i each with a different amount of neurons and activation function. Training process involves determining best set of coefficients ψ to minimize a loss function.

For each problem, a distinct loss functions is formulated, comprising a combination of loss terms (or constraints), each assigned an individual weight. The constraints might enforce closeness to a training set (data constraints) or enforce known physical laws (physics constraints). Regardless of the composition of the loss function, the training process is the same. The loss function is minimized using the Adams algorithm (learning rate = 1e-3), running for a million of interactions with no stop criteria. Additionally, a scheduling technique know as Reduce on Plateau (factor=0.999999, patience=1000, threshold=1e-3, min learning rate = 1e-5, eps = 1e-08) was employed to slightly reduce the learning rate when training staggered.

The validation error and the loss of each term in the learning function were recorded along the training for further analysis. The validation metrics are recorded at each 10000 training steps, when the error for each sample in the validation set is calculated, and the mean and maximum errors over the set are recorded. The error of each sample is defined as:

$$V^i = |U^i_{\text{pred}} - U^i_{\text{true}}| + |W^i_{\text{pred}} - W^i_{\text{true}}|, \tag{3.3}$$

where U^i_{true} and W^i_{true} are the numerical solutions for the ith samples, and U^i_{pred} and W^i_{pred} are the network predictions for them.

3.2.1 Architecture Grid Search and Training Parallelization

Optimizing neural networks to replace numerical solvers for ODEs requires not only training the models but also determining the optimal network architecture. This task is complex due to the lack of general heuristics in the literature for selecting architectures, making an exhaustive grid search a viable approach. By systematically exploring combinations of architectural parameters, grid search helps identify models that balance performance and computational cost.

In this work, we conducted a grid search to fine-tune neural network architectures for several of the tackled problems. Each architecture is defined by a sequence of layers, where each layer is characterized by two parameters: the number of neurons and the activation function. Given a set of layer counts, such as $[2, 3, 4]$, and a set of layer configurations, such as $(\text{TANH}, 16), (\text{TANH}, 32), (\text{RELU}, 16), (\text{SINH}, 16)$, every possible combination of these layers, including repetitions, was explored. For example, all architectures with 2, 3, and 4 layers, composed of permutations of the provided configurations, are included in the search, as shown in Figure 8. Additional training parameters such as batch size, and normalization are also included to efficiently test this techniques, as well as a parameter to include or not a physics informed constraints, to easily compare purely data-driven models with PINNs.

The grid search process is computationally intensive, as it involves training numerous models over millions of iterations. To mitigate this, an MPI-based implementation was developed to parallelize the grid search across a multi-GPU cluster. The code takes the search space as input, distributing model training tasks across available GPUs, allowing different models to be trained in parallel to leverage the cluster capabilities.

During training, statistics on model evolution, iteration times, and final model states are recorded to a shared file system for subsequent analysis. Post-training, this data is used to generate visualizations of loss and validation error evolution, highlighting regions of high error in the domain. Furthermore, ANOVA models are employed to evaluate the influence of model attributes on performance metrics, such as inference speed or accuracy. These models use predictor variables describing model characteristics and the attribute of interest as the response variable, providing a more rigorous assessment of each technique's impact.

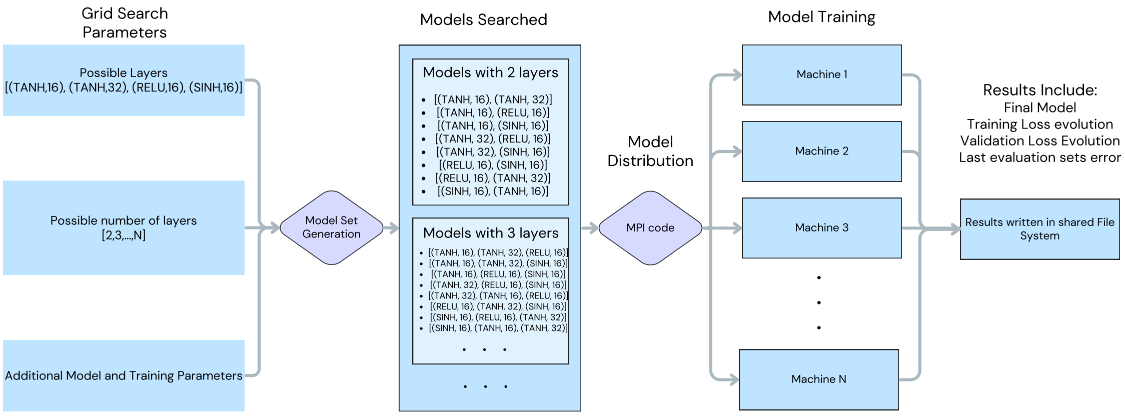
Grid Search Parameters
Possible Layers
[(TANH,16), (TANH,32), (RELU,16), (SINH,16)]
Possible number of layers
[2,3,...,N]
Additional Model and Training Parameters
Model Set Generation
Models Searched
Models with 2 layers
[(TANH, 16), (TANH, 32)]
[(TANH, 16), (RELU, 16)]
[(TANH, 16), (SINH, 16)]
[(TANH, 32), (RELU, 16)]
[(TANH, 32), (SINH, 16)]
[(RELU, 16), (SINH, 16)]
[(RELU, 16), (TANH, 32)]
[(SINH, 16), (TANH, 16)]
Models with 3 layers
[(TANH, 16), (TANH, 32), (RELU, 16)]
[(TANH, 16), (TANH, 32), (SINH, 16)]
[(TANH, 16), (RELU, 16), (SINH, 16)]
[(TANH, 32), (RELU, 16), (SINH, 16)]
[(TANH, 32), (TANH, 16), (RELU, 16)]
[(RELU, 16), (TANH, 32), (SINH, 16)]
[(SINH, 16), (RELU, 16), (TANH, 32)]
[(SINH, 16), (TANH, 16), (TANH, 32)]
. . .
. . .
Model Distribution
MPI code
Model Training
Machine 1
Machine 2
Machine 3
Machine N
Results Include:
Final Model
Training Loss evolution
Validation Loss Evolution
Last evaluation sets error
Results written in shared File System

Figure 8 – Scheme showing grid search of neural network models and MPI code to implement model distribution and training parallelization across multiple machines.

3.3 Model Classes Employed

In this section the three classes of models used to tackled the multiple problems in this work are presented. They follow the same described MLP parameterized architecture, use the same minimization Adam's algorithm, but employs different techniques to learn the solution of the FHN model, mainly, using different types of loss functions.

3.3.1 Data-Driven Neural Network Models

The first class of models employed is purely data-driven models, henceforth referred to as DDNN. These models take the following form:

$$u_{\mathrm{ddnn(w)}}(t,w) \approx U, W, \tag{3.4}$$

where U and W are the solutions of the FitzHugh-Nagumo model, and u_{model} represents a multi-layer perceptron regressor with parameters w. The MLP is trained to approximate the solutions U and W of the FHN model, learning the continuous behavior of the system over time from observed data. Additionally, this model may also learn effects of some parameters on the solution, with ϕ including parameters of the FHN model or initial conditions.

The model are trained by minimizing a loss function over a dataset consisting of numerical solutions of the FHN model:

$$L_D = \frac{1}{N_{\mathrm{set}}} \sum_{i=1}^{N_{\mathrm{set}}} \frac{1}{2} \left(u_{\mathrm{ddnn}}(\mathrm{w})(t_i, \phi_i) - u_{\mathrm{solver}}(t_i, \phi_i)\right)^2, \tag{3.5}$$

where the term ϕ may include initial conditions or FHN parameters. For a model with both initial conditions U_0 and W_0 parameterized along with the FHN model parameter K, the loss function is:

$$L_D = \frac{1}{N_{\mathrm{set}}} \sum_{i=1}^{N_{\mathrm{set}}} \frac{1}{2} \Big(u_{\mathrm{ddnn}}(\mathrm{w})(t^i, U_0^i, W_0^i, K^i) - u_{\mathrm{solver}}(t^i, U_0^i, W_0^i, K^i)\Big)^2, \tag{3.6}$$

where U_0, W_0, and K are the initial conditions and parameters of the FHN model in each sample i. The training set is not taken as whole at each iteration of the training process. Instead, at each iteration a random subset of samples, of a pre-determined size, is selected and used to assemble a training batch. At each training iteration, the loss evaluated is:

$$L_D = \frac{1}{N_{\mathrm{batch}}} \sum_{i=1}^{N_{\mathrm{batch}}} \frac{1}{2} \Big(u_{\mathrm{ddnn}}(\mathrm{w})(t^i, U_0^i, W_0^i, K^i) - u_{\mathrm{solver}}(t^i, U_0^i, W_0^i, K^i)\Big)^2. \tag{3.7}$$

It is also possible to include various training datasets with different weights; for an arbitrary number of sets N we have the final loss as the weighted sum:

$$L_t = \sum_{i=1}^{N} L_{D^i} \lambda^i, \tag{3.8}$$

where D^i is the i-th dataset with weight λ^i.

3.3.2 Physics-Informed Neural Network Models

The second class of models employed is Physics-Informed Neural Networks, where the model leverages both data and the governing physical laws of the system. The FHN's model differential equations are directly incorporated into the loss function to improve training. The model learns the solution along the continuous time axis, with additional constraints imposed on the derivatives of this solution, based on the system's prescribed equations. It has the following form:

$$u_{\text{pinn}}(\text{w})(t,\phi) \approx U, W. \tag{3.9}$$

$$\frac{\partial u_{\text{pinn}}(\text{w})(t,\phi)}{\partial t} \approx f(t,\phi). \tag{3.10}$$

The neural network $u_{\text{pinn(w)}}(t,\phi)$ is also a MLP, but beyond only learning the solutions for U and W based on observed data, it also simultaneously satisfies the FHN model's governing equations $f(t,\phi)$, a set of two ODEs, prescribing values for $\frac{\partial U}{\partial t}$ and $\frac{\partial W}{\partial t}$. They are:

$$f_u(t,\phi) = KU(t)(U(t) - (1 - U(t)) - W(t) + I_{\text{e}}, \tag{3.11}$$

$$f_w(t,\phi) = \eta(\beta U(t) - \gamma W(t)), \tag{3.12}$$

where the model parameter ϕ can change some of the coefficients. In each training iteration, a batch is sampled from the parameter space, with it's size being a training hyperparameter. At each sample in the batch the derivatives are estimated using Automatic Differentiation and the PINN constraint expression is evaluated, the residuals from the entire batch are aggregated with the $L2$ norm. Considering a model with a FHN parameter K and both the inital conditions parameterized, the final expression is:

$$L_{\text{physics}} = \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} \left(\frac{\partial u_{\text{pinn(w)}}(t^i, U_0^i, W_0^i, K^i)}{\partial t} - f(t^i, U_0^i, W_0^i, K^i) \right)^2. \tag{3.13}$$

The expression is evaluated with homogeneous samples from the whole parameter space. Additional constraints can also be enforced for specific regions of the domain, and may also have different weights:

$$L_{\text{physics}} = \sum_{i=1}^{N} L_{\text{physics}}^i \lambda^i, \tag{3.14}$$

with the final physics loss L_{physics} being composed of losses taken from multiple regions i of the parameter space. This is useful to increase the amount of information in critical regions of the domain.

An additional PINN constraint is also required to enforced the initial conditions for U, W at $t = 0$. The loss evaluate points at the border $t = 0$ and enforces the prescribed values:

$$L_B = \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} \left(u_{\text{pinn}}(t = 0, U_0, W_0, K) - (U_0, W_0)\right)^2, \tag{3.15}$$

where U_0, W_0 might be fixed or model parameters. In most cases, the PINN models combines data-driven and physics informed learning, therefore also including a training dataset and it's data loss term. The final loss function is the weighted sum of the PINN terms and the data term, given by:

$$L_{\text{total}} = \lambda_D L_D + \lambda_P L_{\text{physics}} + \lambda_B L_B. \tag{3.16}$$

3.3.3 Iterator Neural Network Model

The final class of neural networks explored in this work is also a data-based but that doesn't learn the solution over the whole time domain, but instead learn to advance the solution for some discrete time interval, much like a numerical method. They take the form:

$$u_{\text{IT}}(U, W, \phi) \approx U(t + \Delta t), W(t + \Delta t), \tag{3.17}$$

where Δt is the time interval that each pass in the network advances the solution. To evaluate a time domain $t \in (0, t_f)$, the network starts with a initial condition U_0, W_0 advances the solution until t_f incrementing Δt per pass:

$$U_f, W_f = u_{\text{IT}}(U_{f-1}, W_{f-1}, \phi) \ldots = u_{\text{IT}}(U_0, W_0, \phi). \tag{3.18}$$

To learn the update function, the model learns on observed data. The dataset is assembled, similarly to the previous ones, but with the format $U_t, W_t, \phi \rightarrow U_{t+\Delta t}, W_{t+\Delta t}$ with data generated numerically with the Euler method. A training loss is constructed to enforce simillarity to the set, learning the relation $Y_t \rightarrow y_{t+\Delta t}$. Allowing for parametrization with a ϕ parameter, the resulting training loss has the form:

$$L_D = \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} \left(u_{\text{IT}}(U_t, W_t, \phi) - (U_{t+i}, W_{t+1})\right)^2. \tag{3.19}$$

An important parameter for this type of model is Δt, the time step that the network learns to advance the solution per pass. Larger Δt reduces the number of iterations needed to reach a final time, making the model faster, but also requiring it to learn more complex dynamics over larger time intervals, often necessitating larger, more expressive networks. Conversely, smaller Δt allows for more accurate training with smaller models that have faster inference times, but at the cost of requiring more passes to reach the same final time. Balancing Δt and model size is critical. If Δt approaches the discretization of the numerical method (e.g., 0.1 for Euler), the Iterative Neural Network (ITNN) might fail to outperform traditional methods, as each iteration would not be computationally cheaper.

3.4 Advanced Training Techniques

3.4.1 Subdividing the Time Domain

A key challenge in the FHN model is the heterogeneous behavior of the solution across the time domain. The action potential generation features a rapid activation phase, followed by a slower recovery and eventually a prolonged steady-state, leading to distinctly different behaviors over time. This variability in time scales makes it difficult for models, particularly smaller ones with limited expressive power, to capture the entire solution accurately. To address this problem, the temporal domain can be effectively divided in smaller pieces to be solved separately, as illustrated in Figure 9. This approach enables models to more easily capture the diverse dynamics present in different phases of the solution.

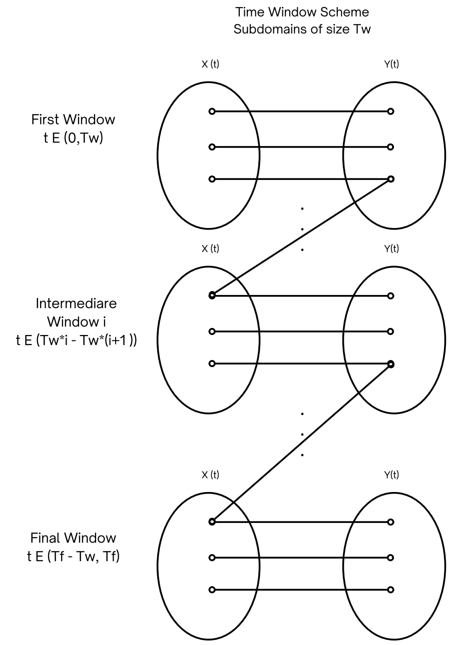

Figure 9 – Temporal domain divided in subdomains to be solved separately. Continuity is ensured by including shared border points present in neighboring windows.

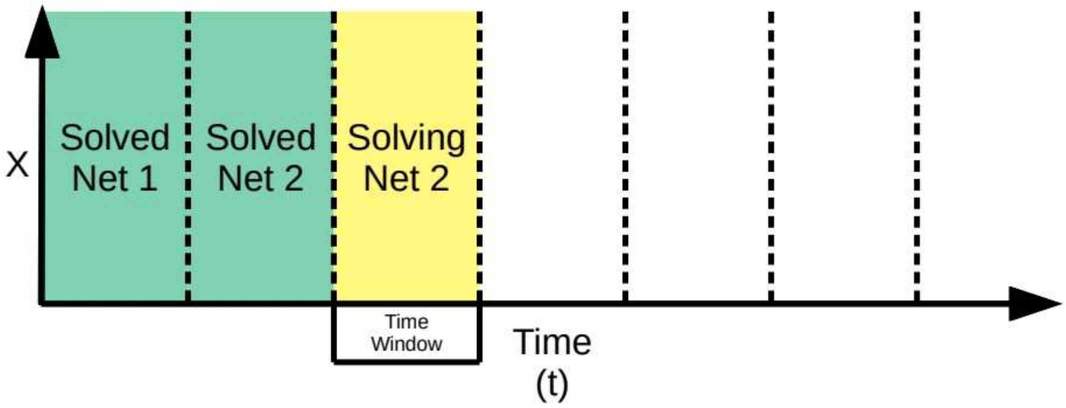

Figure 10 – Marching scheme: each training window corresponds to a subdomain in the t axis to be addressed independently.

A separate neural network is trained to solve each subdomain, as shown in Figure 10, using only the training data within its bounds. An additional continuity constraint is imposed to ensure smooth transitions between solutions across adjacent time domains. This constraint connects the solution of one window to the next, ensuring consistency at the boundaries. Mathematically, this can be expressed as:

$$L_W = \frac{1}{N_{\text{batch}}} \sum^{\text{batch}} \left(u_{\text{model}}(t^i, U_0^i, W_0^i, K^i)^{\text{prevW}} - u_{\text{model}}(t, U_0^i, W_0^i, K^i)^{\text{nextW}} \right)^2 . \tag{3.20}$$

Here, t_i represents the interface point between two subdomains, marking the end of prevW and the start of nextW. Each subdomain has its own neural network model, and the continuity constraint ensures that the predictions at this interface point are identical for both models. This guarantees that the initial conditions of the subsequent window align with the final conditions of the previous window, effectively enforcing consistency between subdomains. This mechanism functions similarly to boundary condition constraints, ensuring smooth transitions across the entire solution

The time-window marching technique enables training smaller models to capture complex temporal dynamics that would otherwise require larger, more expressive networks. During inference, this approach introduces some overhead, as the model must iterate through each subdomain to reach the desired final time. However, this overhead is reduced if intermediate solutions at the end of each subdomain are also of interest, since the iterative process naturally provides them. The main drawbacks arise during training, where separate models must be trained for each subdomain, increasing computational and memory demands. Moreover, as more model parameters are introduced, the dimensionality and complexity of the interface constraint L_W grow, making training more difficult.

