



UNIVERSIDADE FEDERAL DE PERNAMBUCO  
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS  
DEPARTAMENTO DE ENGENHARIA MECÂNICA

Henrique Guthierry Ataíde Santos

**SIMULAÇÕES COMPUTACIONAIS DE ENGENHARIA ATRAVÉS DE  
REDES NEURAIS INFORMADAS POR FÍSICA**

Recife  
2024

Henrique Guthierry Ataíde Santos

**SIMULAÇÕES COMPUTACIONAIS DE ENGENHARIA ATRAVÉS DE REDES  
NEURAIS INFORMADAS POR FÍSICA**

Monografia submetida ao Departamento de Engenharia  
Mecânica, da Universidade Federal de Pernambuco  
- UFPE, para conclusão do curso de Graduação em  
Engenharia Mecânica

Orientador: Ramiro Brito Willmersdorf

Recife  
2024

Ficha de identificação da obra elaborada pelo autor,  
através do programa de geração automática do SIB/UFPE

Santos, Henrique Guthierry Ataíde.

Simulações computacionais de engenharia através de redes neurais  
informadas por física / Henrique Guthierry Ataíde Santos. - Recife, 2024.  
64 p. : il., tab.

Orientador(a): Ramiro Brito Willmersdorf

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de  
Pernambuco, Centro de Tecnologia e Geociências, Engenharia Mecânica -  
Bacharelado, 2024.

Inclui referências, apêndices.

1. PINNs. 2. Redes Neurais. 3. Simulações. 4. Transferência de calor. 5.  
Aprendizado de máquina. I. Willmersdorf, Ramiro Brito . (Orientação). II.  
Título.

620 CDD (22.ed.)

HENRIQUE GUTHIERRY ATAÍDE SANTOS

**SIMULAÇÕES COMPUTACIONAIS DE ENGENHARIA ATRAVÉS DE REDES  
NEURAIS INFORMADAS POR FÍSICA**

Trabalho de Conclusão de Curso  
apresentado ao Curso de Engenharia  
Mecânica da Universidade Federal de  
Pernambuco, como requisito parcial para  
obtenção do título de Bacharel em  
Engenharia Mecânica.

Aprovado em: 16/10/2024

**BANCA EXAMINADORA**

---

Prof. Dr. Ramiro Brito Willmersdorf (Orientador)  
Universidade Federal de Pernambuco

---

Prof. Dr. Darlan Karlo Elisiário de Carvalho (Examinador Interno)  
Universidade Federal de Pernambuco

---

Prof. Dr. Francisco Fernando Roberto Pereira (Examinador Interno)  
Universidade Federal de Pernambuco

## **AGRADECIMENTOS**

À senhora Adriana Ataíde Santos e ao senhor Ednaldo José dos Santos, por lutarem incansavelmente por mais de 27 anos pela realização deste sonho; a Vique, pelo seu sincero amor e não ter me deixado desistir desse sonho; a meus irmãos Euller e Débora, sem a qual o amor eu teria me perdido no caminho, e aos três sóis de minha vida, Carente, Godinho e Mococa: dedico a realização deste sonho e minha mais sincera gratidão.

*“Ouse gostar de viver”*  
*(Autore: Vique FC)*

## RESUMO

Este trabalho investiga a aplicação de Redes Neurais Informadas por Física, conhecidas como Physics-Informed Neural Networks (PINNs), para simulações computacionais. PINNs são redes neurais projetadas para simular sistemas físicos e resolver problemas inversos, sem depender de grandes volumes de dados para treinamento. Diante da necessidade de simular sistemas complexos e resolver problemas inversos de forma mais rápida, as PINNs surgem como uma alternativa promissora aos métodos tradicionais de simulação, como o método dos elementos finitos. O objetivo deste estudo foi realizar testes práticos dessa técnica, destacando suas vantagens e limitações. Foram desenvolvidos dois estudos de caso utilizando o *framework* NVIDIA Modulus Sym: um problema de condução térmica unidimensional e outro envolvendo dissipadores de calor em regime tridimensional. Testou-se também versões desses problemas em que as geometrias e condutividades térmicas eram parametrizadas. As PINNs demonstraram capacidade para simular ambos os cenários, ainda que a precisão e tempo computacional tenham desejado a desejar. No entanto, o custo-benefício do treinamento das versões parametrizadas são promissores. Os resultados sugerem que PINNs têm potencial como solução tecnológica desde que continuem a se desenvolver, particularmente em simulações parametrizadas. A seleção adequada de hiperparâmetros e a disponibilidade de recursos computacionais são desafios importante na sua implementação.

**Palavras-chaves:** PINNs, redes neurais, simulações, transferência de calor, aprendizado de máquina.

## ABSTRACT

This work investigates the application of Physics-Informed Neural Networks (PINNs) for computational simulations. PINNs are neural networks designed to simulate physical systems and solve inverse problems without relying on large volumes of training data. Given the need to simulate complex systems and solve inverse problems more quickly, PINNs emerge as a promising alternative to traditional simulation methods, such as the finite element method. The objective of this study was to conduct practical tests of this technique, highlighting its advantages and limitations. Two case studies were developed using the NVIDIA Modulus Sym framework: a one-dimensional heat conduction problem and another involving heat sinks in a three-dimensional regime. Parametrized versions of these problems, where geometries and thermal conductivities were varied, were also tested. The PINNs demonstrated the capability to simulate both scenarios, although the accuracy and computational time left room for improvement. Nevertheless, the cost-benefit of training the parametrized versions appears promising. The results suggest that PINNs have potential as a technological solution, provided they continue to evolve, particularly in parametrized simulations. Proper hyperparameter selection and the availability of computational resources are important challenges in their implementation.

**Key-words:** PINNs, neural networks, simulations, heat transfer, machine learning.



## LISTA DE FIGURAS

Figura 1 – Representação gráfica do algoritmo de um neurônio. . . . .	19
Figura 2 – Arquitetura <i>feedforward</i> ou <i>Multilayer Perceptron</i> . . . . .	20
Figura 3 – Exemplo didático para algoritmo de <i>backpropagation</i> . . . . .	26
Figura 4 – Etapa <i>forward pass</i> do algoritmo de <i>backpropagation</i> . . . . .	28
Figura 5 – Etapa <i>backward pass</i> do algoritmo de <i>backpropagation</i> . . . . .	29
Figura 6 – Fluxograma de treinamento de uma PINN. . . . .	31
Figura 7 – Fluxograma de um script do Modulus. . . . .	40
Figura 8 – Barra composta do problema 1. . . . .	42
Figura 9 – Geometria do problema 2 . . . . .	43
Figura 10 – Parametrização da geometria do dissipador . . . . .	44
Figura 11 – Comparação solução analítica com previsão da PINN. . . . .	45
Figura 12 – Comparação solução analítica com previsão parametrizada da PINN. . . . .	46
Figura 13 – Comparação solução analítica com previsão parametrizada da PINN. . . . .	47
Figura 14 – Estudo de <i>batch</i> e <i>threads versus</i> tempo . . . . .	49
Figura 15 – Comparação soluções do campo de escoamento na direção $X$ . . . . .	51
Figura 16 – Comparação soluções do campo de temperaturas no canal. . . . .	52
Figura 17 – Comparação soluções do campo de temperaturas no dissipador de calor. . . . .	53
Figura 18 – Treinamento problema 2 (Parte fluida). . . . .	53
Figura 19 – Treinamento problema 2 (Parte térmica). . . . .	54
Figura 20 – Comparação da precisão entre modelo parametrizado e não parametrizado. . . . .	54

## LISTA DE TABELAS

Tabela 1 – Funções de ativação clássicas de redes neurais . . . . .	21
Tabela 2 – Funções de saída clássicas de redes neurais . . . . .	22
Tabela 3 – Parâmetros para validação do problema 1 . . . . .	32
Tabela 4 – Intervalo de variação dos parâmetros do problema 2 . . . . .	44
Tabela 5 – Hiperparâmetros usados para treinamento do problema 1 . . . . .	46
Tabela 6 – Parâmetros de validação problema 1 . . . . .	47
Tabela 7 – Configurações do computador local usado na pesquisa. . . . .	48
Tabela 8 – Tempos por interação computador local <i>versus</i> GPU . . . . .	49
Tabela 9 – Hiperparâmetros usados para treinamento do problema 1 . . . . .	50
Tabela 10 – Parâmetros para validação problema 2 . . . . .	50
Tabela 11 – Tempos de treinamento problema 2 . . . . .	50

## LISTA DE ABREVIATURAS E SIGLAS

AdaGrad	Adaptive Gradient Algorithm
Adam	Adaptive Moment Estimation
CPU	Central Processing Unit
DL	Deep Learning
EDP	Equações diferenciais parciais
GPU	Graphical Processing Unit
ML	Machine Learning
MLP	Multilayer Perceptron
MSGD	Mini-batch Stochastic Gradient Descent
PINN	Physical Informed Neural Network
RAM	Random Access Memory
SGD	Stochastic Gradient Descent

## LISTA DE SÍMBOLOS

$a_h$	Valor de pré-ativação de um neurônio
$h$	Saída de um neurônio
$u$	Solução da equação diferencial do problema
$\hat{u}$	Solução aproximada pela rede neural para a equação diferencial
$u_i^*$	Dado do valor conhecido de $u$ no ponto $x_i$
$x$	Entrada da rede neural ou neurônio
$y$	Saída da rede neural
$\hat{y}$	Aproximação da rede neural para uma função qualquer $y(x)$
$y_i^*$	Dado do valor conhecido de $y$ no ponto $x_i$
$B$	Operador diferencial especificando condições iniciais e de contorno
$F$	Operador diferencial não-linear
$L$	Função de perda
$L_B$	Função de perda associada às condições iniciais e de contorno
$L_{dados}$	Função de perda associada aos dados de treinamento
$L_F$	Função de perda associada às equações diferenciais do sistema
$N_d$	Pontos de dados conhecidos para treinamento
$N_B$	Pontos de colocação na fronteira do domínio
$N_C$	Pontos de colocação no interior do domínio
$\overline{W}$	Vetor de pesos associado a um neurônio
$\overline{X}$	Vetor de entrada de um neurônio
$\alpha$	Taxa de aprendizado
$\theta$	Conjunto de parâmetros da rede neural
$\lambda$	Parâmetros da equação diferencial
$\omega_d$	Peso associado a função $L_{dados}$

$\omega_B$	Peso associado a função $L_B$
$\omega_F$	Peso associado a função $L_F$
$\Phi$	Função de ativação
$\Omega$	Domínio do problema físico
$\partial\Omega$	Fronteira do domínio

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>15</b>
1.1	OBJETIVO GERAL	16
1.2	OBJETIVOS ESPECÍFICOS	16
1.3	ORGANIZAÇÃO DESTE TRABALHO	17
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>18</b>
2.1	INTRODUÇÃO AO DEEP LEARNING	18
2.2	MULTILAYER PERCEPTRON	18
2.3	FUNÇÕES DE ATIVAÇÃO	20
2.4	FUNÇÕES DE SAÍDA	22
2.5	TREINAMENTO DE REDES NEURAIS	22
2.6	ALGORITMO DE BACKPROPAGATION	26
2.7	FORMULAÇÃO DE PINNS	29
2.7.1	<i>Problemas Inversos</i>	32
2.7.2	<i>Modelos Parametrizados</i>	33
2.8	EQUAÇÕES DE TRANSFERÊNCIA DE CALOR E BALANÇO DE ENERGIA	33
2.8.1	<i>Difusão de calor unidimensional em regime permanente</i>	36
2.8.2	<i>Equações de Navier-Stokes para fluido newtoniano e incompressível</i>	36
<b>3</b>	<b>METODOLOGIA</b>	<b>38</b>
3.1	BIBLIOTECA UTILIZADA: NVIDIA MODULUS SYM	38
3.2	O AMBIENTE DE DESENVOLVIMENTO	40
3.3	OS PROBLEMAS ABORDADOS	42
3.3.1	<i>Problema 1: Condução de calor unidimensional e de regime permanente</i>	42
3.3.2	<i>Problema 2: Dissipador de calor tridimensional</i>	43
<b>4</b>	<b>RESULTADOS E DISCUSSÃO</b>	<b>45</b>
4.1	PROBLEMA 1: CONDUÇÃO DE CALOR UNIDIMENSIONAL EM REGIME PERMANENTE	45
4.2	PROBLEMA 2: DISSIPADOR DE CALOR	47
4.2.1	<i>Paralelismo computacional</i>	48
4.2.2	<i>CPU x GPU</i>	49
4.2.3	<i>Análise do treinamento</i>	49
<b>5</b>	<b>CONCLUSÕES</b>	<b>55</b>

<b>6</b>	<b>SUGESTÕES PARA FUTUROS ESTUDOS . . . . .</b>	<b>56</b>
	<b>REFERÊNCIAS BIBLIOGRÁFICAS . . . . .</b>	<b>57</b>
	<b>APÊNDICE A – EXEMPLO DIDÁTICO UTILIZANDO MODULUS .</b>	<b>59</b>

## 1 INTRODUÇÃO

Simulações computacionais representam um pilar fundamental para cientistas e engenheiros<sup>1</sup> na análise de sistemas físicos. Métodos tradicionais como elementos finitos, volumes finitos e diferenças finitas demonstraram ser ferramentas valiosas, evoluindo ao longo de décadas para abordar desafios de complexidade crescente, como sistemas multifísicos e alta dimensionalidade.

Ainda assim, há grandes desafios a serem superados. Karniadakis *et al.* (2021) lista alguns: A geração de malha para geometrias complexas, que é muito onerosa; há uma abundância de dados provinda de sensores em sistemas físicos reais, que poderiam ajudar a analisar sistemas complexos, mas sua incorporação aos métodos tradicionais ainda é dificultosa; a resolução de problemas de alta dimensionalidade e parametrizados que costuma ser muito onerosa; o tratamento de problemas inversos, onde se conhece o estado do sistema em dado instante, e deseja-se encontrar as condições que o geraram. A resolução destes costuma ser proibitivamente cara e requer códigos muito complexos.

Assim, a complexidade crescente dos sistemas simulados apresenta desafios que exigem soluções inovadoras. Nesse contexto, um número crescente de pesquisas tem investigado o uso de redes neurais profundas para simulação computacional de fenômenos físicos (CUOMO *et al.*, 2022). As redes neurais profundas, também conhecido como técnicas de *Deep Learning* (DL), tiveram expressivo avanços na última década, graças a avanços no poder computacional e a disponibilidade de grandes volumes de dados para treinamento. Consistem em técnicas para ensinar a sistemas computacionais, por meio de dados e experiência, modelagem de funções altamente complexas. Essas técnicas têm levado a grandes avanços nas áreas de visão computacional, linguagem natural e veículos autônomos (GUO *et al.*, 2021).

No entanto, métodos convencionais de DL requerem um grande volume de dados para treinamento, nem sempre disponível a ê analista, levando a um desempenho subótimo. Por isso um número crescente de pesquisas tem proposto complementar os vieses observacionais (dados) com conhecimento físico-matemático consolidado;

<sup>1</sup> A escolha pela utilização de uma linguagem neutra neste trabalho reflete o compromisso com a inclusão e o respeito às identidades de gênero diversas. A linguagem neutra busca evitar o uso de pronomes e formas gramaticais que reforçam o binarismo de gênero, proporcionando uma comunicação mais ampla e acessível a pessoas que não se identificam com as categorias de "masculino" ou "feminino". É uma medida de acolhimento e visibilidade, fundamental para a evolução das interações humanas e para a promoção da equidade. Mais informações em <https://portal.unila.edu.br/informes/manual-de-linguagem-neutra>



por exemplo, na forma de equações conhecidas do problema. Uma das abordagens mais populares nesse sentido tem sido as *Physics-Informed Neural Networks* (PINNs). Essa abordagem consiste em associar as equações físicas do problema a uma função de custo. A rede então é treinada para minimizar essa função de custo (RAISSI *et al.*, 2019). As PINNs têm sido pesquisadas com grande entusiasmo: Cuomo, *et al* (2022) relata que entre 2018 e 2021, cerca de 2000 papers sobre o assunto foram publicados.