With parameterized initial conditions, this concept can be extended: if the model learns the solution for any initial condition within a defined time domain, it can be reused to obtain solutions beyond this initial domain. Specifically, the output from the model at the end of the initial time domain can be utilized as the initial condition for subsequent evaluations. Figure 11 illustrates solution mapping in both large and small time domains, showcasing how mapping complexity increases for larger domains. Consequently, this application of the window technique facilitates fitting smaller models with greater accuracy, preserving the same benefits and drawbacks discussed earlier.

3.4.2 Increasing Cloud Point Density

Another common source of complexity are regions of the parameter space where the solution behavior changes rapidly. This often occurs near unstable equilibrium or in proximity to bifurcations. For example, in the FHN equations, such complexity is evident in the activation threshold influenced by the $(u - \alpha)$ term. The term creates an unstable

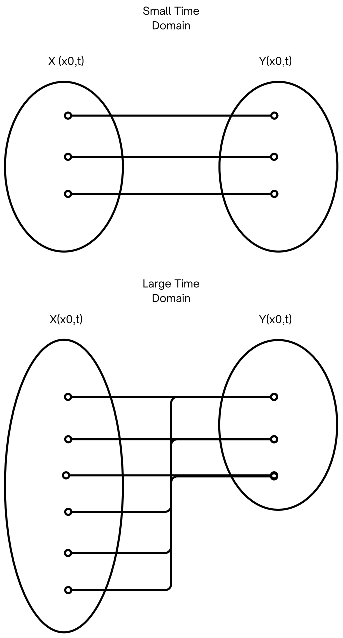

Figure 11 – Subdividing Time Domain technique combined with parametrization of the initial condition. Continuity is ensured by having the neural network learning the solution for any initial condition, and using solution at the end of one subdomain as input to the next. The solution space (image) that the networks needs to learn has the same cardinality, but the mapping is less complex the smaller the window.

equilibrium point around $u = \alpha$ such that solutions with initial conditions U_0 above this threshold, known as supra-threshold, trigger an action potential, while sub-threshold solutions decay toward a steady state as show in 12.

Due to scarce data in these critical regions, training is often difficult, with very few information about transitions regions, with resulting predictions carrying significant error. This issue is mitigated by implementing constraints that evaluate additional points in these transitional regions, forcing the neural network to learn more about these complex regions. This can be achieved with an additional data term with a training set focusing on points only in the critical regions, or by adding a PINN term that only samples points in the regions, Figure 13 showcase this sampling strategy. This strategy helps to accelerates convergence with an overhead cost proportional to the batch sizes of the additional terms. However this cost can be mitigate by having these additional terms with smaller batch sizes. Additionally, this terms can be individually weighted to fine tune training.

3.5 Problems Tackled

The ability of neural networks to model FHN ODEs was tested in three different scenarios: one with no parameters, only time, where the neural network must learn a

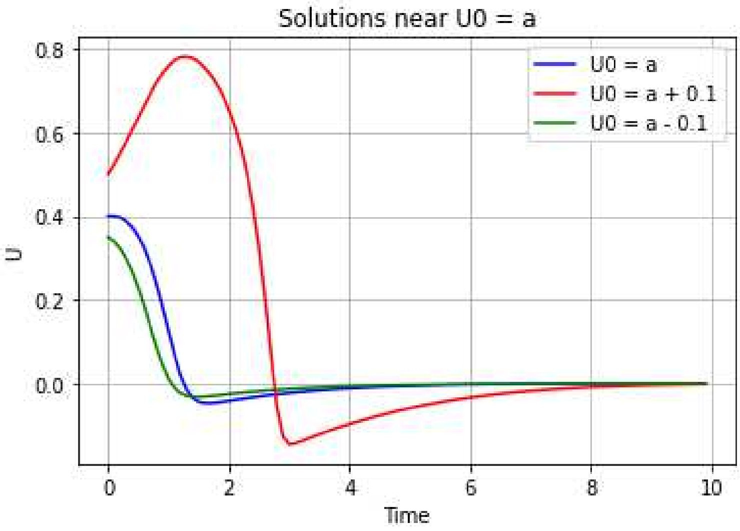

Figure 12 – Unstable equilibrium point caused by the term $(u - \alpha)$, introduces additional complication for model learning, as it causes sharp gradients in the loss function.

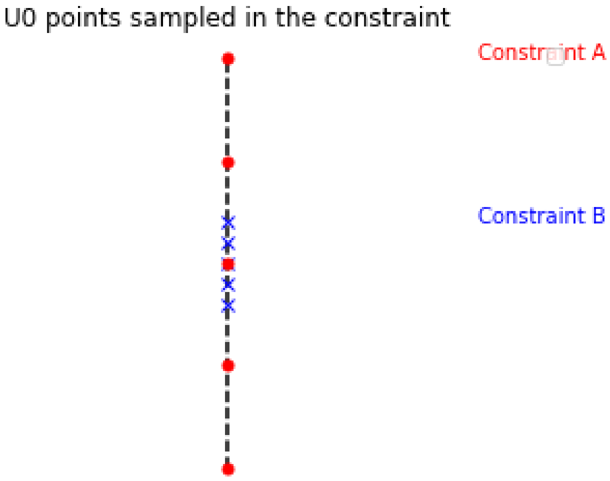

Figure 13 – Samples are taken near the region of interest to produce the new constraints.

single solution along the time axis; one where the initial conditions of the solutions are also parameterized, meaning the network has to learn a complete family of solutions; and one where I_e in Equation (2.6) is also parameterized, changing the solution by increasing the size and frequency of the action potentials (AP).Table 1 summarizes the problems addressed, their designations, and the models employed in each.

3.6 Inference and TensorRT

An important part of this work is to assess the inference performance of the trained models, with the goal of trying to produce models that are more efficient than

Table 1 – The table presents the problems addressed, the models employed in each and their respective forms detailing the solutions being approximated. In problem A, t varies from 0 to 20 time units; in the others from 0 to 50. Initial conditions range from $U_0 \in (0, 1)$ and $W_0 \in (0, 0.18)$, including all values observed in the model's phase plan. The parameter K ranges from 0 to 1, controlling the FHN parameter I_{iapp} from 0.08 to 0.12.

Problem	**Solution Form**	**Description**	**Models Employed**
Problem A	$u(t)$	FHN with only t parameterized, models must learn a single solution along the time axis.	DDNN, PINN
Problem B	$u(t, U_0, W_0)$	FHN with initial conditions parameterized, the model must learn a family of solutions.	DDNN, PINN, ITNN
Problem C	$u(t, U_0, W_0, K)$	FHN with initial conditions and K parameterized, the model must learn families of solutions.	DDNN, PINN, ITNN

the numerical method. To provide a fair comparison under equivalent conditions, model speed was measured using an optimized CUDA code for the numerical solution and a PyTorch inference solution implementing the state-of-the-art SDK TensorRT for the networks. The TensorRT toolkit, developed by NVIDIA, greatly accelerates the inference process by efficiently leveraging tensor cores, specialized arithmetic units designed for matrix multiplication operations, which are present in modern GPUs. In neural network inference, matrix multiplication operations compose a significant part of the cost, since they are required when passing each layer. Therefore, the use of these specialized units, combined with post-training optimizations available in TensorRT performance-oriented model compilation, ensures state-of-the-art inference performance.

The inference process is benchmarked on a machine with an Nvidia GPU RTX 4070 and a Intel i5-12400F. Each model is evaluated sequentially, one at a time, including the numerical solution. Inference performance is measured as the time taken to evaluate a set of samples. For the neural networks, before benchmarking, the best batch size, that is, the amount of data passed at once to the GPU, is determined by trial and error. For the numerical solution, care is taken to ensure only the time taken in the actual solution is measured, excluding memory allocations and copies to and from the GPU. This ensures the best possible comparison for both solutions.

The numerical model was run with $\Delta t = 0.1ms$, the largest, therefore the fastest, discretization that was stable, and the NN models were sampled once per time unit, i.e.,

$\delta t = 1$, the maximum discretization to allow a proper reconstruction of the action potential with its distinguishable features. Figure 14 illustrates the results using these parameters. Figure 15 shows that reducing either parameter further leads to significant numerical errors in the Euler method or a loss of distinguishable features in the neural networks.

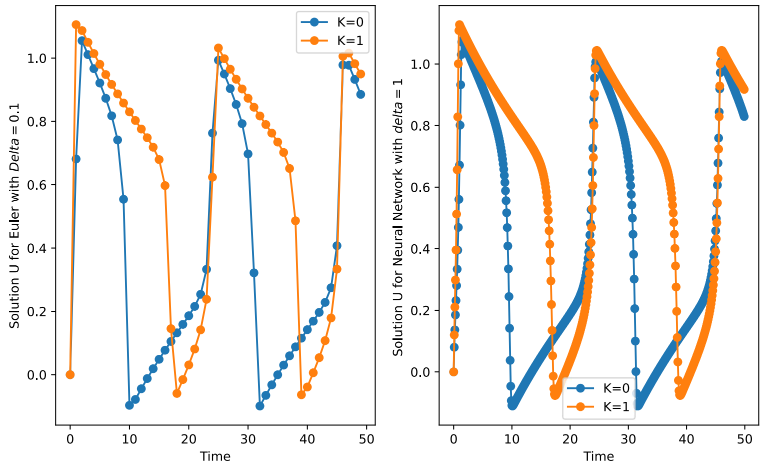

Figure 14 – Solutions with $\delta t = 0.1$ and $\Delta t = 1$, these parameter values provide the fastest possible evaluation for a given parameter set while also preventing numerical error for the Euler method and preserving distinguishable AP features on the NN models.

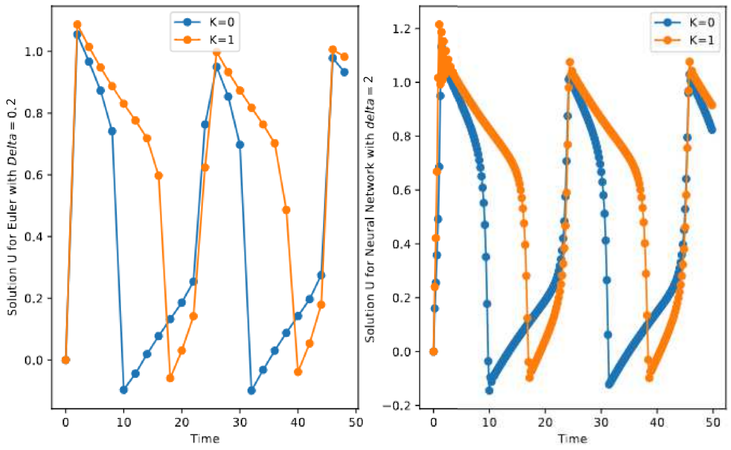

Figure 15 – Solutions showing that previous values of Δt and δt are indeed the fastest possible choice. when Δt is further increased to 0.2 it causes numerical error, while δt, when further increased to 2, causes features such as action potential duration (APD) and max $\frac{dU}{dt}$ to no longer be distinguishable.

4 RESULTS

This chapter presents the main results of the study. First, we discuss the accuracy achieved and the effectiveness of the training techniques applied, organized into sections corresponding to each problem addressed. Next, we analyze inference performance results for the best-performing model in each case.

4.1 Problem A

Problem A addresses a simplified form of the FHN model learning, where the neural network must learn a single solution of the model over time, $u(t)$. The FHN model parameters are set to avoid autopacing, and the initial conditions are set to $U = 0.5$ and $W = 0$, causing a single stimulus at the start, as shown in Figure 3. Using the numerical solution for this initial condition, two datasets were created: one with abundant data that completely describes the solution, and another with scarce data, as shown in Figure 16. The scarce dataset lacks sufficient information to fully characterize an action potential, while the complete set samples the entire time domain at extremely high resolution. Multiple models from the DDNN and PINN classes were trained on each dataset separately, and their final accuracy and training evolution were evaluated.

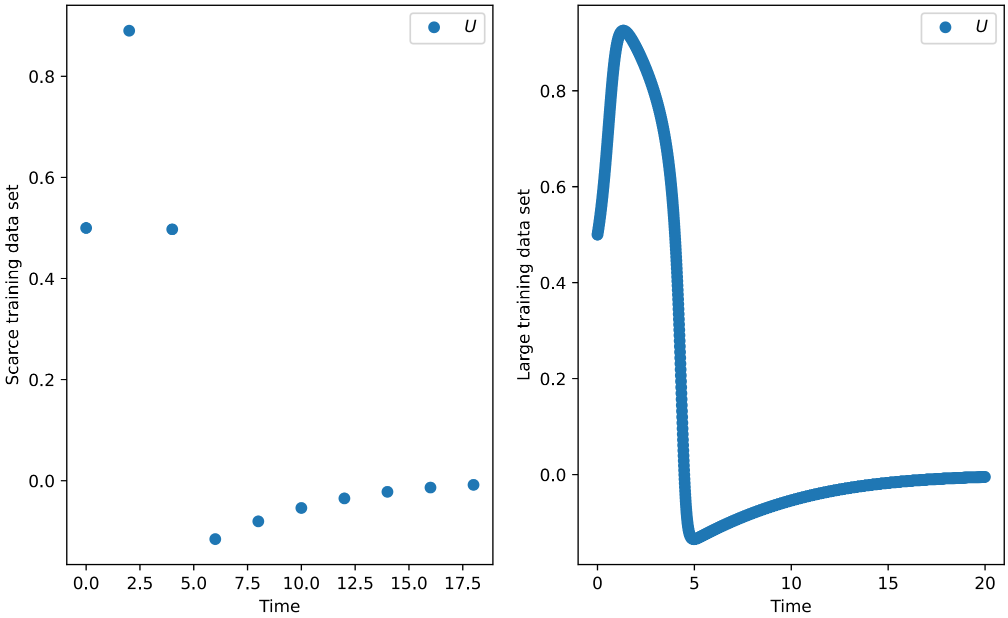

Figure 16 – Datasets used in training. The scarce set contains 10 points along 20 time units, barely able to describe the shape of the action potential. The large training set perfectly defines the action potential in the whole domain, containing 1000 points.

Multiple architectures were tested for both PINN and DDNN models using the grid search algorithm. The same validation set was used across all models and architectures, even for those trained with different datasets, to establish a consistent comparison baseline. Networks with 2 and 3 hidden layers, including the configurations [(nn.SiLU, 16), (nn.Tanh, 8), (nn.SiLU, 32), (nn.SiLU, 16)], were tested. Since the parameter space is one-dimensional along t, small batch sizes were used: 10 for data constraints, 16 for physics constraints, and 2 for boundary conditions. Supplementary material Tables 14 and 15 list trained models.

For the scarce dataset, a clear advantage is observed for PINN models, as shown in Figure 17, where PINNs achieved better accuracy than DDNNs across all architectures. By examining a specific PINN and DDNN model architecture with two (SILU-32) layers, we gain insights into their training behavior. In Figure 19, we observe that the validation loss for the DDNN model stagnates quickly, whereas the PINN model shows significant breakthroughs, followed by a continuous decrease that persists throughout the training. This is reflected in the training error (or loss), shown in Figure 20, where the DDNN exhibits an initial sharp drop, followed by stagnation, while the PINN continues to learn from both physics and data during the entire training process.

The advantage of the PINN model in this case is clearly attributed to the scarcity of the training set. Examining the validation loss evolution in Figure 20, we observe that while the DDNN learns the training points, as indicated by the sharp drop at the beginning, it fails to extrapolate the full solution, performing only simple interpolation, as shown in Figure 18. The significant difference in learning between the PINN and DDNN models with the same architecture indicates that, although the architecture has the expressive capacity to represent the solution space, it cannot reconstruct the solution from the observed data alone. This behavior was consistent across all tested architectures, suggesting a broader pattern. Consequently, in this scenario, PINN models were able to generate synthetic data to augment the training set and accurately recreate the solution with relatively low error.

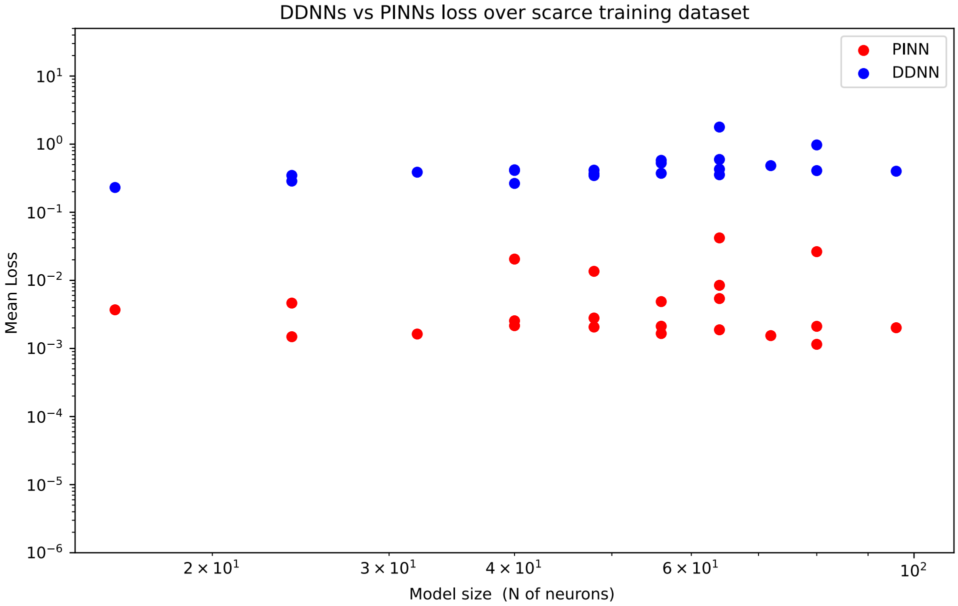

Figure 17 – PINNs and DDNNs trained using the scarce dataset. Results clearly show that PINN models were able to achieve much better accuracy (measured in mean error of the validation set) in all cases, for both smaller and larger models. A T-test value of 7.42 indicates a substantial difference between the groups.

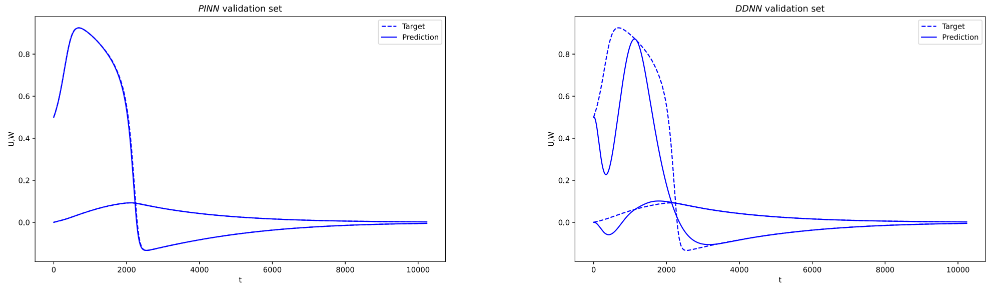

Figure 18 – Validation set evaluation at the end of training for the two (SILU-32) layer PINN and DDNN models trainned with the scarce set. The PINN is able to learn the true solution, whereas the DDNN merely interpolates training data.

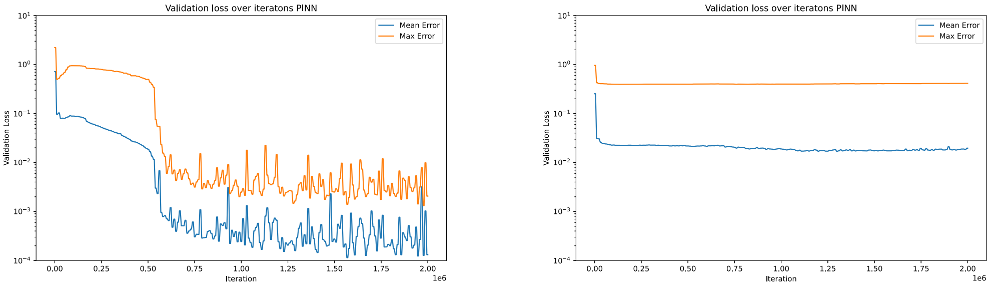

Figure 19 – Validation loss evolution for PINN (left) and DDNN (right) models, showing continuous decay of the loss throughout training, and an early stagnation respectively.

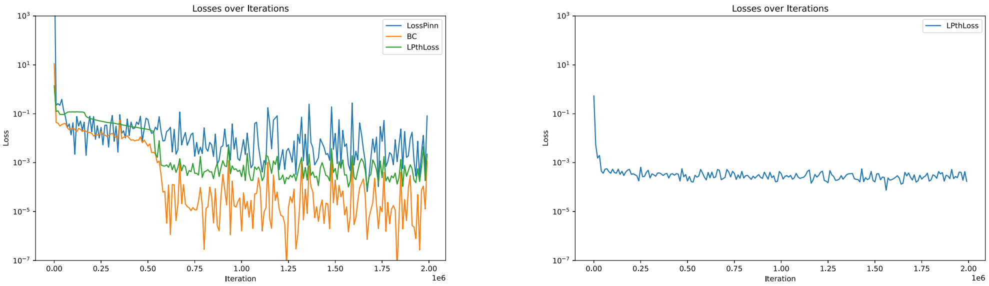

Figure 20 – Training loss evolution. For the PINN model (left), big breakthroughs (sudden drops in the loss) occur early, followed by a continuous decay of the loss. For DDNN (right), the loss quickly becomes very low and then stagnates.

For the more complete training set, i.e. with fine sampling, as shown in Figure 21, the differences between the PINN and DDNN models are not statistically significant. The observed variation is primarily due to the random nature of the training, with a p-value of 0.74 indicating high probability of any observed effect be caused by the stochasticity of the process. This suggests that there is no clear advantage of the PINN constraint in this case. Thus, with sufficient data, DDNNs can also capture the entire solution space. However, it is important to note that in this one-dimensional space, generating enough data to accurately represent the solution is computationally inexpensive. In contrast, for more complex problems with high-dimensional parametrization, this process may become extremely costly. In such cases, as demonstrated by this example, PINNs can offer significant advantages.

In both cases, training with the scarce and the complete datasets, it is notable that an increase in model size does not correlate with an increase in accuracy. This suggests that the solution space is effectively captured by the smaller networks, and further increasing their expressive power offers no significant improvement. In this case, the solution space is simple, consisting of two continuous functions of a single variable, with no discontinuities or sharp gradients, making the smaller models sufficient. Interestingly, the training data was generated using the Euler numerical scheme with $\Delta t = 0.01$, resulting in a global error of the order of 0.01. It is possible that models trained with data generated using a smaller Δt could achieve higher accuracy by learning from less noisy data, potentially benefiting from an increase in model size.

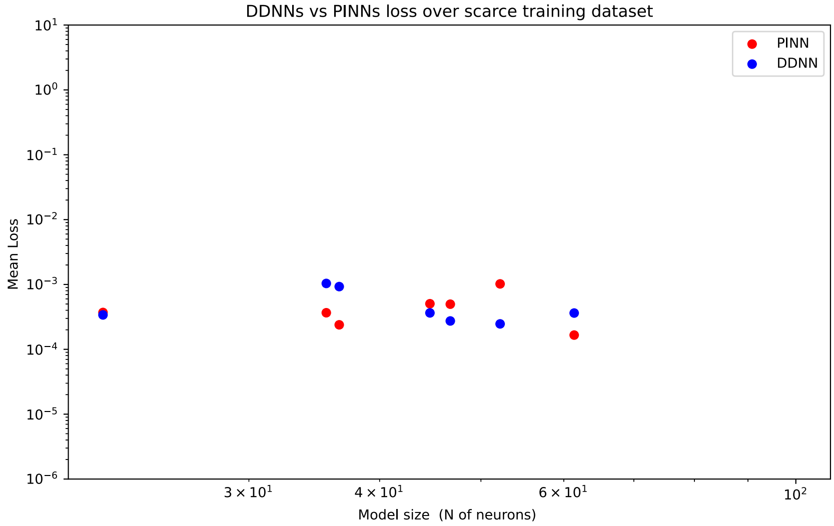

Figure 21 – PINNs and DDNNs trained using the complete dataset. PINNs and DDNNs models with the same architecture (along the same x-axis) show no significant difference in accuracy performance. A T-test value of 0.33 indicates no significant difference between the groups.

4.2 Problem B

Problem B consists of learning a family of solutions for the FHN model, where the initial conditions that define the solution are parameters of the model. The model takes the form $u(t, U_0, W_0)$ and must learn the solution along the time axis for various initial conditions of U and W. This task is more challenging than Problem A, as it involves learning a larger solution space. Two cases are considered for Problem B: one with $I_{app} = 0$, resulting in no continuous stimulation and a single action potential generation, and one with $I_{app} = 0.1$, which produces autopacing and cyclic solutions. Figure 3 illustrates the behavior of both models.

Each model presents unique learning challenges: for the single-stimulus model, the initial conditions U_0 and W_0 determine whether or not an AP occurs, i.e., if U_0 is above the threshold and W_0 is not in the recovery phase. For the cyclic model, the initial condition may either lie within the model's limit cycle, shown in Figure 4, where the cycle resumes from the initial condition, or outside it, where a small perturbation occurs at the beginning before returning to the cycle. This presents a particular challenge because, for each arbitrary initial condition, the model must learn the path to return to the cycle.

The two FHN-parameterized models are referred to as Problem B_S and B_C for the single and cyclic cases, respectively. Both datasets are generated by sampling the parameter space for U_0 and W_0, with no I_{app} for Problem B_S and $I_{app} = 0.1$ for Problem B_C. Larger architectures than those used in Problem A were considered for both PINN and DDNN models, and each was trained for 1 million iterations using the Adam optimization method. Specifically, for Problem B_C, an iterative neural network model (ITNN) with smaller architectures was also tested. Additionally, the increased cloud point density technique was employed for Problem B_S, targeting a region of rapid transition (i.e., unstable equilibrium) caused by the threshold behavior around $U = a$. Both DDNN and PINN models can incorporate an additional term to enforce the ODE constraint in this critical region. When they do, they are referred to as DDNNa and PINNa, respectively.

4.2.1 Single Stimulus

The first set of results investigates the effectiveness of physics-informed learning for Problem B_S, as well as the impact of varying the training batch size, two factors that influences the amount of information used on each training iteration. Four models were tested: a pair of purely data-driven (DDNN) and physics-informed (PINN) models trained using points homogeneously sampled from the entire domain, and a pair that additionally incorporates a physics constraint limited to the unstable equilibrium region (referred to as PINNa and DDNNa), which implements the increased cloud point density technique. All models utilized the same two-layer architecture (SiLU-32), and each was evaluated with five different batch sizes (affecting all constraints): 32, 64, 128, 512, and 1536. A dataset containing 10,000 unique solutions homogeneously sampled from the domain was assembled, with half of the points used for the data constraint present in all models and the other half used for validation.

As shown in Figure 22, increasing the batch size has no significant effect on final accuracy for any of the models, with only a slight trend toward improved accuracy as the batch size grows. However, this improvement is minimal and may be attributed to the stochastic nature of the training process. In the most extreme case, for the PINNa model, optimizing the batch size resulted in a 40% to 60% improvement in accuracy. An ANOVA analysis was performed to assess the effect of each parameter on training, fitting the following explanatory model:

$$Q(\text{Mean err}) \approx C(\texttt{iccs}) \cdot \texttt{bs} + C(\texttt{pinn}) \cdot \texttt{bs}, \tag{4.1}$$

here, the mean error is expressed as a function of `bs` (batch size), along with `iccs` and `pinn`, which are categorical variables representing the use of the increased cloud point density physics constraint and the general physics constraint, respectively. The results are shown in Table 2. The data indicates that the small changes in model accuracy can be equally attributed to both the model class (i.e., the parameters `pinn` and `iccs`) and the batch size, as indicated by the F-value. The fitted coefficients show a small positive effect (negative coefficient, decreasing the error) of increasing the batch size, and a negative effect (positive coefficient, increasing the error) for both `pinn` and `iccs` models. Although the negative coefficients are not large enough to indicate a significant trend, this suggests no clear benefit to using either physics constraint for this problem and architecture. Even for the maximum error, a metric that better captures performance in the unstable equilibrium region enforced by the extra constraints, the results were similar, showing no significant effect.

This is particularly noteworthy because the use of either constraint significantly increases iteration cost and, consequently, overall training time, as shown in Figure 23. The same ANOVA analysis, as in equation 4.1, but with iteration time as the response variable (see Table 3), reveals that the `pinn` and `iccs` variables have a far greater impact than batch size. This indicates that the computational cost of incorporating additional constraints, and performing backpropagation through more terms in the loss function, is substantially higher than merely increasing the batch size for loss evaluation. Therefore, in these cases, the physics constraints not only fail to improve training but also hinder it by increasing computational expense.

Moreover, the results show that using a full PINN model is significantly more costly than applying a localized physics constraint, as in the DDNNa model. This difference stems from the computational burden of the added constraints: in the PINN model, two new constraints—one for the interior and one for the boundary—are evaluated at each iteration, whereas in DDNNa models, only a single localized constraint is introduced. This effect is even more pronounced in the PINNa model, which incorporates all three constraints, further increasing the computational cost.

The results shown in Figures 24 and 25 generalize these findings across different architectures and activation functions. Models with two-layer architectures, using permutations of the set [(nn.SiLU,8),(nn.SiLU,32),(nn.SiLU,64),(nn.Tanh,32), (nn.Tanh,64), (nn.ELU,8)], were trained with the same batch sizes as before. The trend observed for the two-layer (SiLU-32) architecture, where DDNN models outperform their PINN counterparts in most cases, remains consistent across all tested architectures. Furthermore, increasing batch size beyond a certain minimum viable limit does not yield noticeable accuracy improvements for any of the models. However, for larger models, this limit is slightly higher. A batch size of 128 was found to be sufficient even for larger architectures, and it was thus used in subsequent experiments. Based on these findings, the focus shifts to DDNN models for exploring a broader range of architectures.

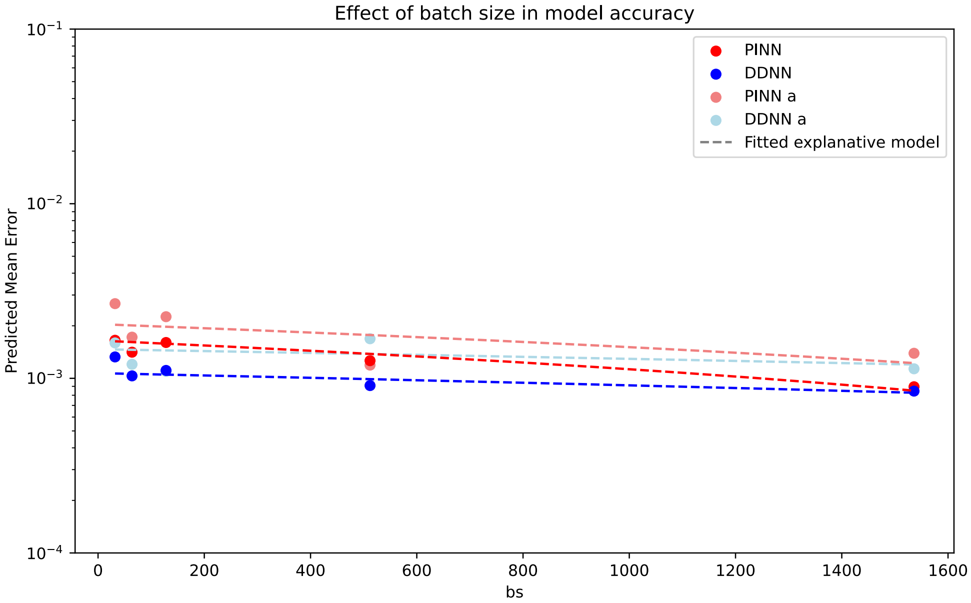

Figure 22 – Model accuracy for different model types, with increasing training batch size. Results show a slight positive effect of increasing batch size, with data-driven models achieving better accuracy in every case.

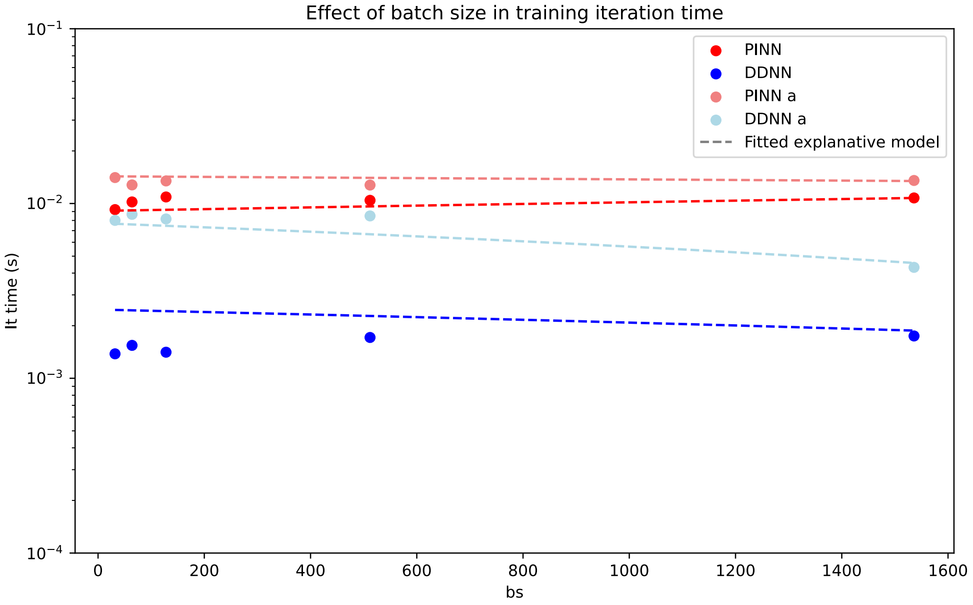

Figure 23 – Models average iteration cost in terms of time spent on loss evaluation, backpropagation, and weight updates. Models of four different classes with various training batch sizes are considered. Results show that physics-informed models are consistently more expensive and that increasing batch size does not significantly increase cost.

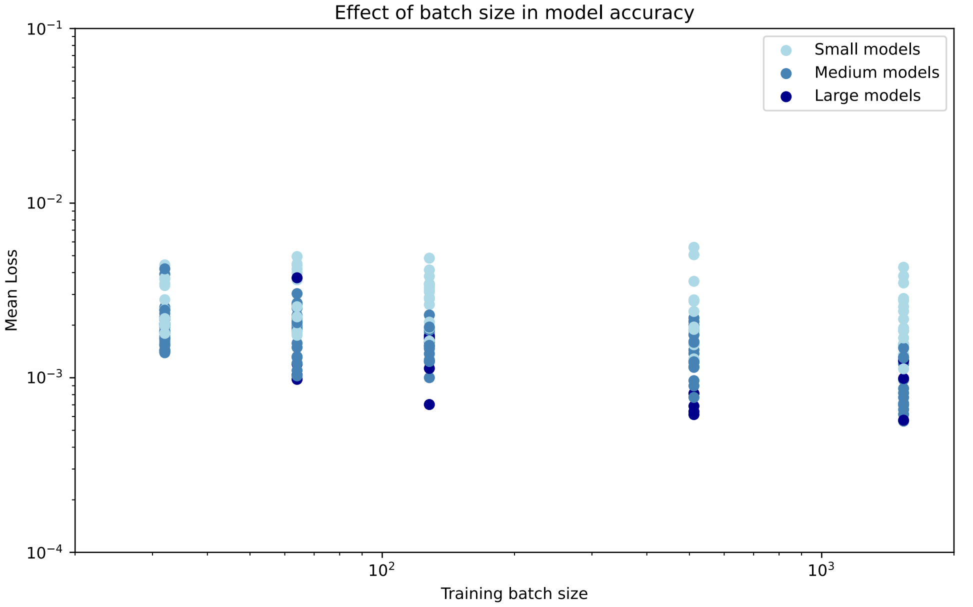

Figure 24 – Model accuracy for different architectures, organized by size (small if less than 32 neurons, large if more than 100, and medium otherwise), and trained with different batch sizes. Larger models require a minimum batch size to achieve maximum efficiency, though increasing it further has minimal impact.