Neste trabalho, investiga-se a fronteira entre o DL e as simulações computacionais, com ênfase nas *Physics-Informed Neural Networks*. Busca-se explorar, compreender e analisar as características de PINNs, evidenciando métodos e ferramentas necessários a sua aplicação, uma vez que PINNs são recentes e pouco compreendidas. Em particular, investiga-se a aplicação por meio do *framework* NVIDIA Modulus Sym<sup>2</sup>, uma biblioteca *Python* que se apresenta como alternativa para desenvolvimento de PINNs.

## 1.1 OBJETIVO GERAL

O objetivo deste estudo é explorar a aplicabilidade das PINNs para realizar simulações computacionais usando o *framework* Modulus: revelar os potenciais da técnica e desvendar os aspectos metodológicos envolvidos na sua implementação.

## 1.2 OBJETIVOS ESPECÍFICOS

Para explorar a aplicação de PINNs delimitou-se os seguintes objetivos específicos:

1. Implementar a biblioteca Modulus Sym nos computadores usados: Estudar e familiarizar-se com a biblioteca Modulus Sym para desenvolvimento de PINNs;
2. Selecionar e modelar dois problemas de transferência de calor de diferentes complexidades;
3. Implementar PINNs nos dois problemas selecionados, com uma acurácia tão boa quanto possível.
4. Realizar uma análise dos desafios, vantagens e desvantagens do método.

<sup>2</sup> Não confundir com o NVIDIA Modulus Core. Atualmente, a NVIDIA segmentou o desenvolvimento de sua biblioteca. O *sym* é focado em engenheiros e não especialistas em inteligência artificial. Já o *Core* é pensado para desenvolvedores de inteligência artificial. Como o foco da pesquisa era o Modulus Sym neste trabalho ele será simplesmente referido como Modulus

### 1.3 ORGANIZAÇÃO DESTE TRABALHO

Este trabalho está organizado da seguinte forma: o capítulo 2 apresentará a fundamentação teórica necessária à compreensão do trabalho, desde conceitos básicos de DL, passando pela construção de PINNs, até a teoria matemática de transferência de calor. No capítulo 3 é descrita a metodologia adotada, incluindo a implementação do Modulus, ferramentas utilizadas e a modelagem matemática dos problemas de transferência de calor. O capítulo 4 contém os resultados e discussão do trabalho. O capítulo 5 apresenta as conclusões. Finalmente, o capítulo 6 fornece sugestões para trabalhos futuros.

## 2 FUNDAMENTAÇÃO TEÓRICA

### 2.1 INTRODUÇÃO AO DEEP LEARNING

Aprendizado de máquina, do inglês *Machine Learning* (ML) refere-se à área preocupada em conferir a computadores a capacidade de aprender com dados e experiência e executar tarefas (SALEHI; BURGUEÑO, 2018). Há várias técnicas de ML, como regressão linear, árvore de decisão, *Support Vector Machine*, para nomear algumas, porém as que vêm ganhando mais notoriedade devido ao seu imenso potencial tecnológico são as técnicas de DL (GUO *et al.*, 2021).

Essencialmente, o DL visa a modelagem de funções matemáticas e para isso subdivide o processamento do problema em unidades menores chamadas neurônios. As entradas da função a ser computada são propagadas dos neurônios de entrada para os neurônios de saída. A propagação da resposta entre neurônios é mediada por parâmetros chamados pesos. O aprendizado ocorre justamente ajustando-se esses pesos (mediante apresentação de dados e experiência). O modelo resultante é chamado de rede neural (AGGARWAL, 2016, p. 2).

As redes neurais têm excelente capacidade de interpolação e extrapolação, dentro e fora do conjunto de dados de treinamento. De fato, já foi demonstrado, que as redes neurais profundas em sua forma mais elementar, podem ser usadas para representar qualquer função matemática contínua (HORNIK *et al.*, 1989). Essa característica tem levado um número cada vez maior de pesquisadoras a utilizá-las para modelar numericamente problemas físicos. A seção seguinte descreve o funcionamento básico de uma rede neural simples.

### 2.2 MULTILAYER PERCEPTRON

Seja  $\bar{X}$  o vetor de entrada de um neurônio e  $\bar{W}$  o vetor contendo uma matriz de coeficientes chamados pesos, um neurônio (Figura 1) em uma rede neural representa a seguinte função (AGGARWAL, 2016, p. 11):

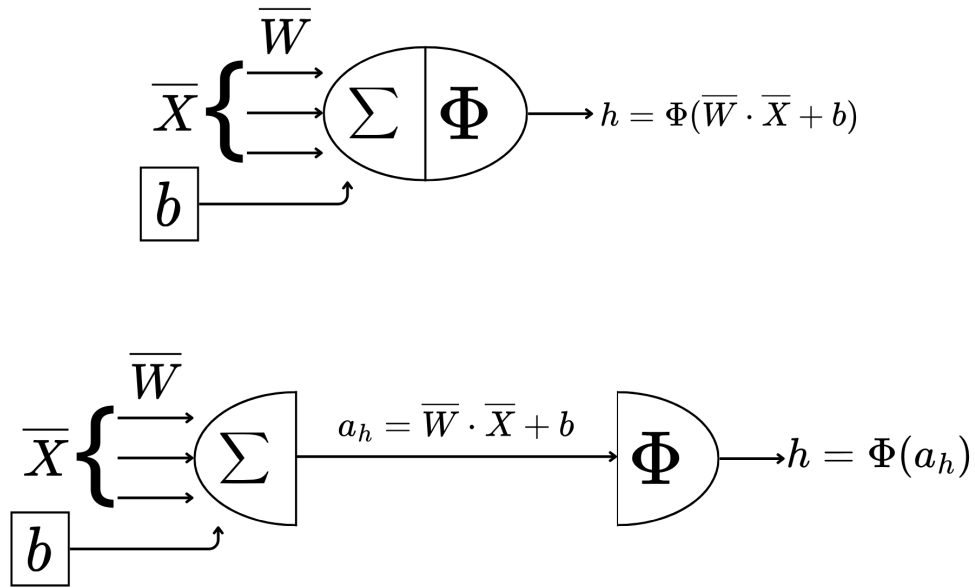
$$h = \Phi(\bar{W} \cdot \bar{X} + b) \quad (2.1)$$

onde  $b$  é um coeficiente chamado *bias* que funciona como uma constante e  $\Phi$  é chamado de função de ativação, que executa uma transformação não-linear sobre seu

argumento. A função de ativação é importante para que a rede neural consiga modelar funções não-lineares (GOODFELLOW *et al.*, 2016, p. 172). O argumento da função de ativação é muitas vezes referido como função de pré-ativação,  $a_h$ :

$$a_h = \overline{W} \cdot \overline{X} + b \quad (2.2)$$

Figura 1 – Representação gráfica do algoritmo de um neurônio.