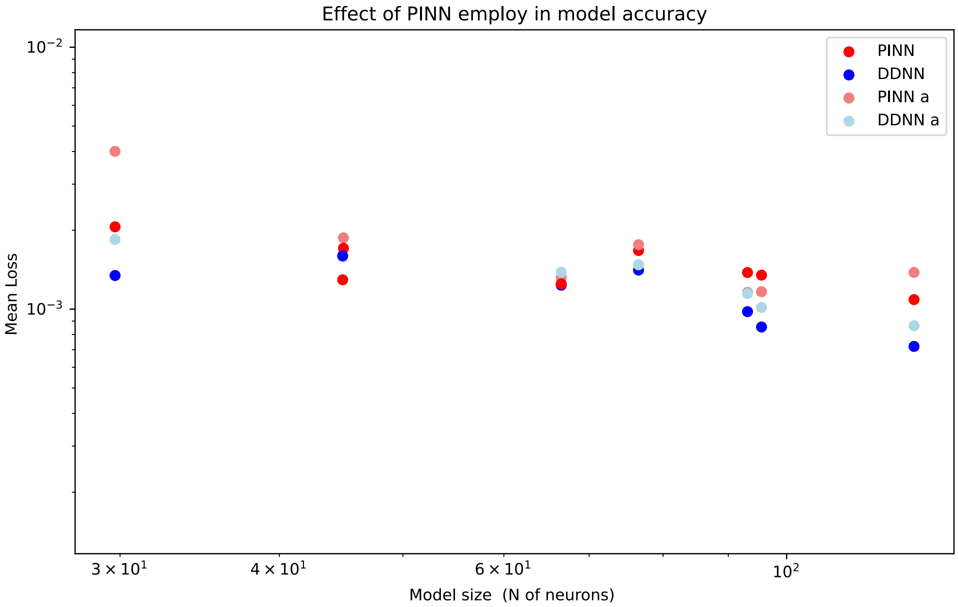

Figure 25 – Model accuracy for different classes and architectures, organized by neuron count (y-axis). Data-driven models consistently outperform physics-informed counterparts in this case.

Table 2 – Summary of ANOVA results for the trained model accuracy in relation to model category (`iccs` and `pinn`) and model training batch size (`bs`). Results show an effect of all three variables in the accuracy, with a significant F value and Sum of squares, than account for the variability caused by each factor. Both the employ of the physics constraints in the whole domain (`pinn`) and in the specified region (`iccs`) have an negative effect on model accuracy (positive coefficient, increasing the error) while model batch size had smaller positive effect. A low P-value ($p<0.05$) indicates low chance of results observed being due to sthocasticity in tranning.

Parameter	Fitted Coef	Sum Sq	F	p-value
C(iccs)	3.97×10^{-4}	7.61×10^{-7}	7.68	1.50×10^{-2}
C(pinn)	5.76×10^{-4}	8.49×10^{-7}	8.57	1.10×10^{-2}
bs	-1.59×10^{-7}	7.75×10^{-7}	7.81	1.43×10^{-2}
C(iccs):bs	-1.61×10^{-8}	4.20×10^{-10}	0.004	9.49×10^{-1}
bs:C(pinn)	-3.60×10^{-7}	2.08×10^{-7}	2.10	1.69×10^{-1}
Residual		1.39×10^{-6}		

Table 3 – Summary of ANOVA results for the explicative model for iteration time in relation to the model category (`iccs` and `pinn`) and model training batch size (`bs`). The results indicate that both the employ of physics constraints across the entire domain (`pinn`) and within the specified region (`iccs`) significantly increase the iteration cost (high F-value and sum of squares paired with positive coefficient), with `pinn` being remarkably more impactfull. In contrast, the batch size (`bs`) exhibits a minimal effect.

Parameter	Fitted Coef	Sum Sq	F	p-value
C(iccs)	5.25×10^{-3}	1.01×10^{-4}	79.70	3.74×10^{-7}
C(pinn)	6.60×10^{-3}	2.64×10^{-4}	209.00	8.36×10^{-10}
bs	-3.90×10^{-7}	1.47×10^{-6}	1.16	2.99×10^{-1}
C(iccs):bs	-1.66×10^{-6}	4.46×10^{-6}	3.52	8.17×10^{-2}
bs:C(pinn)	1.49×10^{-6}	3.56×10^{-6}	2.81	1.16×10^{-1}
Residual		1.77×10^{-5}		

With DDNN models demonstrating superior performance in the first experiment, we conducted a new set of experiments aimed at optimizing these models by exploring a wide variety of architectures. This involved training multiple DDNN model configurations for the same problem, utilizing the same training and validation sets as in the previous experiment. The architectures were generated using permutations of the following layers: [(nn.SiLU, 16), (nn.Tanh, 16), (nn.Tanh, 32), (nn.SiLU, 64), (nn.Tanh, 64)], with configurations consisting of either 2 or 3 layers, allowing for repetitions. These architectures were classified according to their shape: rectangle (all layers of the same size), funnel (layers progressively smaller), bottleneck (layers progressively larger), bowtie (three layers with a smaller middle layer), and diamond (three layers with a larger middle layer). Such shapes correspond to common neural network functions, where funnel and bottleneck structures are frequently used for feature extraction and generative models, while the bowtie shape is typically employed in autoencoders for dimensionality reduction.

In total, over 200 models with either two or three layers were trained for 1 million iterations each, with results presented in Figure 26. The best performance, measured by mean error on the training set, was achieved with the rectangular architecture $(\text{SiLU}, 32) - (\text{Tanh}, 32) - (\text{SiLU}, 32)$, yielding a mean error of 0.00038 with 96 neurons. Figure 28 illustrates the models' accuracy across a slice of the solution space, demonstrating low error throughout most of the domain, except for a high-error region near the unstable equilibrium point at $U_0 = a$, as detailed in Figure 27. This pattern was consistent across all models, with none able to adequately learn this complex region. Other three-layer architectures also achieved comparable accuracy, remaining within the noise range introduced by the stochastic nature of the training process. Table 16 (supplementary material) lists the best-performing models, while Table 4 highlights notable architectures.

Models with two layers demonstrated significantly poorer performance, with the best two-layer architecture, $(\text{Tanh}, 32) - (\text{SiLU}, 64) - (\text{Tanh}, 32)$, achieving a mean error of 0.00085. This error is notably larger than that of the worst-performing three-layer models with the same neuron count, indicating a limitation in the expressive power of two-layer architectures, which would require a substantially larger number of neurons to achieve comparable performance. As highlighted in (LU et al., 2017), model depth generally plays a critical role in accuracy, particularly for smaller models. However, this increased depth comes at the expense of efficiency; the training iterations for models with the same neuron count are approximately twice as costly when comparing two-layer to three-layer architectures, which also adversely affects inference performance, as will be discussed later.

On average, bowtie architectures outperformed other configurations with the same neuron count, although several exceptions existed, including some rectangular architectures that performed better. The architecture type was particularly influential for smaller neuron counts, where greater performance variability among different groups was observed. This finding is especially relevant for achieving efficient models, particularly in small networks with low error rates. Notably, the model $(\text{SiLU}, 8) - (\text{Tanh}, 32) - (\text{SiLU}, 8)$ achieved a mean error of 0.00050 with only 48 neurons—half the size of the best-performing architecture—while incurring less than a 15% increase in error. This model belongs to a set of bowtie architectures that consistently outperformed all other models of similar size, as illustrated in Table 17 (supplementary material), highlighting the significance of optimizing model shape. In contrast, for larger models, the best architectures across different shapes exhibited similar performance, suggesting that the influence of architecture shape diminishes in favor of optimizing the arrangement of activation functions.

This effect can be analyzed by evaluating the error across model architecture categories, with intra-group variation representing differences between models with the same architecture (due to different activation function sequences) and extra-group variation reflecting differences caused by architecture shapes. These variations serve as proxies for the influence of activation functions and architectural design. Models were categorized into small (fewer than 120 neurons) and large (120 neurons or more). For smaller models the extra-group variance (1.46×10^{-5}) in the mean error is significantly higher than the intra-group variance (1.06×10^{-6}), indicating that architecture shape plays a much larger role in performance. In contrast, for larger models the extra-group variance (3.17×10^{-7}) is much closer to the intra-group variance (2.08×10^{-7}), suggesting that architecture shape and activation function choice contribute more equally to performance as model size increases.

The ANOVA results, shown in Tables 5 and 6, further support the trend observed with a variance analysis using the explanatory model:

$$Q(\text{Mean err}) \approx C(\texttt{neuron}) \cdot \texttt{shape}, \tag{4.2}$$

where the mean error is defined as a function of `neuron` count and `shape`, a categorical variable representing the architecture shape. Results show that for smaller models, architecture shape has a strong effect, as reflected by the F-value for the shape variable of 19.6. However, for larger models, this influence diminishes, with the F-value dropping to 4.00, indicating a reduced impact of architecture shape. Neuron count remains a consistently significant factor across both groups, with F-values of 42.8 and 34.5 for smaller and larger models, respectively. Thus the effect of architecture optimization is shown to be particularly significant in smaller models. As models grow larger, the influence of architecture shape becomes comparable to the effect of rearranging activation functions, with neuron count emerging as the dominant factor in performance.

Table 4 – Summary of notable models found for Problem B_S.

Layers	**Mean err**	**Neuron count**	**Class**	**Note**
(ELU-8, SILU-8)	6.42e-03	16	rectangle	Smallest model < 0.01
(TANH-32, ELU-8, SILU-8)	8.94e-04	48	funnel	Smallest model < 0.001
(SILU-8, TANH-32, SILU-8)	5.06e-04	48	bowtie	Best model < 64
(SILU-32, TANH-32, SILU-32)	3.88e-04	96	rectangle	Best model < 128
(TANH-32, SILU-32)	9.62e-04	64	rectangle	Best two-layer
(SILU-32, TANH-32, SILU-32)	3.88e-04	96	rectangle	Best model

Table 5 – Summary of ANOVA results for the explicative model mean error in relation to class and `neuron` count, for models with less than 120 neurons. Both the model category and the neuron count are very significant with high F value and Sum of Squares.

Source	**Sum Sq**	**Mean Sq**	**F-value**	**P-value**
C(class)	5.87×10^{-5}	1.47×10^{-5}	19.6	2.80×10^{-13}
neuron	3.20×10^{-5}	3.20×10^{-5}	42.8	6.88×10^{-10}
C(class):neuron	2.73×10^{-5}	6.83×10^{-6}	9.14	1.05×10^{-6}

Table 6 – Summary of ANOVA results for the explicative model for mean error in relation to class `shape` and `neuron` count for models with more than 120 neurons. The model category remains significant, F-value of 4, but its impact is notably reduced (by a factor of 20) compared to smaller models.

Source	**Sum Sq**	**Mean Sq**	**F-value**	**P-value**
C(class)	1.27×10^{-6}	3.18×10^{-7}	4.00	0.0057
neuron	2.74×10^{-6}	2.74×10^{-6}	34.5	1.49×10^{-7}
C(class):neuron	6.76×10^{-6}	1.69×10^{-6}	21.3	2.30×10^{-11}

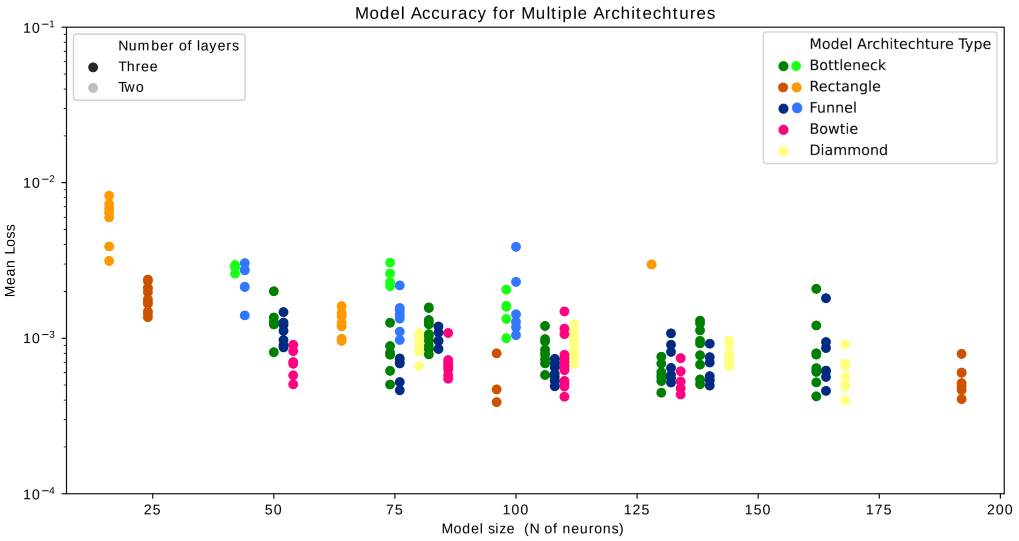

Figure 26 – Model accuracy over the validation set (mean error) after training for Problem B_S. Results for all tested model architectures are shown. Models along the same y-axis share the same neuron count but have varying architecture, with different sizes and activation functions for each layer, and have 3 layer (darker shading) or 2 (lighter shading). For ease of visualization, each category has been slightly dislocated on the x-axis. Results show that Bowtie architectures have a notable advantage for smaller neuron counts, being able to produce the most efficient models in terms of size and accuracy. Increasing neuron count did not lower model average error significantly after around 5×10^{-4} but lessened the difference between shapes.

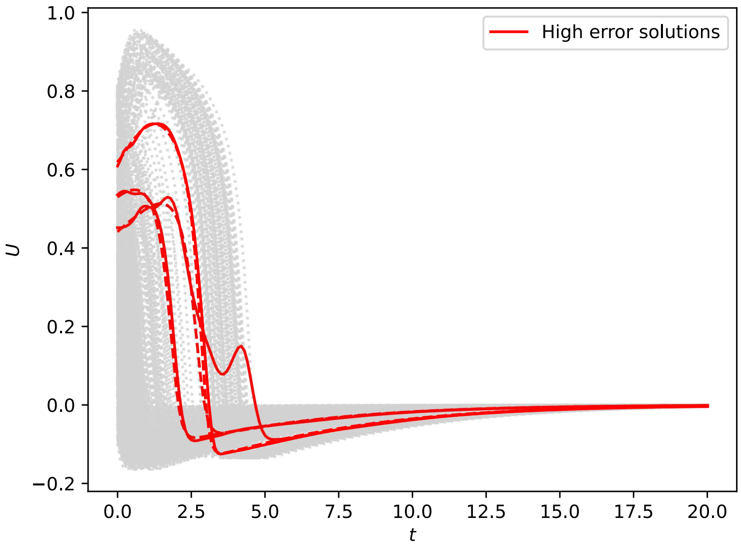

Figure 27 – Model (SILU'>-32), (TANH'>-32), (SILU'>-32) prediction for Problem B_S over each solution in the validation set, the network prediction and actual solution (doted line) are show. High solutions error, those containing points with at least 80% of the max error, are highlighted. High error solution all start at the bifurcation region.

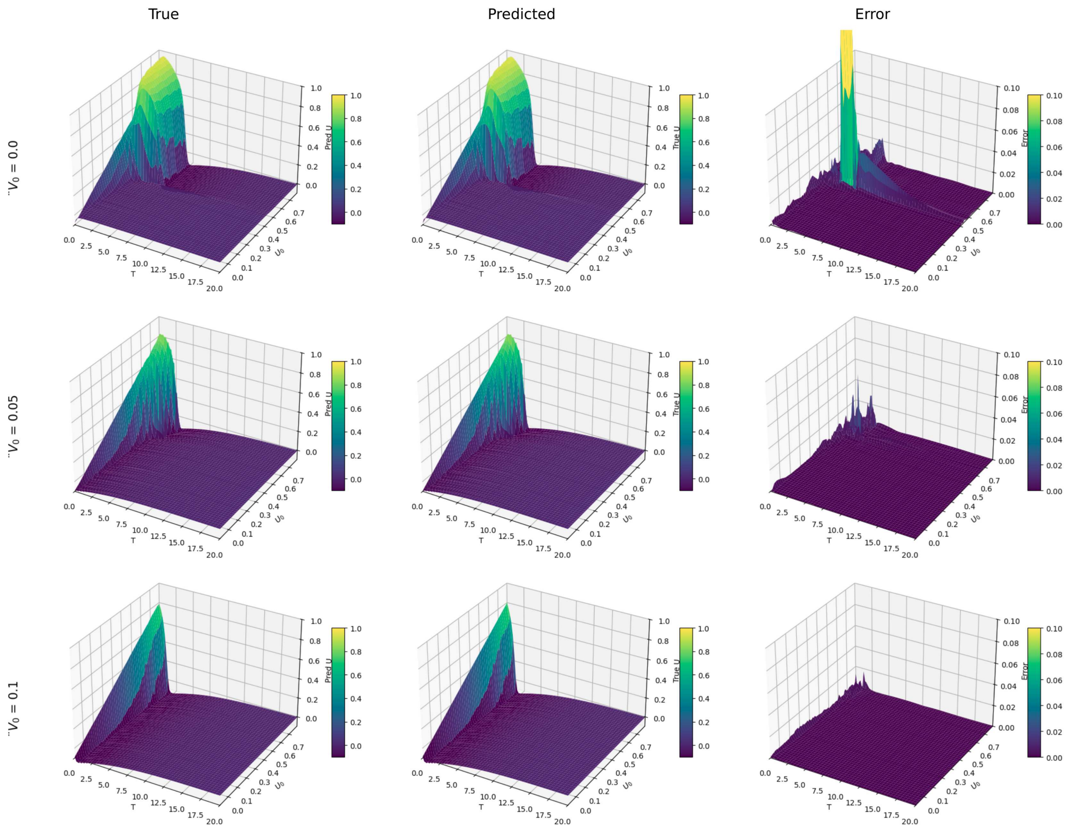

Figure 28 – Model (SILU'>-32), (TANH'>-32), (SILU'>-32) accuracy for Problem B_S over three cut of the solution space. Each cut show solutions along the t axis, for varying initial condition U_0 in the x axis, with the initial condition V_0 fixed in each cut. Z axis show the accuracy in each point. Results show low error along most of the domain with a high error region arround the bifurcation.

4.2.2 Autopacing

Problem B_C involves approximating the periodic solution of an ODE with parameterized initial conditions that may perturb or shift the solution. The model must capture the entire phase plane (as seen in Figure 5) and the trajectory that results from any perturbations to the limit cycle.

To demonstrate the effectiveness of the Time Domain Splitting Technique, models were tasked with learning both a large time domain—spanning roughly three oscillations—and a small time domain, approximately one-third the size of an oscillation. Two training sets of 10,000 samples were generated by sampling the parameter space: one containing solutions up to $t = 5$, and another up to $t = 50$. A variety of model architectures were tested using combinations of layers [(nn.SiLU, 8), (nn.SiLU, 32), (nn.SiLU, 64), (nn.Tanh, 32), (nn.Tanh, 64), (nn.ELU, 8)], with 2, 3, and 4-layer configurations. As shown in Table 7, models trained on the time domain consistently performed better (Figure 29). This demonstrates the potential of the technique, as it allows models to focus on learning simpler mappings over small time domains, enabling models with fewer neurons (and thus lower computational cost) to meet a given accuracy threshold.

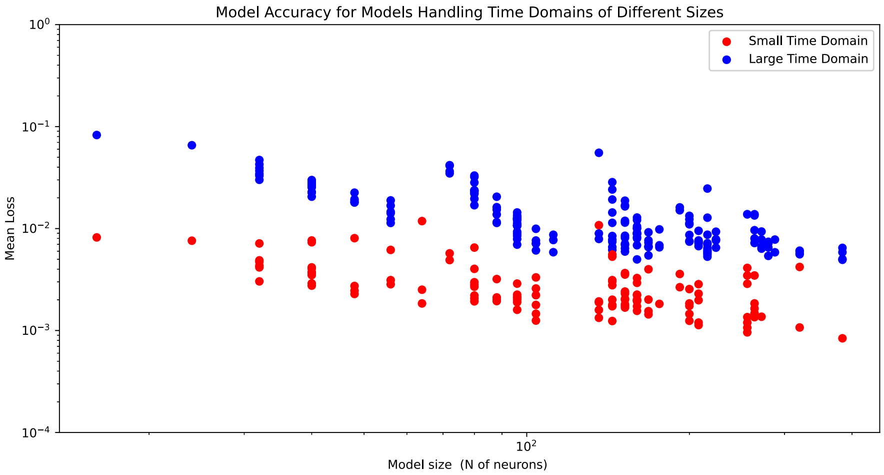

Figure 29 – Model accuracy over the validation set (mean error) for Problem B_C in the small and the larger time domains. Models along the same y axis share the same neuron count but have varying architecture, with different sizes and activation functions for each layer. Results show that all models perform much better when dealing with the smaller time domain. Notably, for both the larger and small time domains, the mean error of all the models was larger for Problem B_C than for Problem B_S.

Next, we aim to identify the smallest possible model capable of solving Problem B_C by exploring a larger pool of models trained on a small time domains of $t = 5$. Models were generated with 2 and 3 layers using combinations of the following layers: [(nn.SiLU, 8), (nn.SiLU, 16), (nn.Tanh, 32), (SIN, 16), (nn.ELU, 8)]. Notably, the periodic activation function, SIN, was included in the layer combinations. The same training

Table 7 – Best, worst, and average model performance measured in mean error over the validation set for large and small time domain cases. Results clearly show better results for the smaller domain.

Time Domain size	Mean ± Std	Best	Worst
50	$1.44 \times 10^{-2} \pm 1.14 \times 10^{-2}$	4.94×10^{-3}	8.27×10^{-2}
5	$2.96 \times 10^{-3} \pm 1.93 \times 10^{-3}$	8.39×10^{-4}	1.18×10^{-1}

protocol, including the Adams optimization method and the training parameters used in previous cases, was followed. Results shown in Figure 30 indicate that training for the small time domains consistently yielded an error of less than 10^{-2}, even with smaller models. Remarkably, models with a neuron count as low as 16, utilizing two layers of size 8, achieved this level of accuracy. Table 8 summarizes the results of the model optimization, highlighting the most accurate model along with some notable models that exhibited good accuracy relative to their size.

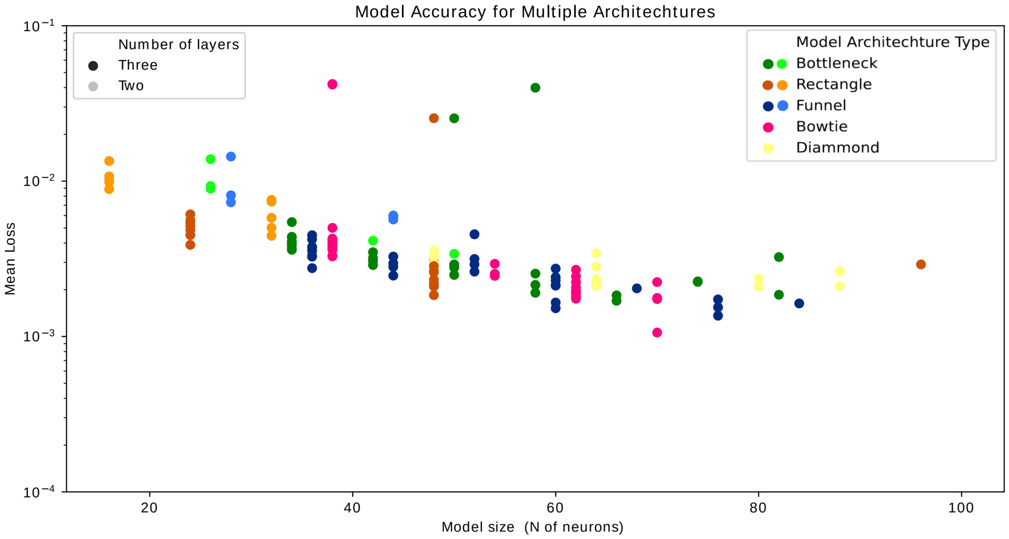

Figure 30 – Model accuracy over the validation set (mean error) after training end for Problem B_C with the small time domain. Results for all tested models are show. Models along the same y axis share the same neuron count, but have varying architecture. Results shows great variation in performance of models with the same neuron count, even within the same architecture.

While the mean error for the best models was satisfactorily low, a high error region phenomenon similar to that of Problem B_S was observed. In this case, the high error region is linked to specific ranges of initial conditions for U_0 and W_0, which introduce significant noise at the beginning of the solution. This high error concentration occurs near

Table 8 – Summary of selected models based on their mean error and neuron count for Problem B_C. The table includes the smallest models under various error thresholds, the best performing models with neuron counts less than 32, 64, the overall best model, and the best two-layer model.

Layers	**Mean err**	**Neuron count**	**Class**	**Note**
(SILU-8, ELU-8)	9.90e-03	16	rectangle	Smallest model $<$ 0.01
(SIN-16, SILU-16, SIN-16)	1.85e-03	48	rectangle	Smallest model $<$ 0.002
(SIN-16, TANH-32, SIN-16)	1.06e-03	64	diamond	Best model $<$ 64
(SIN-16, SILU-8, ELU-8)	2.75e-03	32	funnel	Best model $<$ 32
(TANH-32, TANH-32)	2.17e-03	64	rectangle	Best two-layer model
(SIN-16, TANH-32, SIN-16)	1.06e-03	64	diamond	Best model

an unstable equilibrium region this region, where small variations in the initial condition can lead to substantial changes in the resulting solution, resulting in a chaotic behavior characterized by minor oscillations outside the limit cycle at the start and sharp gradients upon returning to the regular limit cycle. This chaotic behavior poses significant challenges for the network's learning process. Figure 31 displays some of these solutions. Additionally, Figures 33 and 32 illustrate the high error region in various cuts of the parameter space for both the large and small time domain models. The results indicate that while employing a smaller time domains reduces the average error, it does not significantly mitigate the error in the high error region.

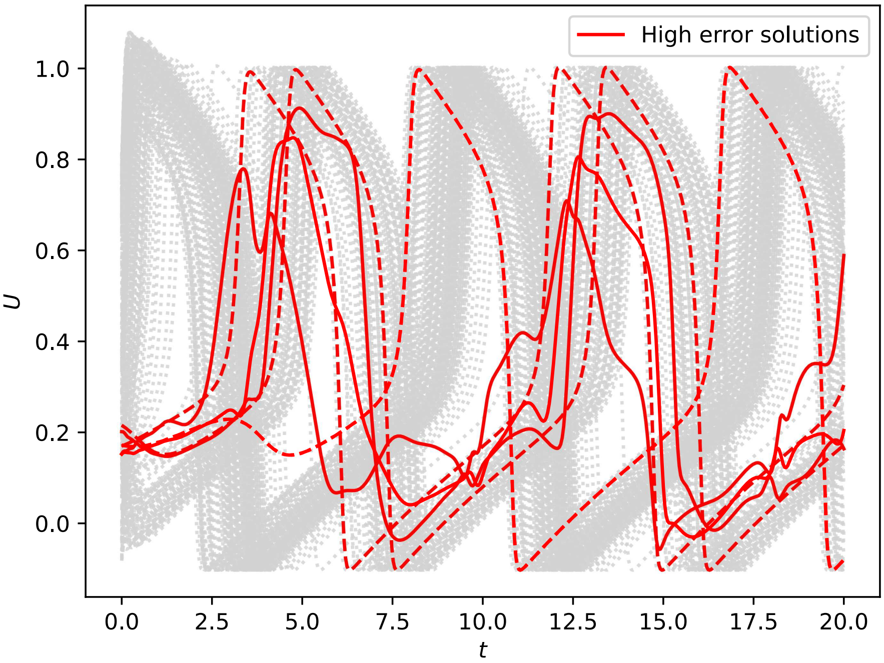

Figure 31 – Model predictions for Problem B_S using the architecture (SILU'>-32), (TANH'>-32), (SILU'>-32). The network's predictions are plotted alongside the actual solutions (dotted line) for each sample in the validation set. Solutions with high prediction errors, defined as containing points with at least 80% of the maximum error, are highlighted. Notably, these high error solutions initiate from an unstable equilibrium region where the derivative is of low magnitude.

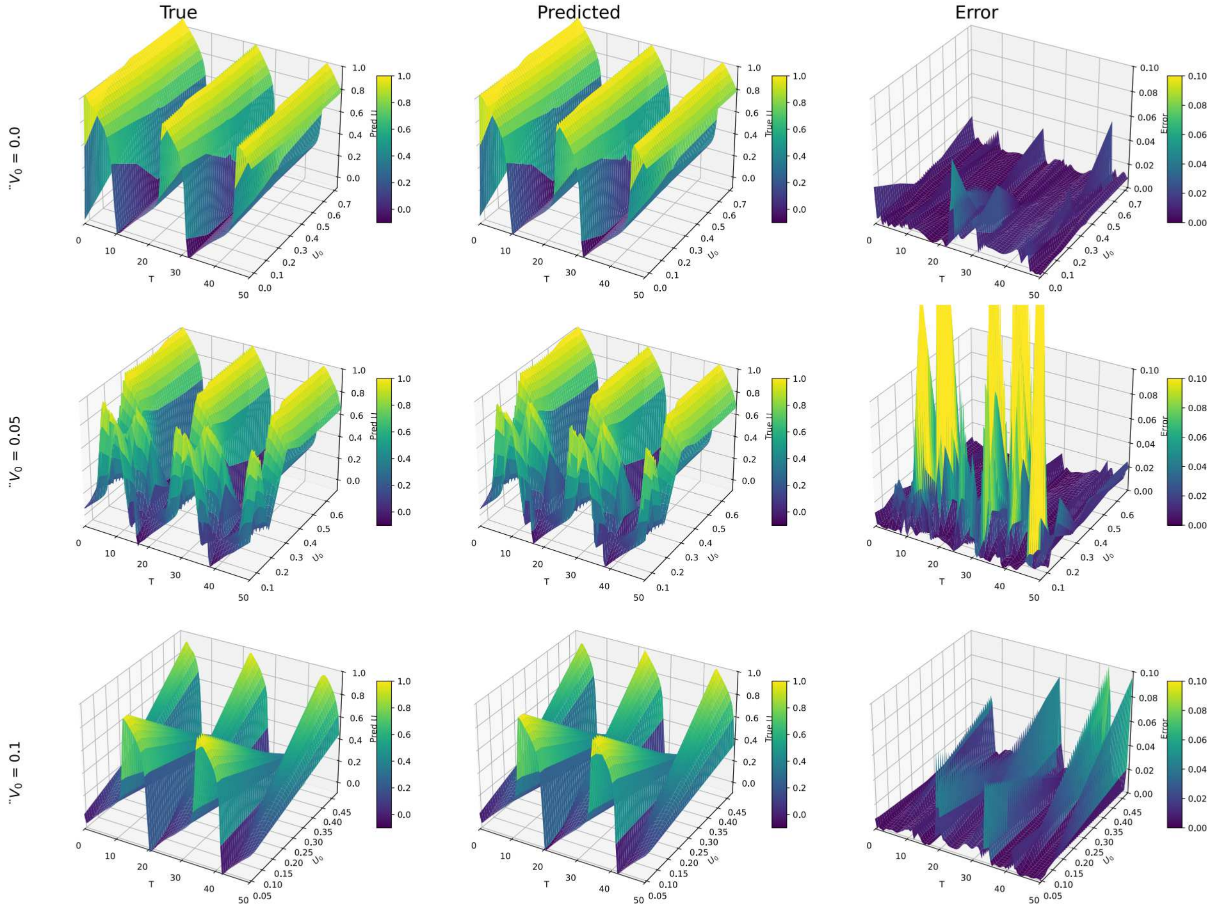

Figure 32 – High error region, illustrated through a cut in the parameter space for models trained on the larger time domain. This figure shows the distribution of errors across different initial conditions for U_0 and W_0, emphasizing the correlation between specific initial conditions and increased prediction errors.

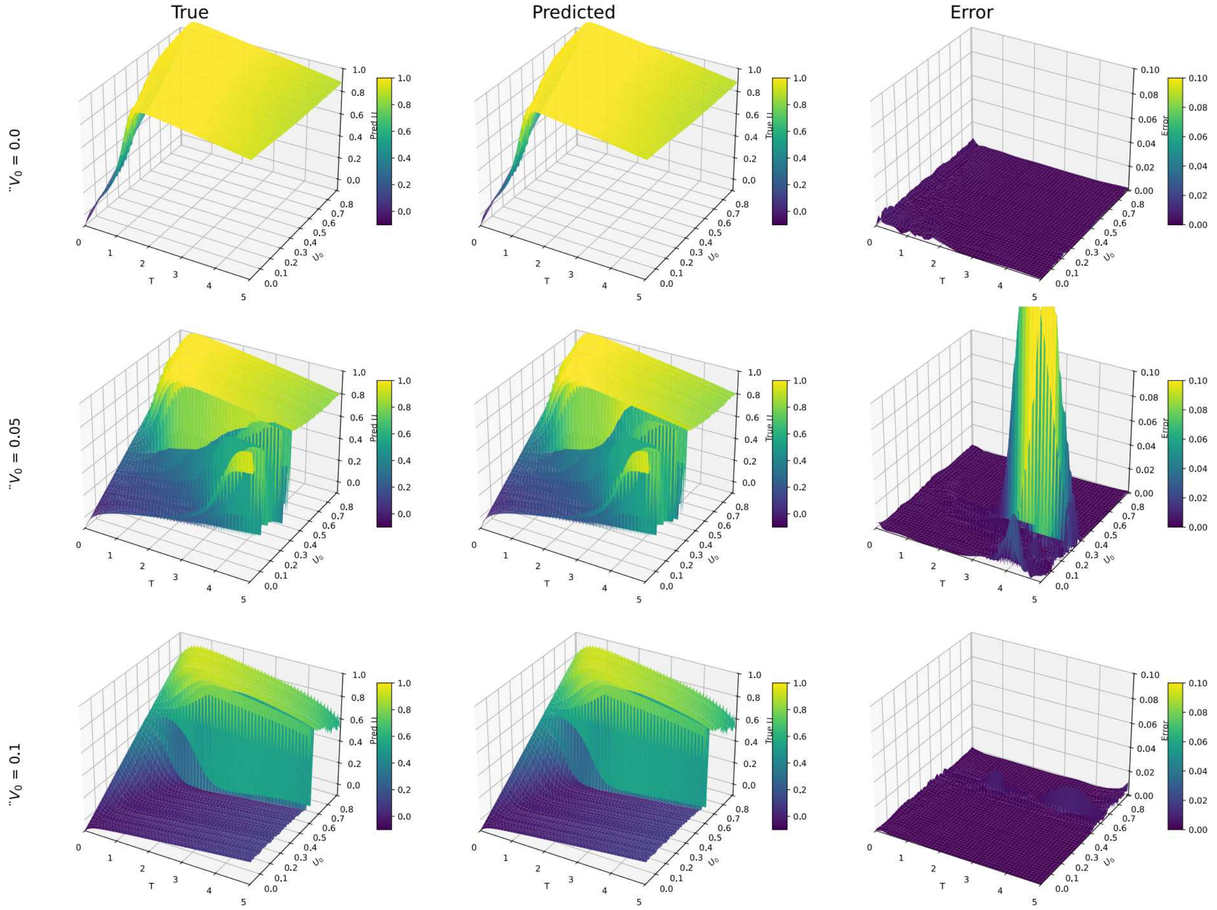

Figure 33 – High error region, illustrated through a cut in the parameter space for models trained on the smaller time domain. This figure highlights how the error distribution varies with different initial conditions for U_0 and W_0 and shows the impact of the decreasing the size of the time domain on the model's predictive accuracy.

4.2.3 Iterator Model

The extension of the concept of time domain splitting to its limit involves reducing the time domain to a single time step. For Problem B_C, Iterator Models were employed, requiring the model to learn the solution after 2 time units for any given initial condition. Due to the simpler nature of this problem, smaller models were tested, utilizing permutations of the following layer combinations: [(nn.SiLU, 16), (nn.Tanh, 16), (nn.SiLU, 8), (nn.Tanh, 8)]. The models were trained on two datasets: one comprised a homogeneous sample of the parameter space with a size of 5,000,000, while the other included points from the previously identified high error region. This region is identified by training the model with the homogeneous set and selecting points of high error (top 5%) and sampling points near then.