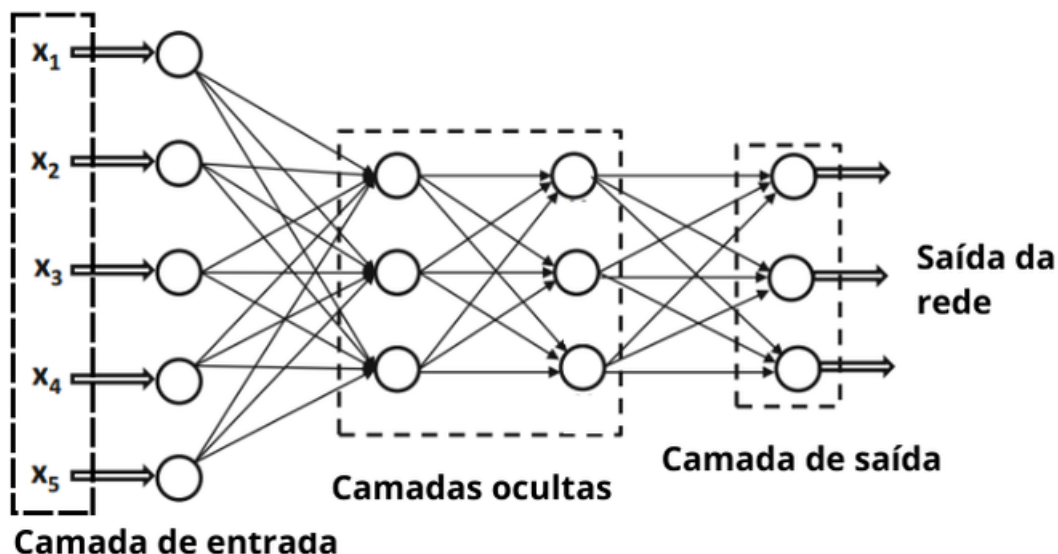


Acima, algoritmo executado pelo neurônio. A baixo, detalhamento e subdivisão desse algoritmo em fase de pré-ativação e ativação. Fonte: Adaptado de Aggarwal (2016).

No arranjo mais simples, cada neurônio se comunica com todos os neurônios da camada anterior e passa sua saída a todos neurônios da camada seguinte. As funções de ativação costumam ser as mesmas para neurônios da mesma camada (VASILEV *et al.*, 2019, p. 43). Essa arquitetura é chamada *Multilayer Perceptron* (MLP), *fully connected* ou *feedforward* e está ilustrada na Figura 2.

Considera-se que a primeira camada da rede são as próprias entradas. A última camada é dita camada de saída. As camadas intermediárias são conhecidas como camadas ocultas. Desde que a rede tenha pelo menos uma camada escondida, a rede neural pode modelar qualquer função contínua, dado um número suficiente de neurônios (HORNICK *et al.*, 1989).

Os pesos dos neurônios devem ser otimizados através de treinamento para modelar a função desejada. As seções seguintes discutem como isso pode ser feito.

Figura 2 – Arquitetura *feedforward* ou *Multilayer Perceptron*

Nessa figura vê-se uma rede *feedforward* com uma camada de entrada, duas camadas ocultas e uma camada de saída. Esse é um dos exemplos mais simples de rede neural.

Fonte: Adaptado de Aggarwal, (2016).

### 2.3 FUNÇÕES DE ATIVAÇÃO

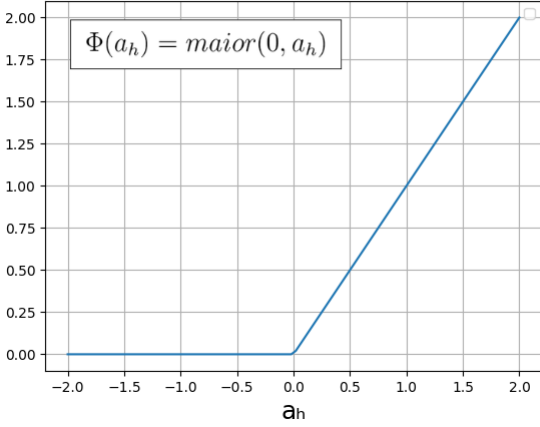
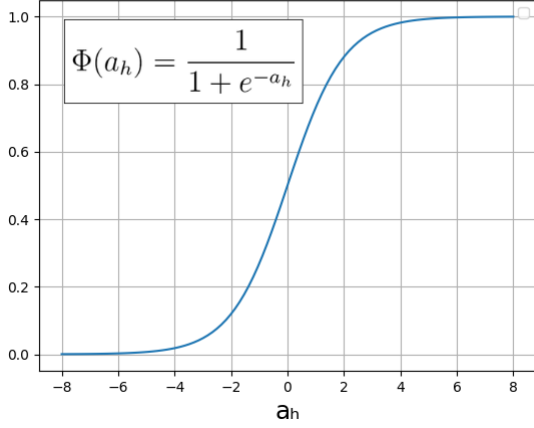
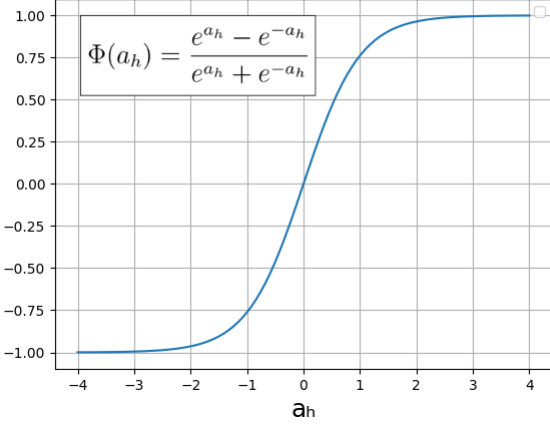
A função de ativação deve ser uma função não-linear. Na prática, apenas algumas funções específicas de eficácia comprovada são utilizadas, com novas funções de ativação sendo um tópico de estudo ativo. A Tabela 1 apresenta as funções de ativação mais comuns segundo Silva (2023).

É importante considerar o comportamento das derivadas das funções de ativação. As funções tanh e sigmóide possuem gradientes quase nulos longe de zero, o que pode levar a dificuldades de convergência no treinamento, que é baseado em métodos de gradiente descendente (GOODFELLOW *et al.*, 2016, p. 191 a 195).

A descontinuidade na derivada da ReLu não é um problema, mas o fato de ela ter derivadas segundas nulas a inviabiliza para PINN's quando a física do problema inclui derivadas dessa ordem. Nesse tipo de rede, tanh é mais usada.

A escolha ótima da função de ativação costuma ser empírica e varia conforme o problema. Ela também é um tópico de pesquisa intenso, com diversas outras funções sendo propostas ao longo dos anos (GOODFELLOW *et al.*, 2016, p. 191). Por exemplo, Cuomo, *et al.* (2022) cita a função Swish-B, que é uma função com um parâmetro livre treinável,  $\beta$ , de forma similar a um peso (Equação 2.3). O autor cita que a função de ativação Swish-B pode ter um desempenho melhor que outras funções clássicas.

Tabela 1 – Funções de ativação clássicas de redes neurais

Função	Características
<p><b>Rectified Linear Unit (ReLU)</b></p>  <p><math>\Phi(a_h) = \text{maior}(0, a_h)</math></p>	<ul style="list-style-type: none"> <li>- Computacionalmente eficiente;</li> <li>- Derivadas não saturam para valores altos de <math>a_h</math>;</li> <li>- Escolha robusta para maioria dos problemas;</li> <li>- Dificuldade no treinamento quando região negativa está sendo sensibilizada;</li> <li>- A derivada segunda é zero em praticamente todos os pontos</li> </ul>
<p><b>Sigmóide</b></p>  <p><math>\Phi(a_h) = \frac{1}{1 + e^{-a_h}}</math></p>	<ul style="list-style-type: none"> <li>- Derivadas saturam para valores distantes de zero;</li> <li>- Importância histórica, mas uso é desencorajado em favor da tanh;</li> <li>- Mais adequada que ReLu para outras arquiteturas de rede como <i>Recurrent Neural Networks</i> e PINNs.</li> </ul>
<p><b>Tangente Hiperbólica (tanh)</b></p>  <p><math>\Phi(a_h) = \frac{e^{a_h} - e^{-a_h}}{e^{a_h} + e^{-a_h}}</math></p>	<p>Treinamento mais eficiente que Sigmóide devido a região mais linear próxima ao zero;</p> <ul style="list-style-type: none"> <li>- Mais adequada que ReLu para outras arquiteturas como <i>Recurrent Neural Networks</i> e PINNs.</li> <li>- Derivadas saturam para valores distantes de zero;</li> </ul>

Fonte: Autor. Informações de Silva (2023) e Goodfellow, *et al.* (2016).

$$\Phi(a_h, \beta) = a_h \cdot \text{sigmóide}(a_h \cdot \beta) \quad (2.3)$$

## 2.4 FUNÇÕES DE SAÍDA

A última camada da rede geralmente é formada por poucos neurônios cuja função é converter a saída para um formato interpretável por humanos. De forma análoga à função de ativação, esses neurônios contêm funções especiais chamadas funções de saída. A Tabela 2 lista as funções de saída mais comuns. Em PINNs é mais comum que se use a função linear para prever as saídas, justamente por se tratar de uma tarefa de regressão.