The Iterator Models are validated using two distinct sets. The first set maintained the same structure as the training set, simply mapping solution states to their values after the time interval had elapsed. The second set tracked the evolution of the solutions across the time domain. In this case, the initial condition was interactively passed through the Iterator Model to obtain predictions over the time domain, which were then compared with the validation set. The sets included 100,000 and 10,000 different initial conditions homogeneously sampled from the domain, respectively.

The initial results, presented in Figure 34, demonstrate the effectiveness of incorporating the extra data constraint in the high error region by comparing models trained with and without this constraint. Overall, models utilizing the extra constraint exhibited superior performance across both validation sets. Notably, this effect was more pronounced in the evaluation of the entire time domain set, likely due to the compounding effect of errors from successive passes through the network. Consequently, small reductions in error during initial evaluations can lead to significant improvements in later parts of the solution.

Despite training with the extra constraint, the same high-error region phenomena were observed for the iterative model, as depicted in Figure 35. However, for the iterative model, the error becomes more spread throughout the domain. This occurs because, when the iterative model's U and W approach the high-error region, increased errors are introduced, detaching the network's predictions from the actual solutions, as illustrated in Figure 36. This leads dispersed errors across the domain, as shown in Figure 35, compared to the continuous model.

Next, after establishing its effectiveness, a larger pool of models was trained, with the extra constraint, in order to produce the smallest and most efficient models possible. A set of small two-layer models was considered, generated with permutations of the layers: [(nn.SiLU, 16),(nn.Tanh, 16),(nn.SiLU, 8),(nn.Tanh, 8),(nn.ELU, 8),(nn.ELU, 16)], resulting in models with only 16 to 32 neurons. The results notably show great variation in model performance, with the best model [(ELU,16), (SILU,8)] having an average error of 0.0043 and the worst model [(ELU,8),(ELU,16)] having 0.162 for the whole time domain validation set -a difference of more than 40 times- while having the same neuron count. Additionally, although the error in the single time point and whole time domain validation sets are highly correlated, results show that if one model has a lower mean error over a single iteration than another model, it does not necessarily mean it will have a smaller error over the whole domain. Table 9 highlights the best-performing models. Notably, smaller iterator models were not able to beat models trained with a smaller window accuracy wise, seen as a 16 neuron iterator model achieved around $2 \cdot 10^{-2}$ of mean error while a continuous model with the same neuron count handling a small time window achieved $9 \cdot 10^{-3}$.

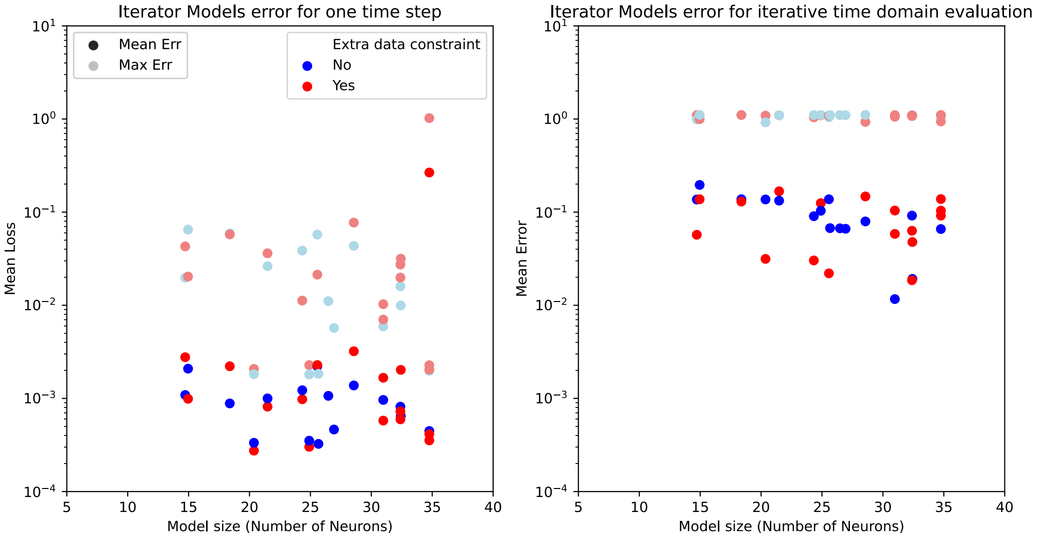

Figure 34 – Model accuracy over both validation sets (single time step, and whole time domain solution) after training end for models trained with and without the extra data constraint in the high error region. Models along the same y-axis share the same neuron count.

Table 9 – Summary of notable models for Problem B_C based on their mean error and neuron count. The table includes the smallest models for two error thresholds, the latter one also being the best overall model.

Layers	**Mean err**	**Neuron count**	**Note**
(TANH-8, SILU-8)	2.78e-02	16	Smallest model < 0.1
(ELU-16, SILU-8)	4.36e-03	24	Smallest model < 0.01

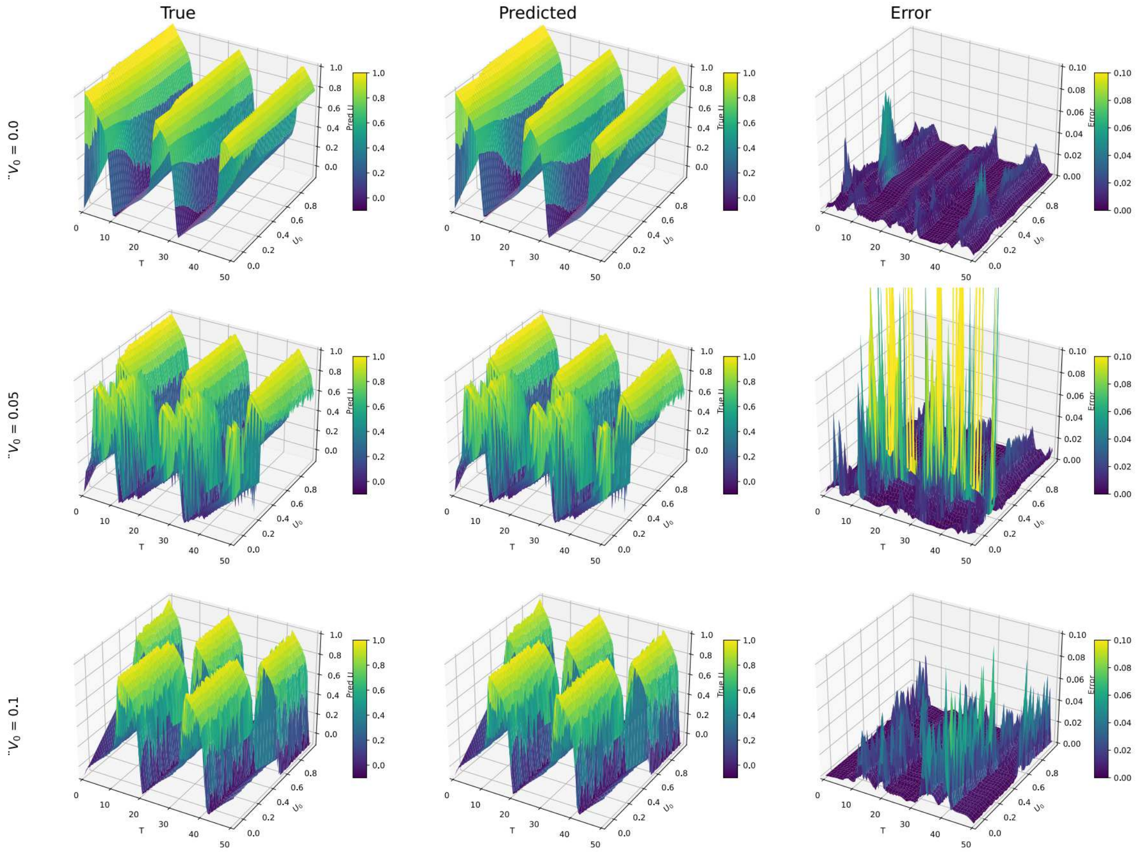

Figure 35 – Best Iterator Model error in some cuts of the parameter space. Results show regions at the border (first and last rows) and within the region of highest error (middle row). Notably, results show a maximum error of similar scale as the continuous model, but more spread through the domain.

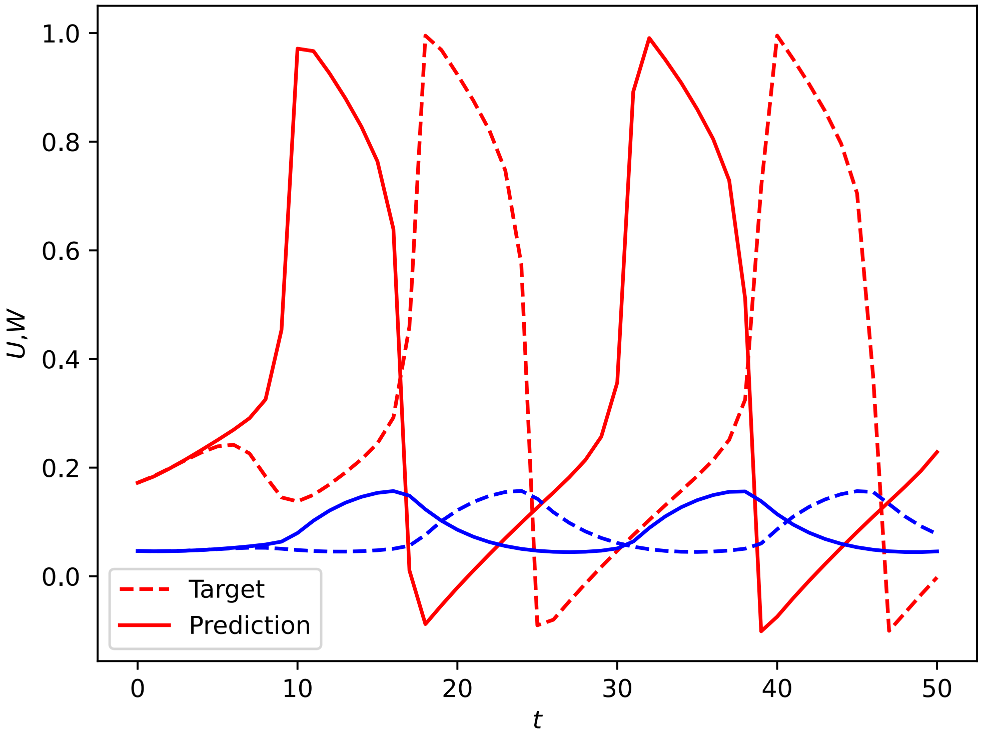

Figure 36 – High error solution achieved with the best trained iterator model. Plot exemplifies decoupling occurring near the high error region.

4.3 Problem C

In this final problem, the task involves learning multiple families of solutions for different configurations of the FHN model, specifically by varying the model parameter K. This parameter, in the cyclic formulation, regulates the applied current I_{app}, controlling both the size and frequency of the oscillations. K is normalized such that $I_{\text{app}} = 0.08$ when $K = 0$, and $I_{\text{app}} = 0.12$ when $K = 1$, with a linear scaling between these two extremes. The solution behavior within these boundaries is illustrated in Figure 4. This parametrization introduces the most complex scenario tested in the study, as the network must learn solutions for any initial conditions (U_0, W_0), across the entire range of K, and throughout the full time domain.

To further enhance model performance in this complex scenario, the increased cloud point density technique was applied in this problem in the form of an extra data constraint with only points in high error regions. An initial study assessed the effectiveness of this technique before conducting the full architecture grid search. The high-error regions were identified by training a model, with a base set that uniformly explored the parameter space, and selecting points (U_0, W_0, K) where the solution had a mean error exceeding 0.1. A new set was then created by sampling the region around these points. Figure 37 demonstrates the accuracy of models trained with and without the extra constraint, showing that the best-performing models were trained with the extra constraint, validating its inclusion in the grid search process. Furthermore, the use of physics-informed constraints was tested—either across the entire domain or limited to the high-error region. However, they did not yield any significant improvement in model performance.

The grid search was conducted on both 2- and 3-layer models, using combinations of the following activation functions and layer sizes: $(\text{nn.SiLU}, 8)$, $(\text{nn.SiLU}, 16)$, $(\text{SIN}, 8)$, $(\text{nn.Tanh}, 8)$, $(\text{nn.Tanh}, 16)$, $(\text{SIN}, 16)$, $(\text{nn.ELU}, 8)$, and $(\text{nn.ELU}, 16)$. These models were categorized according to their shape, based on previously defined classifications. Figure 38 displays the performance of all trained models on the validation set. Once again, the optimization of architecture yielded models with widely varying performances, despite having the same neuron count. In this instance, however, architecture shape was less influential than in previous cases; models with identical shapes demonstrated significant performance variation depending on the choice of activation functions. A notable exception was the Bowtie architecture, which produced a few models that outperformed any other architectures with the same neuron count (32). Table 10 summarizes the best models identified for this problem.

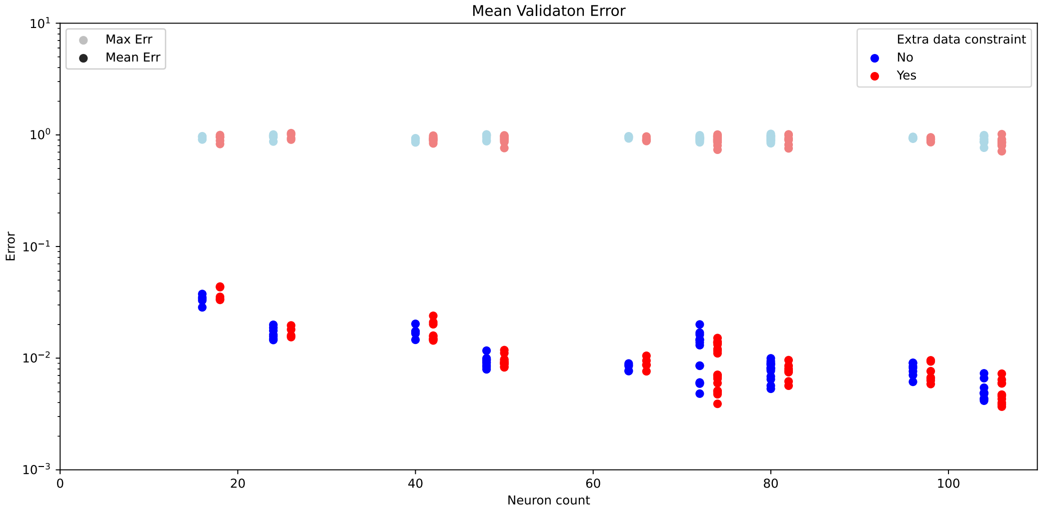

Figure 37 – Accuracy for multiple models trained for Problem C with and without the extra data constraint. Models along the same x-axis have the same neuron count. For ease of visualization, different categories have been slightly displaced on the x-axis. Results show that the best-generated model is trained with the extra constraint.

Table 10 – Summary of selected models based on their mean error and neuron count for Problem C. The table includes the smallest models under various error thresholds and the best performing models.

Layers	**Mean Err**	**Neuron Count**	**Note**
(ELU−8, SILU−16, TANH−8)	9.08e-03	32	Smallest model < 0.01
(ELU−8, TANH−8)	2.80e-02	16	Smallest model < 0.1
(ELU−8, SILU−16, TANH−8)	9.08e-03	32	Best model (neuron count less than 32)
(ELU−16, TANH−16, SIN−16)	5.42e-03	48	Overall best model

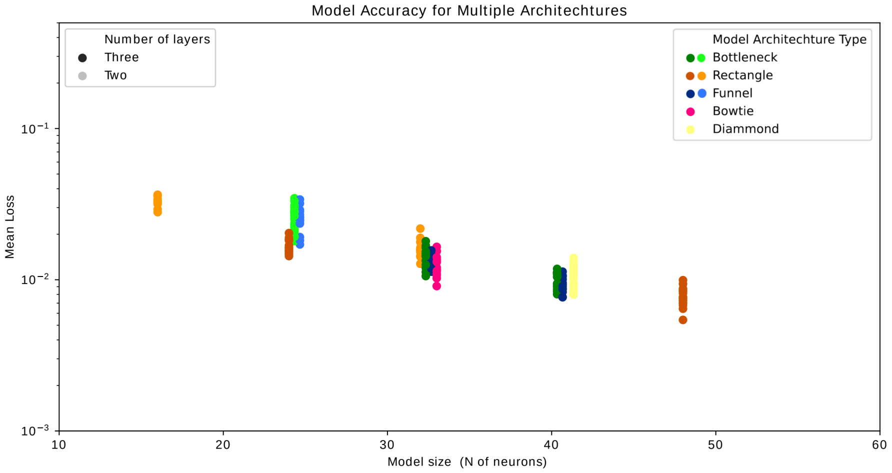

Figure 38 – Accuracy for multiple models trained for Problem C with the extra data constraint. Models along the same x-axis have the same neuron count but have different activation functions and model architectures. For ease of visualization, different categories have been slightly displaced on the x-axis. Results show great variation for the same architecture and a clear advantage for three-layered models.

4.4 Inference

To assess the viability of using trained models as substitutes for numerical solvers, it is crucial to consider not only accuracy but also computational efficiency. The goal is to develop neural network surrogate models that are less computationally expensive than their numerical counterparts while maintaining similar levels of error. For this reason, model inference performance was compared to a CUDA-optimized implementation of the Euler method. Table 11 provides benchmark results for the Euler CUDA implementation, which serve as the baseline for comparison. The reference performance is based on processing a set of 16,000,000 samples with a time step $\Delta t = 0.1$, running until $t = 10$. Neural network inference performance was measured using the same sample set, following batch size optimization.

Table 11 – Total runtime results of the Cuda implementation of the Euler method applied to the FHN model with dt=0.1 . Since the base model is the same, the runtime results for the numerical method are also the same for Problems A,B and C.