Tabela 2 – Funções de saída clássicas de redes neurais

Nome	Função	Características
Linear	$y(a_h) = a_h$	Para tarefas de regressão, quando se quer encontrar valores contínuos e reais
Logística sigmóide	$y(a_h) = \frac{1}{1+e^{-a_h}}$	Para tarefas de classificação binária, quando se deseja descobrir a probabilidade de a entrada pertencer a uma classe ou não
<i>Softmax</i>	$y(a_i) = \frac{e^{a_i}}{\sum_k e^{a_j}}$	Para tarefas de classificação multi-classe, quando se deseja descobrir a probabilidade de a entrada pertencer a uma classe i dentro de um conjunto de k-classes. Note que como particularidade essa função normaliza a saída de neurônio i pela soma das saídas dos outros k-neurônios

Fonte: Autor. Informações de Goodfellow, *et al.* (2016).

## 2.5 TREINAMENTO DE REDES NEURAIIS

Costuma-se categorizar o treinamento em supervisionado e não-supervisionado. Supervisionado é aquele na qual detém-se um conjunto de dados e rótulos associados a eles. A rede então é treinada de forma a associar corretamente as entradas aos rótulos apropriados. No não supervisionado somente é fornecido os dados sem os rótulos (SALEHI; BURGUEÑO, 2018). PINNs podem tanto ser consideradas redes treinadas com supervisão ou sem supervisão, a depender se são fornecidos ou não dados com rótulos ao seu treino. (CUOMO *et al.*, 2022).

Sendo as redes neurais aproximadoras de funções, elas diferem destas por um erro. A função objetivo ou função de perda,  $L(\theta)$ , quantifica este erro, dado um o conjunto de parâmetros livres  $\theta^1$ , que normalmente inclui apenas pesos e *bias*, mas podem incluir outros parâmetros. O objetivo do treinamento é encontrar  $\theta^*$  que minimiza  $L(\theta)$  (VASILEV *et al.*, 2019, p. 49), (GOODFELLOW *et al.*, 2016, p. 82):

$$\theta^* = \operatorname{argmin} L(\theta) \quad (2.4)$$

A escolha da função de erro depende da arquitetura e da aplicação em mãos. Para tarefas de regressão, como a executada por PINNs, a função de erro mais comum é o erro quadrático médio, também conhecido como erro L2 (RAISSI *et al.*, 2019):

$$L(\theta) = \frac{1}{N_d} \sum_{i=1}^{N_d} [\hat{y}(x_i) - y^*(x_i)]^2 \quad (2.5)$$

Para realizar o processo de minimização usa-se ostensivamente o método de gradiente descendente (VASILEV *et al.*, 2019, p. 49). Esse método iterativo, baseia-se em aplicar correções aos parâmetros, em direção oposta ao gradiente da função, fazendo com que decresça a cada interação. O algoritmo básico, chamado *Stochastic Gradient Descent* (SGD) é (AGGARWAL, 2016, p. 123):

$$\theta \Leftarrow \theta - \alpha \nabla_{\theta} l(\theta, x_i) \quad (2.6)$$

onde  $l(\theta, x_i)$  é a função de perda aplicada individualmente a um ponto de treinamento  $x_i$  qualquer.

Assim a cada ponto de treinamento os parâmetros recebem uma correção proporcional ao gradiente e a  $\alpha$ , chamado de taxa de aprendizado. A taxa de aprendizado é um hiperparâmetro. Taxas mais altas reduzem as chances da rede estagnar em um mínimo local, mas também podem atrapalhar a convergência.

<sup>1</sup> Normalmente, na modelagem de redes neurais, se distingue parâmetros livres  $\theta$  dos hiperparâmetros. Enquanto os primeiros são definidos pelo processo de treinamento, os últimos são definidos a priori pelo programador e não são ajustados pelo algoritmo de aprendizado. Exemplos de hiperparâmetros incluem: quantidade de camadas, quantidade de neurônios por camadas, arquitetura da rede, função de ativação etc. Note que alguns modelos de DL propõe tornar em parâmetros livres alguns hiperparâmetros, como por exemplo, a função de ativação de cada neurônio.



Na prática as interações do SGD não são calculadas sobre cada ponto de treinamento. Na verdade, a cada interação a perda é agregada sobre um subconjunto dos pontos de treinamento, chamado *mini-batches*. A quantidade  $B$  de pontos a cada interação é chamada tamanho de *mini-batch*. O algoritmo resultante é dito *Mini-Batch Stochastic Gradient Descent* (MSGD):

$$\theta \leftarrow \theta - \alpha \sum_{i=1}^B \nabla_{\theta} l(\theta, x_i) \quad (2.7)$$

A vantagem do MSGD sobre o SGD é que ele é computacionalmente mais eficiente, consome menos memória, ao mesmo tempo que garante certa estabilidade na solução. Normalmente, o tamanho de *mini-Batch* é um hiperparâmetro (AGGARWAL, 2016, p. 122 e 123). Quando todos os pontos de treinamento passam pelo algoritmo de aprendizado, diz-se que o treinamento concluiu uma época. Tipicamente um treinamento tem várias épocas.

Outros algoritmos foram desenvolvidos ao longo dos anos, como forma de compensar os problemas do SGD como a tendência a estagnar em mínimos locais ou dificuldades de convergência. Uma primeira estratégia consiste em adicionar um termo de *momentum* à equação 2.7:

$$\begin{aligned} V &\leftarrow \beta V - \alpha \nabla_{\theta} L(\theta) \\ \theta &\leftarrow \theta + V \end{aligned} \quad (2.8)$$

O termo de *momentum* age como uma memória, intensificando o passo do otimizador em direções que mostraram melhorias mais consistentes ao longo do tempo. Resulta, portanto, em melhor convergência e boa estabilidade (AGGARWAL, 2016, p. 136 e 137).

O algoritmo *Adaptive Gradient Algorithm* (AdaGrad) normaliza o gradiente por um termo quadrático, cujo papel é evitar que o gradiente entre em regiões de grande instabilidade, contribuindo para a solução mais estável. A velocidade de aprendizado, no entanto, tende a se tornar devagar após algumas interações. Seja  $A_i$  um valor agregado para o peso  $w_i$ , o AdaGrad faz:

$$A_i \Leftarrow A_i + \left( \frac{\partial L}{\partial w_i} \right)^2; \forall i$$

A atualização feita no peso correspondente então é feita:

$$w_i \Leftarrow w_i + \frac{\alpha}{\sqrt{A_i}} \left( \frac{\partial L}{\partial w_i} \right) \forall i \quad (2.9)$$

O algoritmo *Adaptive Moment Estimation* (ADAM) é atualmente uma das escolhas mais robustas e a escolha padrão de otimizador. Ele incorpora várias melhorias de outros algoritmos existentes e envolve o cálculo de dois valores intermediários,  $A_i$  e  $F_i$  antes de atualizar os pesos:

$$\begin{aligned} A_i &\Leftarrow \rho A_i + (1 - \rho) \left( \frac{\partial L}{\partial w_i} \right)^2 \forall i \\ F_i &\Leftarrow \rho_f F_i + (1 - \rho) \left( \frac{\partial L}{\partial w_i} \right) \forall i \\ w_i &\Leftarrow w_i - \frac{\alpha}{A_i} F_i \end{aligned} \quad (2.10)$$

Diversos outros algoritmos existem além dos citados, como RMSProp, AdaDelta e *Nesterov Momentum*, embora o ADAM seja o mais comum.