Set size	**Cuda Time (s)**
16000000	1.325e-01
32000000	2.621e-01
64000000	5.270e-01

In the first set of results, we examine how model size and architecture affect inference performance, including the effectiveness of tensor core utilization. A large pool of untrained models was generated using permutations of SiLU layers: [(SiLU, 8), (SiLU, 16), (SiLU, 32), (SiLU, 64), (SiLU, 128)], with up to 5 layers. Only the SiLU activation function was used because activation functions had little impact on inference cost, as detailed in Figure 42. The resulting models ranged from 8 to 640 neurons and had 1 to 5 layers. Figure 39 illustrates the relationship between neuron count, number of layers, and inference cost. The plot suggests a linear relationship between neuron count and cost, with a smaller yet significant effect from the number of layers. To further explore this relationship, we consider the explicit model:

$$Q(\text{Inference time}) \approx N_{\texttt{neuron}} \cdot C(N_{\text{layer}}), \tag{4.3}$$

where the inference time is explained by the number of neurons and the number of layers (as a categorical variable with 5 levels). Results in Table 12 show that inference time can largely be explained by the number of neurons, as indicated by the high F-value, while subdividing the same number of neurons into additional layers incurs only a small additional cost. This contrasts with the accuracy improvements observed from increasing the number of nonlinear activation functions, as discussed in previous sections.

Next, we assessed the impact of utilizing Tensor Cores across different models. Tensor Cores primarily accelerate linear transformations, and their performance is highly sensitive to matrix sizes that align with their optimal dimensions. As a result, the effectiveness of Tensor Cores varies significantly based on the architecture of each model. Figure 40 illustrates the speedup obtained when running models with TensorRT (leveraging Tensor Cores) compared to PyTorch (using only CUDA cores), with all other variables kept constant. The results indicate that Tensor Cores provide greater benefits in larger models (up to almost twice as fast), although with diminishing returns as the model size continues to increase. Interestingly, for a given neuron count, models divided into more layers exhibited reduced benefit from Tensor Cores, suggesting that architectural complexity plays a role in limiting the efficiency gains from Tensor Cores in deep networks.

An ANOVA analysis, using a model similar to Equation 4.3 but with speedup as the response variable, confirmed these observations. The F-value for the number of layers was a much larger proportion of the total variance compared to the ANOVA for inference time, indicating a stronger influence on speedup. The fitted coefficients, which were positive for neuron count and negative for layer count, further supported the observed trend: models with a higher neuron count benefit more from Tensor Cores, while deeper models with more layers show diminished performance improvement. This suggests that while Tensor Cores excel at accelerating models with large matrix operations, their advantage decreases in deeper architectures where other computational bottlenecks may arise.

Inference primarily consists of two operations: linear transformations and the evaluation of non-linear activation functions, each of which occurs once per layer. Deeper architectures, as opposed to wider ones, tend to have a higher percentage of their inference cost dominated by the evaluation of activation functions. Since TensorRT does not accelerate the execution of these functions, its overall benefit is reduced for deeper models. This explains the observed trend, where deeper models receive less acceleration from Tensor Cores due to the unchanged cost of non-linear operations.

Table 12 – ANOVA summary showing the impact of the number of layers and `neuron` count on model inference performance. Results demonstrate a significant effect of both the number of layers and neurons, with both fitted coefficients being positive, indicating an increase in cost for increasing either, although neuron count has a clearly dominant influence indicated by the larger F-value.

Source	**Fitted Coef**	**Sum Sq**	**F-value**	**P-value**
layers	0.0128	3.14×10^{-2}	171.0	7.22×10^{-36}
neuron	0.0015	1.29×10^{1}	70300.0	0.00
layers:neuron	$> 10^{-5}$	1.72×10^{-3}	9.33	2.32×10^{-3}
Residual		1.66×10^{-1}		

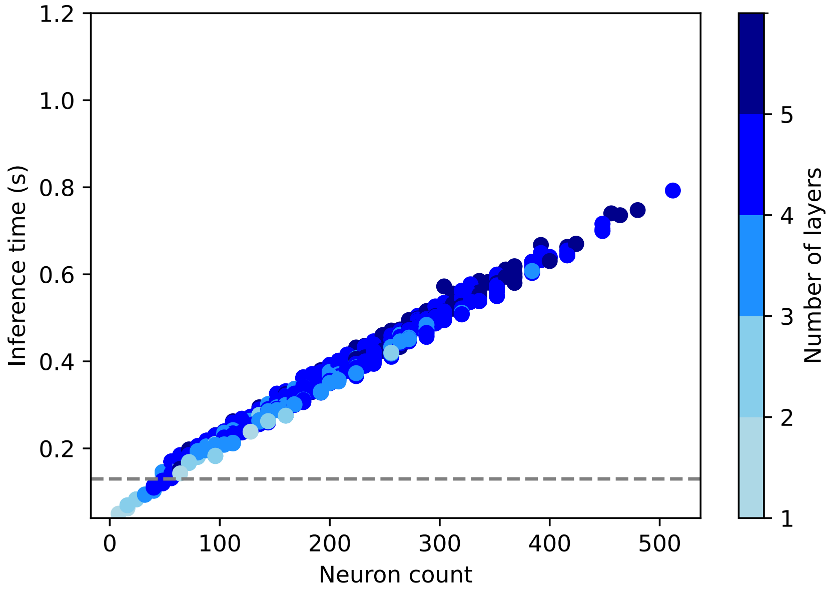

Figure 39 – Inference time for the reference set for multiple untrained models with various architectures. Inference is done using TensorRT. Results show a linear increase in cost as neuron count increases. Dotted grey line shows the Euler implementation results for the same set.

Table 13 – ANOVA summary showing the impact of the number of layers and `neuron` count on the speedup due to running inference with tensor cores as opposed to only cuda cores. The results indicate significant effects of both layers and neurons, with a high F-value. With a positive coefficient, a higher neuron count increases the efficacy of the tensor cores, while with a negative coefficient, a higher number of layers decreases it.

Source	**Fitted Coef**	**Sum Sq**	**F-value**	**P-value**
layers	-0.0122	1.91×10^{0}	129.0	5.85×10^{-28}
neuron	0.0043	4.80×10^{1}	3230.0	9.23×10^{-301}
layers:neuron	$> 10^{-5}$	6.15×10^{-1}	41.4	2.01×10^{-10}
Residual	905	1.35×10^{1}		

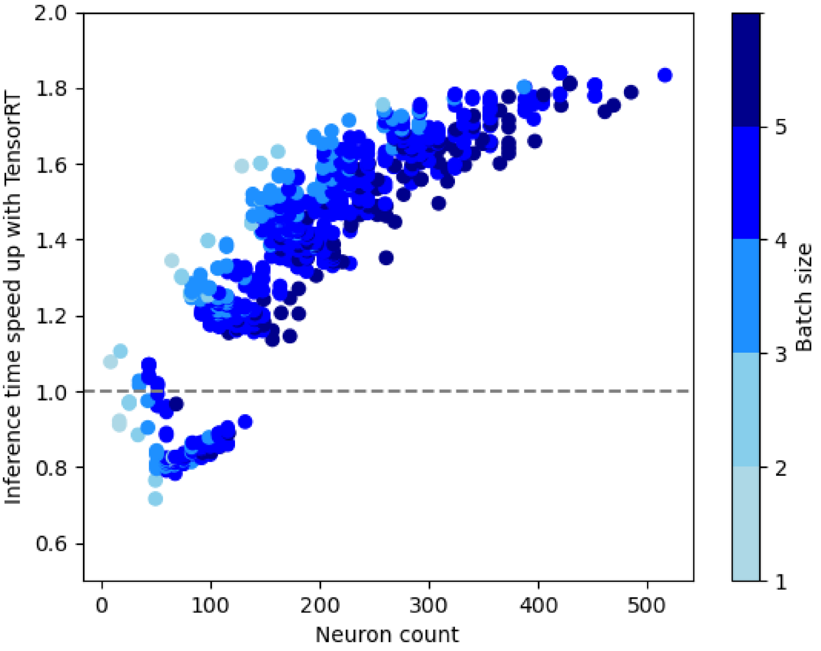

Figure 40 – Speedup in inference due to the use of TensorRT and tensor cores as opposed to only CUDA cores. Models are divided by the number of layers and organized by neuron count. The plot suggests a strong positive relation for neuron count and a negative relation for the number of layers.

Next, we take the best-performing models in terms of accuracy, i.e., mean error, across all the previous problems and evaluate each model's balance between accuracy and inference performance relative to the numerical implementation. Figure 5 shows that, for all problems tackled, several neural networks achieved both higher accuracy and faster inference times compared to the baseline numerical solver. Furthermore, models up to 1.8x faster than the numerical counterpart can be utilized when a certain error tolerance is acceptable. Finally, it is notable that as the complexity of the problem increases, the models become more computationally expensive, especially when higher accuracy is required.

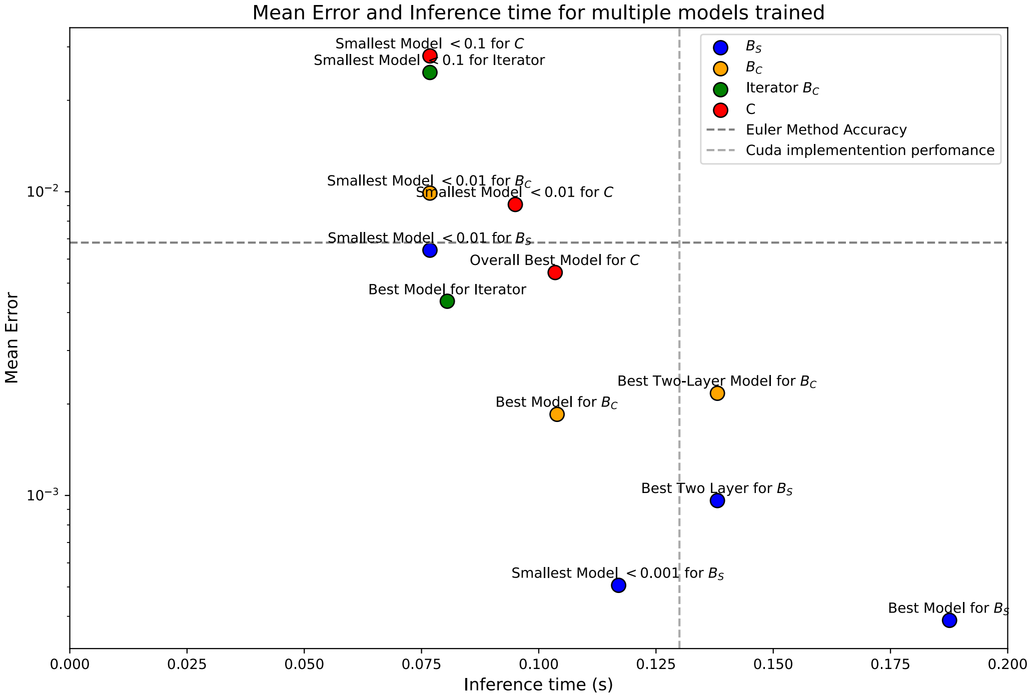

Figure 41 – Model error and Model inference performance for multiple models trained during this work. Numerical solution for the model using the Euler method and dt= 0.1 is show for reference with the grey doted lines.

4.5 Discussion

This study investigated the use of neural networks to solve complex differential equations in the context of modeling action potentials. Several key observations emerge from the results, highlighting the effectiveness of the approaches taken and the implications for future research. The results demonstrate that neural networks can offer some advantages over traditional numerical solvers, particularly when optimized to leverage modern hardware like Tensor Cores. These advantages are most notable when the complexity of the problem is managed through proper model design and optimization techniques. Overall, neural networks can provide faster solutions while maintaining the accuracy of traditional methods.

The incorporation of physical constraints through PINNs proved to be highly beneficial, especially in scenarios where data was scarce. By embedding physical laws directly into the loss function, PINNs demonstrated superior accuracy when trained with a scarce dataset, as shown for Problem A in Figure 17. This aligns with existing literature, such as (BHATNAGAR; COMERFORD; BANAEIZADEH, 2023), suggesting that PINNs are particularly effective in modeling complex physical systems with limited data availability. However, it is also clear from the results that the use of PINNs comes at significant computational cost, with iterations nearly 10 times more expensive than regular data-driven models, as shown in Figure 23. Training with these constraints required significantly more resources in terms of iteration time and offers little benefit if the training set is abundant, as shown in Figure 21, thus our results show that PINNs should be relegated to only data scarce scenario, or when the cost to produce the data surpass the cost of training with it.

Two specific techniques were employed in this study: increasing cloud point density and time domain splitting, both yielding meaningful results. The idea behind increasing the cloud point density is to enhance the model's ability to learn complex regions by introducing more training points in areas of interest. This technique effectively allowed the model to focus on regions where greater accuracy was required, improving overall performance. Notably, this technique only worked when the additional constraint incorporated new data of that specific region, as shown in Figure 34, specially reducing the max error. When the additional constraint included only physics for those regions, Figure 26 shows that it did not yield any benefit. However, the process of identifying these high-error regions and generating high-resolution data for them comes at a substantial computational cost, particularly when large datasets or complex models are involved. The other technique employed, time domain splitting, has the objective of reducing the complexity of the solution space. By breaking down the temporal domain into smaller subdomains, the model has a reduced mapping of inputs to outputs to learn, allowing for better accuracy results. As shown in Table 7 with a reduction of the time domain to a tenth of the original size, yielded an improvement in accuracy of 5× in average. Both techniques yielded meaningful results, enabling the use of smaller, and thus faster, models than would otherwise be possible, highlighting their use in applications focused on inference performance as wells as for improving training.

The architecture and optimization of the neural networks played a pivotal role in their performance. As shown in your results, models with the same neuron count achieved up to 10× less error when optimized with specific structures. Figure 30 illustrates that smaller models with carefully optimized architectures, such as the bowtie structure, achieved competitive accuracy with significantly fewer parameters than other top-performing models. This underscores the importance of selecting the right architecture for specific tasks, as larger models with more neurons and layers did not always lead to improved performance. However, it is important to acknowledge the high computational cost of model optimization. Since the ideal architecture is not known beforehand, multiple models must be trained, which is computationally expensive. To address this challenge, optimization techniques such as adaptive learning rates (e.g., the Adam optimizer) and parallel training with two A-100 GPUs were employed to ensure timely completion of the training process. Furthermore, our results suggest that, for this particular problem, the bowtie and simple rectangular shapes should be prioritized. A promising avenue for future research would be to verify if these patterns hold for other ODEs.

The comparison between the neural networks and traditional numerical solvers, particularly in terms of inference time, demonstrated that while it is possible to achieve an advantage for the former in some cases, some restrictions apply. Problem complexity is a significant one, with neural networks trained on smaller problems being able to match the accuracy of traditional numerical solvers while achieving up to 1.8× faster inference times (Figure 5). This performance boost is particularly relevant for real-time or large-scale simulations where computational efficiency is critical. However, as problem complexity increased, larger models were required to achieve levels of accuracy similar to the numerical method, and the advantage diminished, even after applying techniques to significantly reduce problem complexity or improve training. This highlights the possibilities and limitations of using neural networks for solving ODEs. For small parametrization of the models, small and very cheap models can be trained that beat their numerical counterparts, but as more parameters are considered, larger and more expensive models are required, and the numerical solution becomes cheaper, since it is not affected by parametrization.

When considering both the FHN model and neural networks as surrogates for the complex Hodgkin-Huxley model, it is clear that for most applications requiring full parametrization, the numerical solution of the ODE model is the better choice. This is particularly true for reduced-order models like FHN, which reduce the number of equations, using prior mathematical insights. However, in cases where problem complexity can be controlled, neural networks can offer significant speedups, especially when utilizing modern hardware like tensor cores. With careful handling, neural networks could also tackle more complex models, providing even greater benefits.

5 CONCLUSION

This work aimed to evaluate the effectiveness of Physics-Informed Neural Networks (PINNs) and data-driven models as efficient alternatives to traditional numerical solutions in modeling cardiac action potentials. The FitzHugh-Nagumo model was chosen to test the neural networks across various complexity levels, starting with only the independent variable as a parameter, followed by the inclusion of initial conditions, and finally expanding to additional model parameters. Multiple techniques were employed to improve training and speed up inference.

The neural networks were trained using a combination of numerical data and physical laws derived from the underlying ODEs. By embedding physical constraints through PINNs, we achieved significant gains in data-scarce scenarios. However, our results show the substantial cost of training PINNs and the lack of benefit in abundant data settings. Thus, our work demonstrates a key advantage of PINNs over purely data-driven approaches when data is scarce, while emphasizing the costly trade-off that makes their use situational.

Techniques were employed to enhance training, improve model accuracy, and reduce problem complexity. Architecture optimization played a crucial role in balancing the trade-off between model size and performance, leading to faster and more accurate predictions. Additionally, time-domain splitting divided the temporal domain into smaller segments significantly reducing problem complexity, while increasing the cloud point density in high-error regions ensured better training for stiff regions. These techniques allowed the neural networks to efficiently handle the complexities of the action potential model, maintaining a good balance between model size and accuracy even as the problem's complexity increased.

Moreover, significant improvements in model inference were achieved through the use of the TensorRT SDK, which leveraged specialized tensor cores to accelerate matrix calculations present in neural network inference. This optimization resulted in faster inference times, up to twice as fast when compared to standard PyTorch-based implementations, making the neural networks more viable for real-time applications. The improvements in speed due to the use of the tensor cores were essential for some networks to be up 1.8× faster than the numerical counterparts.

In conclusion, this work demonstrated the potential of PINNs and data-driven neural networks as powerful tools for modeling excitable cells in cardiac electrophysiology, offering surrogates for HH model as good as the phenomenological FHN model. The neural networks showed improvements in efficiency over FHN in many scenarios, as shown , though the computational advantages diminished as problem complexity increased. However, this work also demonstrated that complexity can be effectively managed through proper formulation of the problem and model design. With well-structured problem design, it is likely that for more computationally expensive models than FHN, the improvements could be even more substantial, possibly achieving up to 10× speedup. Future work should focus on further optimizing neural network architecture and hardware acceleration, tackling more complex ODE or PDE models such as the Ten Tusscher (TUSSCHER; PANFILOV, 2006) and Monodomain (FRANZONE; PAVARINO, 2004) respectively, and exploring integration with traditional solvers to enhance the scalability and applicability of these neural network models.