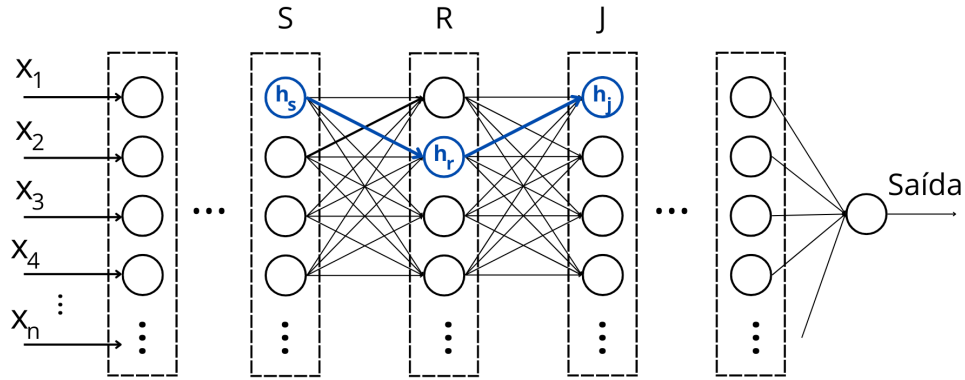
Salienta-se que a minimização da função de perda garante apenas uma boa modelagem para os pontos de treinamento. Frequentemente ocorre que quando testado com dados fora do conjunto de treino ou então tenta-se extrapolar para outras entradas a rede apresente previsões ruins. Por isso, é importante testar a rede neural usando dados fora do conjunto de treinamento. No contexto de aprendizado supervisionado, é praxe se particionar o conjunto de dados em dados para treinamento e dados para testar a rede.

No caso de PINNs, nem sempre almeja-se a capacidade de extrapolação: normalmente ele só é capaz de modelar bem um problema com geometria, condições de contorno e iniciais únicas. Uma forma de contornar essa limitação é parametrizar a geometria/condição de contorno e inclui-la como entrada da rede. Desta forma a PINN pode ser treinada para uma gama maior de problemas simultaneamente.

## 2.6 ALGORITMO DE BACKPROPAGATION

Conforme visto na seção anterior, o processo de aprendizado da rede necessita computar o gradiente da função de perda ( $\nabla_{\theta}$ ). Em redes neurais, que costumam ter de centenas a milhões de parâmetros, calcular todas essas derivadas só é viável graças ao algoritmo de *backpropagation*. Uma explicação simplificada do algoritmo de *backpropagation* é dada a seguir. Uma explicação mais detalhada pode ser encontrada em Aggarwal (2016), Goodfellow, *et al.* (2016) e Vasilev, *et al.* (2019).

Figura 3 – Exemplo didático para algoritmo de *backpropagation*



Nessa figura vê-se uma rede neural genérica com uma quantidade qualquer de camadas e neurônios por camadas. Em destaque estão três camadas consecutivas, camadas S, R e J, e seus respectivos neurônios,  $h_s$ ,  $h_r$  e  $h_j$ . Fonte: Autor.

Suponha-se uma rede neural genérica como a da Figura 3, onde há três camadas sequenciais, S, R e J. Seja  $h_r$  a saída de um neurônio qualquer pertencente a R. Deseja-se computar o gradiente da função de perda em relação a  $w_{(h_s, h_r)}$ , que é o peso que liga o neurônio  $h_s$  a  $h_r$ . De acordo com a regra da cadeia do Cálculo Diferencial, essa derivada é:

$$\frac{\partial L}{\partial w_{(h_s, h_r)}} = \frac{\partial L}{\partial h_r} \cdot \frac{\partial h_r}{\partial w_{(h_s, h_r)}} \quad (2.11)$$

Por sua vez  $\frac{\partial L}{\partial h_r}$  pode ser convenientemente expressa, usando a regra da cadeia para funções multivariáveis como na Equação 2.12:

$$\frac{\partial L}{\partial h_r} = \sum_{j \in J} \frac{\partial L}{\partial h_j} \cdot \frac{\partial h_j}{\partial h_r} \quad (2.12)$$

que é reescrita aqui segundo a notação na Equação 2.13:

$$\Delta(h_r, L) = \sum_{j \in J} \Delta(h_j, L) \cdot \frac{\partial h_j}{\partial h_r} \quad (2.13)$$

Assim a equação 2.11 torna-se:

$$\frac{\partial L}{\partial w_{(h_s, h_r)}} = \frac{\partial h_r}{\partial w_{(h_s, h_r)}} \sum_{j \in J} \Delta(h_j, L) \cdot \frac{\partial h_j}{\partial h_r} \quad (2.14)$$

Os termos  $\frac{\partial h_r}{\partial w_{(h_s, h_r)}}$  e  $\frac{\partial h_j}{\partial h_r}$  são fáceis de calcular. Das equações 1 e 2.2 decorre que eles são:

$$\frac{\partial h_j}{\partial h_r} = \frac{\partial h_j}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \Phi'(x) \cdot w_{(h_r, h_j)} \quad (2.15)$$

,

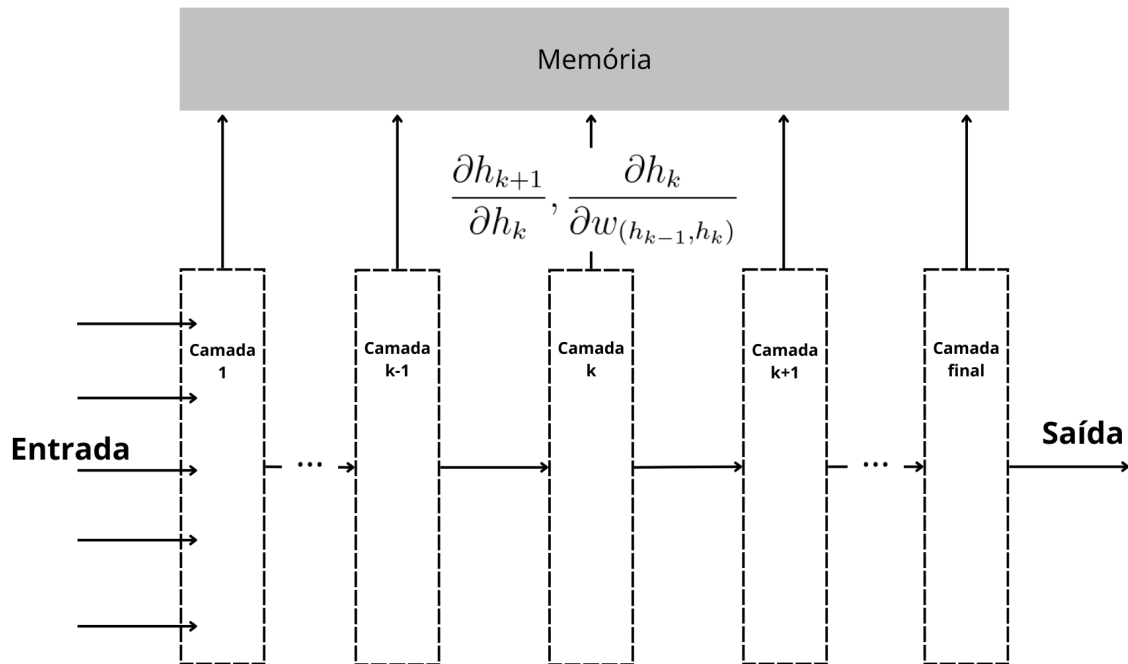
e:

$$\frac{\partial h_r}{\partial w_{h_s, h_r}} = \Phi'(a_h) \cdot h_s \quad (2.16)$$

,

onde  $w_{h_r, h_j}$  é o peso liga o neurônio  $h_r$  a um neurônio qualquer  $h_j$  da camada J. De fato o algoritmo de *backpropagation* calcula os termos das equações 2.15 e 2.16 juntamente com o cálculo das saídas dos neurônios, aproveitando que os valores de  $h$  e  $a_h$  já estão sendo computados. Essa etapa do algoritmo de *backpropagation* é conhecida como *forward pass*, Figura 4.

Já os termos  $\Delta(h_j, L)$ , se fossem esmiuçados a exaustão pela regra da cadeia, gerariam uma quantidade exponencialmente grande de derivadas parciais. Isso porque cada neurônio é uma função multivariável dos neurônios anteriores. Porém, a equação 2.14 sugere que, se as derivada dos neurônios na camada subsequente a  $R$ ,  $\Delta(h_j, L)$

Figura 4 – Etapa *forward pass* do algoritmo de *backpropagation*

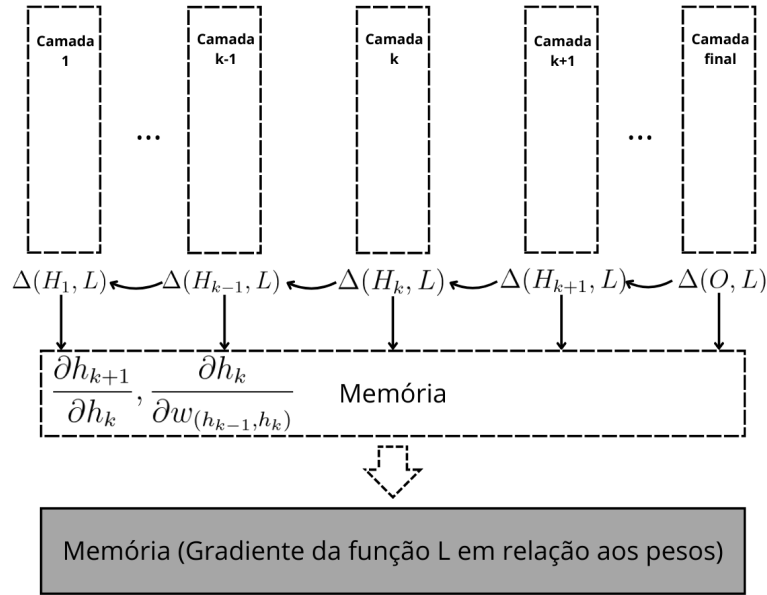
Nesse etapa uma parte dos termos do gradiente são calculados em conjunto com as saídas da rede neural. Esses termos são então armazenados na memória. Fonte: Autor.

forem conhecidas a priori, a derivada de qualquer neurônio da camada  $R$ ,  $\Delta(h_r, L)$  fica simples de calcular. Além disso o mesmo raciocínio pode ser aplicada a camada  $G$ , anterior a  $J$ , e quaisquer camadas anteriores a estas.

O algoritmo de backpropagation usa essa redundância para calcular as derivadas recursivamente das camadas posteriores para as anteriores na etapa conhecida como *backward*. Dai a expressão de que as derivadas se propagam de uma camada a outra. A Figura 5 ilustra etapa.

Apesar de se supor nesse exemplo que as saídas  $h$  são escalares, o mesmo raciocínio pode ser aplicado quando  $h$  são tensores de qualquer dimensão, sem perda de generalidade.

O algoritmo de backpropagation permite não só calcular o gradiente em relação aos pesos, mas também em relação às entradas da rede. Como será visto nas seções seguintes, este é um fato importante que viabiliza as PINNs pois é necessário computar as derivadas em relação às entradas para avaliar o resíduo das equações diferenciais.

Figura 5 – Etapa *backward pass* do algoritmo de *backpropagation*

Nessa etapa do algoritmo de *backpropagation* as derivadas são calculadas recursivamente de trás para frente. As derivadas da camada K+1 serve para simplificar o cálculo da camada K e assim por diante. Fonte: Autor.

## 2.7 FORMULAÇÃO DE PINNS

De acordo com Cuomo, *et al.* (2022), uma PINN é uma rede neural pensada para resolver equações diferenciais da forma geral:

$$\begin{aligned}\mathcal{F}(u(\mathbf{x}); \gamma) &= f(\mathbf{x}) & \mathbf{x} \text{ em } \Omega \\ \mathcal{B}(u(\mathbf{x})) &= g(\mathbf{x}) & \mathbf{x} \text{ em } \partial\Omega\end{aligned}\tag{2.17}$$

onde  $\mathcal{F}$  representa um operador diferencial não-linear,  $\mathbf{x}$  representam um vetor de coordenadas espaço-temporais do problema,  $u$  representa a solução da equação diferencial,  $\gamma$  é um vetor de parâmetros da equação diferencial e  $\mathcal{B}$  também é um operador diferencial representando as condições iniciais e de contorno. Por fim,  $f$  e  $g$  representam funções quaisquer de  $\mathbf{x}$ .

A ideia por trás das PINN's é ser uma aproximação da função  $u$ . Aqui essa aproximação é referida como  $\hat{u}_\theta$ . Para isso, acrescentam termos associados a equação 2.17

(CUOMO *et al.*, 2022) à função de perda:

$$L(\theta) = \omega_{\mathcal{F}} L_{\mathcal{F}}(\theta) + \omega_{\mathcal{B}} L_{\mathcal{B}}(\theta) + \omega_d L_{dados}(\theta) \quad (2.18)$$

Onde  $L_{\mathcal{F}}$  e  $L_{\mathcal{B}}$  são os resíduos da equação 2.17, associados a equação diferencial e às condições de contorno, respectivamente.  $L_{dados}$  corresponde a formulação tradicional de função de perda baseada em dados. Os coeficientes  $\omega_{\mathcal{F}}$ ,  $\omega_{\mathcal{B}}$  e  $\omega_d$  são pesos que definem a importância relativa de cada termo na equação 2.18.

Uma formulação comum, mas não a única, para os termos da equação 2.18 é a seguinte:

$$L_{\mathcal{F}} = \frac{1}{N_c} \sum_{i=1}^{N_c} (\mathcal{F}(\hat{u}_{\theta}(x_i)) - f(x_i))^2 \quad (2.19)$$

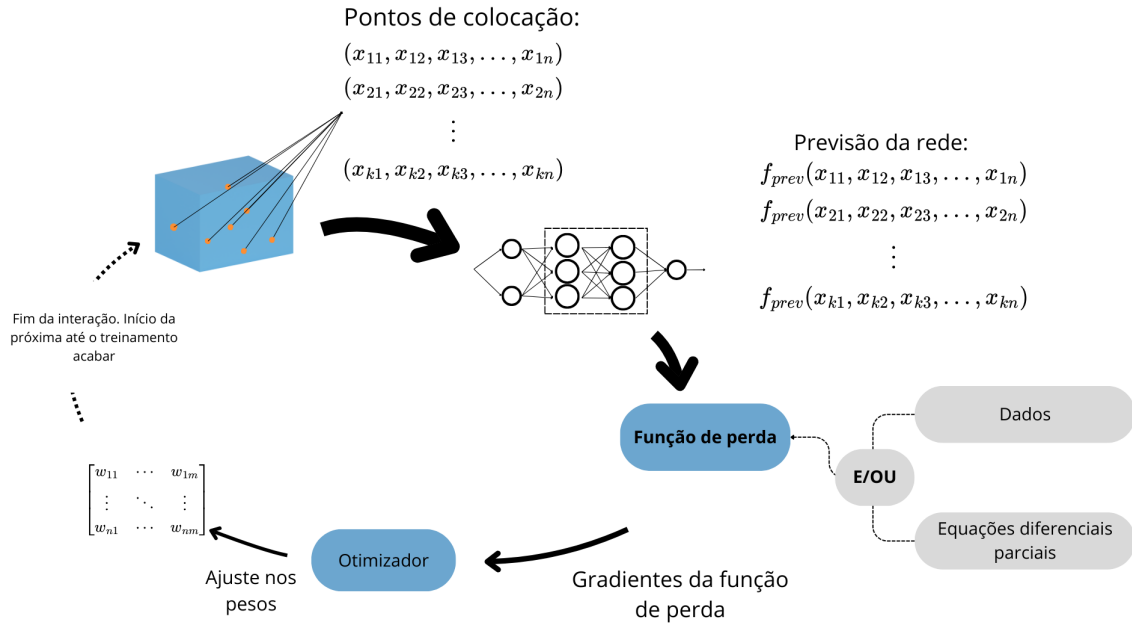
$$L_{\mathcal{B}} = \frac{1}{N_b} \sum_{i=1}^{N_b} (\mathcal{B}(\hat{u}_{\theta}(x_i)) - g(x_i))^2 \quad (2.20)$$

$$L_{dados} = \frac{1}{N_d} \sum_{i=1}^{N_b} (\mathcal{B}(\hat{u}_{\theta}(x_i)) - u^*(x_i))^2 \quad (2.21)$$

onde  $N_c$  e  $N_b$  são chamados pontos de colocação, que nada mais são que pontos do domínio escolhidos para realizar o treinamento. Enquanto  $N_c$  são pontos do interior do domínio,  $N_b$  são pontos do contorno. Os pontos de colocação podem ser selecionados de maneira aleatória ou serem uniformemente distribuídos pelo domínio (NVIDIA MODULUS TEAM, 2023).