Bibliography

BAYDIN, A. G. et al. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, v. 18, n. 153, p. 1–43, 2018.

BHATNAGAR, S.; COMERFORD, A.; BANAEIZADEH, A. Physics informed neural networks for modeling of 3d flow-thermal problems. *Journal of Machine Learning and Model Computation*, 2023. Disponível em: <https://dx.doi.org/10.1615/jmachlearnmodelcomput.2024051540>.

BOTTOU, L.; CURTIS, F. E.; NOCEDAL, J. Optimization methods for large-scale machine learning. *SIAM Review*, v. 60, n. 2, p. 223–311, 2018.

BOYD, S.; VANDENBERGHE, L. *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004. ISBN 9780521833783.

BROWN, T. B. et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

BUTCHER, J. C. *Numerical Methods for Ordinary Differential Equations*. [S.l.]: John Wiley & Sons, 2016.

CAMPOS, J. et al. Uncertainty quantification and sensitivity analysis of left ventricular function during the full cardiac cycle. *Philosophical Transactions of the Royal Society A*, The Royal Society Publishing, v. 378, n. 2173, p. 20190381, 2020.

CHEN, J. et al. Physics-informed machine learning for electromagnetic field predictions and parameter inversion. *IEEE Transactions on Antennas and Propagation*, IEEE, v. 68, n. 10, p. 7276–7287, 2020.

COOREY GLEN, F. G. A. F. D. F. S. V. J. The health digital twin to tackle cardiovascular disease—a review of an emerging interdisciplinary field. *npj Digital Medicine*, Nature Publishing Group, v. 5, n. 1, p. 6, 2022.

FRANZONE, P. C.; PAVARINO, L. F. A parallel solver for reaction-diffusion systems in computational electrocardiology. *Mathematical Models and Methods in Applied Sciences*, v. 14, n. 6, p. 883–911, 2004.

GLASNER, K. Data-driven discovery of differential equations with uncertainty quantification. *Journal of Computational Science*, v. 36, p. 101020, 2019.

GRANDITS, T. et al. Neural network emulation of the human ventricular cardiomyocyte action potential for more efficient computations in pharmacological studies. *Scientific Reports*, Nature Publishing Group, v. 11, n. 1, p. 7365, 2021.

HAGHIGHAT, E. et al. Physics-informed deep learning for computational elastodynamics without labeled data. *Journal of Computational Physics*, Elsevier, v. 404, p. 109014, 2021.

HAYKIN, S. *Neural Networks: Principles and Practice*. [S.l.]: Bookman Publisher, 2001.

HODGKIN, A. L.; HUXLEY, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, Wiley Online Library, v. 117, n. 4, p. 500–544, 1952.

HORNIK, K.; STINCHCOMBE, M.; WHITE, H. Multilayer feedforward networks are universal approximators. *Neural Networks*, Elsevier, v. 2, n. 5, p. 359–366, 1989.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. [s.n.], 2015. Disponível em: <https://arxiv.org/abs/1412.6980>.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, v. 60, n. 6, p. 84–90, 2012.

LEPPER, A. D.; BUCK, C. From evidence-based medicine to digital twin technology for predicting ventricular tachycardia in ischaemic cardiomyopathy. *Journal of the Royal Society Interface*, Royal Society, v. 19, n. 188, p. 20220317, 2022.

LU, Z. et al. The expressive power of neural networks: A view from the width. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2017. p. 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

MASLYAEVA, M.; HVATOVA, A. Discovery of the data-driven differential equation-based models of continuous metocean process. *Procedia Computer Science*, v. 156, p. 159–166, 2019.

MENG, C. et al. When physics meets machine learning: A survey of physics-informed machine learning. *arXiv preprint arXiv:2204.02679*, 2022.

NVIDIA. *CUDA C Best Practices Guide*. [S.l.], 2023. Accessed: 2024-09-16. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#instruction-optimization>.

OTTO, S. E.; ROWLEY, C. W. Linearly recurrent autoencoder networks for learning dynamics. *SIAM Journal on Applied Dynamical Systems*, v. 18, p. 558–593, 2019.

PASSERINI, D. C. A. T. M. T. *Artificial Intelligence for Computational Modeling of the Heart*. Elsevier, 2020. Disponível em: <https://www.sciencedirect.com/book/9780128175941/artificial-intelligence-for-computational-modeling-of-the-heart>.

PLANK, G.; MESAROVIC, S.; TRAYANOVA, N. Machine learning for computational modeling of cardiac electrophysiology: A review. *Journal of Cardiovascular Electrophysiology*, Wiley, v. 31, n. 1, p. 123–135, 2020.

QIAN et al. Accelerating sparse deep neural network inference using gpu tensor cores. In: IEEE. *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. [S.l.], 2022. p. 1–6.

RAISSI, M.; PERDIKARIS, P.; KARNIADAKIS, G. E. Inferring solutions of differential equations using noisy multi-fidelity data. *Journal of Computational Physics*, v. 340, p. 127–143, 2017.

RAISSI, M.; PERDIKARIS, P.; KARNIADAKIS, G. E. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017. Disponível em: <https://arxiv.org/abs/1711.10561>.

RAISSI, M.; PERDIKARIS, P.; KARNIADAKIS, G. E. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *Journal of Computational Physics*, Elsevier, v. 378, p. 686–707, 2019.

SAHLI, M.; RAISSI, M.; KARNIADAKIS, G. E. Ep-pinns: Cardiac electrophysiology characterisation using physics-informed neural networks. *Journal of Computational Physics*, Elsevier, v. 423, p. 109787, 2020.

SEL K., O. D. Z. F. Building digital twins for cardiovascular health: From principles to clinical impact. *Journal of the American Heart Association*, American Heart Association, v. 13, p. e031981, 2024.

SICILIANO, R. *The Hodgkin-Huxley Model: Its Extensions, Analysis and Numerics.* 2012. Retrieved from <https://www.math.mcgill.ca/gantumur/docs/reps/RyanSicilianoHH.pdf>.

TUSSCHER, K. H. W. J. T.; PANFILOV, A. V. Alternans and spiral breakup in a human ventricular tissue model. *American Journal of Physiology-Heart and Circulatory Physiology*, v. 291, n. 3, p. H1088–H1100, 2006.

VASWANI, A. et al. Attention is all you need. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [S.l.: s.n.], 2017. p. 5998–6008.

WANG, Q. et al. Shfl-bw: Accelerating deep neural network inference with tensor-core aware weight pruning. In: *2021 IEEE International Conference on Computer Design (ICCD)*. [S.l.]: IEEE, 2021. p. 470–477.

WANG, S.; YU, X.; PERDIKARIS, P. Understanding and mitigating gradient pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, SIAM, v. 43, n. 5, p. A3055–A3081, 2021.

WU, Y.; SICARD, B.; GADSDEN, S. A. Physics-informed machine learning: A comprehensive review on applications in anomaly detection and condition monitoring. *Expert Systems with Applications*, Elsevier, v. 214, p. 124678, 2024.

6 Suplementary Material

Table 14 – Accuracy of models for the Problem A, with trained with a scarce set after 2 million iterations. Results show that the employ of a PINN constraint is beneficial for all models in this scarce data scenario. .

Layers	Mean err	Max err	PINN
(SILU'>-16), (SILU'>-16)	3.17e-04	1.63e-03	True
(SILU'>-16), (SILU'>-16)	2.07e-02	3.87e-01	False
(SILU'>-16), (SILU'>-16), (SILU'>-16)	4.38e-04	2.80e-03	True
(SILU'>-16), (SILU'>-16), (SILU'>-16)	1.92e-02	3.45e-01	False
(SILU'>-16), (SILU'>-16), (SILU'>-32)	2.37e-04	1.88e-03	True
(SILU'>-16), (SILU'>-16), (SILU'>-32)	2.25e-02	3.55e-01	False
(SILU'>-16), (SILU'>-16), (TANH'>-8)	1.79e-04	2.55e-03	True
(SILU'>-16), (SILU'>-16), (TANH'>-8)	1.93e-02	4.11e-01	False
(SILU'>-16), (SILU'>-32)	7.65e-04	1.36e-02	True
(SILU'>-16), (SILU'>-32)	1.66e-02	3.68e-01	False
(SILU'>-16), (SILU'>-32), (SILU'>-16)	3.93e-04	8.45e-03	True
(SILU'>-16), (SILU'>-32), (SILU'>-16)	8.02e-02	1.78e+00	False
(SILU'>-16), (SILU'>-32), (TANH'>-8)	1.66e-04	2.12e-03	True
(SILU'>-16), (SILU'>-32), (TANH'>-8)	2.58e-02	5.78e-01	False
(SILU'>-32), (SILU'>-16)	1.31e-04	2.06e-03	True
(SILU'>-32), (SILU'>-16)	1.97e-02	4.15e-01	False
(SILU'>-32), (SILU'>-16), (SILU'>-16)	1.10e-03	5.40e-03	True
(SILU'>-32), (SILU'>-16), (SILU'>-16)	2.99e-02	4.29e-01	False
(SILU'>-32), (SILU'>-16), (SILU'>-32)	2.03e-04	1.15e-03	True
(SILU'>-32), (SILU'>-16), (SILU'>-32)	4.37e-02	9.74e-01	False
(SILU'>-32), (SILU'>-16), (TANH'>-8)	1.08e-04	1.66e-03	True
(SILU'>-32), (SILU'>-16), (TANH'>-8)	1.49e-02	3.73e-01	False
(SILU'>-32), (SILU'>-32)	9.57e-04	4.20e-02	True
(SILU'>-32), (SILU'>-32)	3.37e-02	5.96e-01	False
(SILU'>-32), (SILU'>-32), (SILU'>-16)	1.50e-03	2.64e-02	True
(SILU'>-32), (SILU'>-32), (SILU'>-16)	1.77e-02	4.09e-01	False
(SILU'>-32), (SILU'>-32), (SILU'>-32)	1.47e-04	2.01e-03	True
(SILU'>-32), (SILU'>-32), (SILU'>-32)	2.24e-02	4.00e-01	False
(SILU'>-32), (SILU'>-32), (TANH'>-8)	1.17e-04	1.54e-03	True
(SILU'>-32), (SILU'>-32), (TANH'>-8)	2.30e-02	4.83e-01	False

Table 15 – Accuracy of models for the Problem A, with trained with a very dense complete set after 2 million iterations. Results show that the employ of a PINN constraint slightly beneficial for some models and not for others, the difference is not larger enough to be significant.

Layers	**Mean err**	**Max err**	**PINN**
(SILU'>-16), (SILU'>-16)	2.39e-04	1.64e-03	True
(SILU'>-16), (SILU'>-16)	9.27e-04	8.34e-03	False
(SILU'>-16), (SILU'>-32)	2.47e-04	4.24e-03	False
(SILU'>-16), (SILU'>-32)	1.02e-03	7.23e-03	True
(SILU'>-32), (SILU'>-16)	2.74e-04	5.23e-03	False
(SILU'>-32), (SILU'>-16)	4.97e-04	3.27e-03	True
(SILU'>-32), (SILU'>-32)	1.66e-04	1.36e-03	True
(SILU'>-32), (SILU'>-32)	3.63e-04	1.16e-02	False
(SILU'>-32), (TANH'>-8)	3.66e-04	2.72e-03	True
(SILU'>-32), (TANH'>-8)	1.04e-03	7.71e-03	False
(TANH'>-8), (SILU'>-16)	3.39e-04	4.92e-03	False
(TANH'>-8), (SILU'>-16)	3.70e-04	2.05e-03	True
(TANH'>-8), (SILU'>-32)	3.64e-04	5.52e-03	False
(TANH'>-8), (SILU'>-32)	5.05e-04	7.50e-03	True

Table 16 – Best performing models in terms of medium accuracy over the validation set, trained for Problem B_S. Results show that all best performance are three layers models, and that while larger models in terms of neuron count also dominated the chart, smaller models with 96 and 72 neurons were able to be trained with good results trough model optimization.

Layers	**Mean err**	**Neuron count**	**class**
(SILU-32, TANH-32, SILU-32)	0.00038831	96	rectangle
(TANH-64, TANH-32, SILU-64)	0.00039900	160	bowtie
(SILU-64, SILU-64, SILU-64)	0.00040598	192	rectangle
(SILU-32, TANH-64, SILU-8)	0.00042069	104	diamond
(TANH-32, TANH-64, SILU-64)	0.00042301	160	bottleneck
(SILU-32, SILU-64, SILU-32)	0.00043483	128	diamond
(TANH-32, TANH-32, SILU-64)	0.00044650	128	bottleneck
(SILU-64, SILU-64, SILU-32)	0.00045884	160	funnel
(SILU-32, TANH-32, ELU-8)	0.00046274	72	funnel
(TANH-64, TANH-64, SILU-64)	0.00046464	192	rectangle
(TANH-32, TANH-32, SILU-32)	0.00046883	96	rectangle
(SILU-32, TANH-64, SILU-32)	0.00047633	128	diamond
(TANH-64, SILU-64, SILU-64)	0.00048618	192	rectangle
(TANH-64, SILU-32, SILU-64)	0.00048809	160	bowtie
(ELU-8, TANH-64, SILU-32)	0.00048935	104	diamond
(SILU-64, TANH-32, ELU-8)	0.00049106	104	funnel
(SILU-64, SILU-64, SILU-8)	0.00049591	136	funnel
(SILU-64, TANH-32, SILU-64)	0.00049913	160	bowtie
(ELU-8, TANH-32, SILU-32)	0.00050388	72	bottleneck
(SILU-8, TANH-32, SILU-8)	0.00050557	48	diamond

Table 17 – Model performance in terms of mean error over the validation set for models with 48 neurons and different three layers architectures in Problem B_S. Results show that models with the same size had vastly different performance depending on architecture, showcasing importance of model optimization. In this case, bowtie models were the best performing.

Layers	**Mean err**	**Neuron count**	**class**
(SILU-8, TANH-32, SILU-8)	0.00050557	48	bowtie
(SILU-8, TANH-32, SILU-8)	0.00057729	48	bowtie
(ELU-8, SILU-32, ELU-8)	0.00068503	48	bowtie
(SILU-8, TANH-32, SILU-8)	0.00070264	48	bowtie
(ELU-8, ELU-8, SILU-32)	0.00081290	48	funnel
(SILU-8, SILU-32, SILU-8)	0.00082717	48	bowtie
(SILU-32, SILU-8, SILU-8)	0.00087317	48	diamond
(TANH-32, ELU-8, SILU-8)	0.00089392	48	bowtie
(TANH-32, SILU-8, SILU-8)	0.00090197	48	diamond
(SILU-8, SILU-32, ELU-8)	0.00090502	48	bowtie
(SILU-8, SILU-32, SILU-8)	0.00090711	48	bowtie
(TANH-32, ELU-8, SILU-8)	0.00097331	48	bowtie
(TANH-32, SILU-8, ELU-8)	0.00111592	48	diamond
(SILU-32, ELU-8, SILU-8)	0.00121844	48	bowtie
(ELU-8, SILU-8, SILU-32)	0.00122527	48	funnel
(SILU-32, SILU-8, ELU-8)	0.00126189	48	diamond
(ELU-8, SILU-8, SILU-32)	0.00126918	48	funnel
(ELU-8, SILU-8, TANH-32)	0.00127633	48	funnel
(ELU-8, ELU-8, TANH-32)	0.00135926	48	funnel
(SILU-32, SILU-8, ELU-8)	0.00147253	48	diamond
(ELU-8, ELU-8, TANH-32)	0.00200439	48	funnel

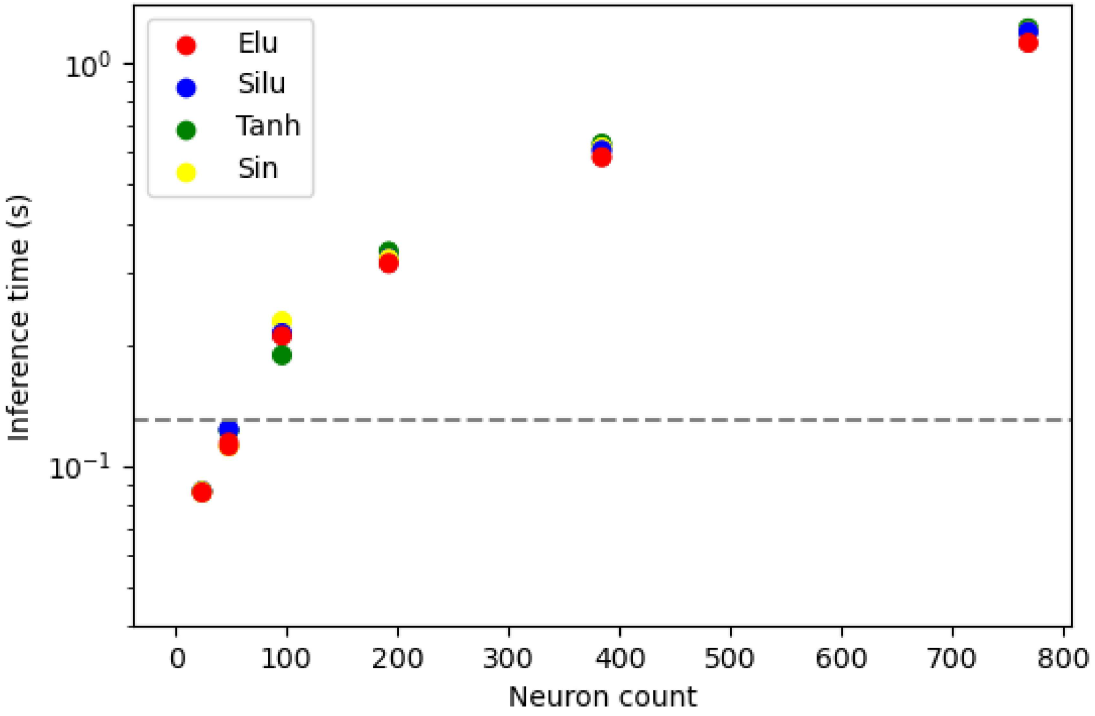

Figure 42 – Model inference performance for multiple models sizes, models have an homogeneous architecture, i.e. with all layers equal, and use one of multiple activation functions. Results shows that the activation function has a very minor effect on inference cost when compared to model size.