Os pontos de colocação podem ser constituídos de coordenadas espaciais, temporais e/ou outros parâmetros do problema. Para poder gerar as coordenadas espaciais, *softwares* como o Modulus utilizam a geometria do problema, na forma de arquivos .STL por exemplo, e coletam pontos que sejam pertencentes a geometria. Já os dados  $u^*(x_i)$  podem ser tanto dados coletados de sistemas reais, como a partir de dados sintéticos, provenientes de simulações por métodos tradicionais. A Figura 6 mostra o fluxograma de treinamento de uma PINN.

Figura 6 – Fluxograma de treinamento de uma PINN.



Nesta figura vê-se o fluxograma de treinamento de PINNs. Primeiro são tomados pontos de colocação no domínio (em laranja); as coordenadas desses pontos são inseridas na rede neural que tenta prever o valor da função nesses pontos. A previsão então é comparada com os dados e com as equações matemáticas. O gradiente do erro é inserido no otimizador que atualiza os pesos da rede. Fonte: autor.

De acordo com as equações 2.19 e a função de perda de PINN precisa das derivadas de  $\hat{u}_\theta$  em relação às entradas,  $x$ . De forma similar ao cálculo do gradiente dos pesos, as PINNs usam o algoritmo de *backpropagation* e autodiferenciação para computar tais derivadas (RAISSI *et al.*, 2019).

A equação de perda mostrada permite às PINNs incorporar tanto dados rotulados como equações físicas do problema em mãos. Isso reduz a quantidade de dados necessários e melhora a convergência. Pode-se até mesmo realizar o treinamento sem dado algum (caso em que  $\omega_d$  é zero). O melhor caso de uso de PINNs, no entanto, é um meio termo entre conhecimento das equações físicas e alguma disponibilidade de dados (KARNIADASKIS, *et al.*, 2021).

Outra vantagem é que é possível definir a importância de cada parcela da equação 2.18 para o treinamento. Se os dados possuem pouca confiabilidade ou têm muito ruído, por exemplo,  $\omega_d$  pode assumir valores baixos. Assim a função de perda é altamente customizável às diferentes fontes de conhecimento prévio.

É possível até mesmo dividir  $L_{dados}$  em vários subtermos, cada um referente a



uma fonte diferente de dados e atribuir pesos relativos a cada uma (KARNIADAKIS *et al.*, 2021). A Tabela 3 traz um comparativo entre PINNs e métodos tradicionais de simulação.

Tabela 3 – Parâmetros para validação do problema 1

<b>Característica</b>	<b>Métodos tradicionais</b>	<b>PINN</b>
Processo de simulação	Resolvem as equações do problema por meio da discretização do domínio	Treinam uma rede neural para aproximar a relação entrada-saída do problema. Para isso usam as equações conhecidas do problema na função de erro.
Necessita de malha	Sim	Não
Tratamento de problemas inversos	Muito difícil e complexo de se implementar	Naturalmente capaz de resolver esses problemas. Com algumas poucas modificações o mesmo código utilizado para problemas diretos pode ser utilizado para problemas inversos
Modelos parametrizados	Sim	Sim
Uso de dados	Difícil utilizar dados reais para aprimorar resultados	PINNs por serem redes neurais são naturalmente capazes de utilizar dados para treinamento e tarefas de regressão

Fonte: autor.

### 2.7.1 Problemas Inversos

Até aqui, foi discutido a capacidade de PINNs resolverem problemas diretos, isto é, achar a solução de equações diferenciais. No entanto, PINN's desde sua concepção, foram pensados para também resolver problemas inversos. Essa classe de problemas consiste em, dada uma série de observações de um sistema, encontrar os fatores causais que o produziram (NVIDIA MODULUS TEAM, 2023). Um problema inverso pode consistir em encontrar as condições de contorno ou descobrir alguns parâmetros

físicos desconhecidos. Matematicamente falando, conhecidos  $u(x)$  e a forma geral de  $\mathcal{F}$  e  $\mathcal{B}$  ache  $\gamma$ , tal que (CUOMO *et al.*, 2022):

$$\begin{aligned}\mathcal{F}(u(\mathbf{x}); \gamma) &= f(\mathbf{x}) & \mathbf{x} \text{ em } \Omega \\ \mathcal{B}(u(\mathbf{x})) &= g(\mathbf{x}) & \mathbf{x} \text{ em } \partial\Omega\end{aligned}\tag{2.17}$$

Para que a rede consiga resolver o problema inverso  $\gamma$  é incorporado aos parâmetros livres (de forma similar aos pesos) e é aproximado através do treinamento (RAISSI *et al.*, 2019).

Nesse tipo de problema, o conhecimento de  $u(\mathbf{x})$  é modelado por uma série de observações (dados),  $u^*(x_i)$ . A presença de dados é, nesse caso, obrigatória, ao contrário do problema direto. A forma geral das equações diferenciais é implementada na função de erro para que o treinamento convirja mais rápido e não precise de tantos dados.

### 2.7.2 Modelos Parametrizados

PINNs podem ser usados para criar modelos parametrizados, com ou sem auxílio de dados observacionais. Para isso, precisam apenas que os parâmetros  $\lambda$  que desejam ser estudados sejam incluídos como entradas da rede neural  $\mathbf{x}$ . Ao fazer isso, o treinamento irá otimizar a rede para prever diferentes cenários com geometrias e parâmetros diferentes. O Modulus por exemplo, pode ser usado para criar modelos que parametrizam geometria, condições de contorno/inicial ou parâmetros da equação diferencial. No entanto a generalidade do modelo só vai até onde o intervalo de parametrização permitir. Parâmetros discretos (por exemplo, zero ou um, em contraste com "entre zero e um") apresentam dificuldade de convergência (NVIDIA MODULUS TEAM, 2023).

## 2.8 EQUAÇÕES DE TRANSFERÊNCIA DE CALOR E BALANÇO DE ENERGIA

No decorrer do trabalho escolheu-se problemas de transferência de calor como estudo de caso, sendo assim, uma base teórica acerca desses problemas é fornecida a seguir.

A condução de calor é o processo de transferência de energia térmica entre partículas mais energéticas de uma substância para partículas vizinhas adjacentes menos energéticas, como resultado da interação entre elas. Um exemplo é o aquecimento de

uma lata de bebida gelada em um ambiente quente como resultado da transferência de calor do ambiente para a bebida por condução através da lata (ÇENGEL; GHAJAR, 2012, p. 9, 17 e 18).

A equação 2.22, conhecida como lei de Fourier, descreve a transferência de calor por unidade de área,  $q$ , através de condução (BERGMAN *et al.*, 2014):

$$q = -k \left( \mathbf{i} \frac{\partial T}{\partial x} + \mathbf{j} \frac{\partial T}{\partial y} + \mathbf{k} \frac{\partial T}{\partial z} \right) = -k \nabla T \quad (2.22)$$

onde  $T$  é a distribuição de temperatura do corpo e  $k$  é a condutividade térmica. A condutividade térmica é a taxa de transferência de calor de um material, medida em  $J/smK$  (ÇENGEL; GHAJAR, 2012, p. 20).

Suponha um sistema em que as seguintes hipóteses são válidas (WHITE, 2011, p. 254 a 256):

1. O meio é contínuo;
2. A transferência de calor por radiação é desprezível (ela só ocorre por condução);
3. A geração de calor por reação química ou nuclear é desprezível.

então a seguinte equação estabelece a conservação de energia dentro de um elemento de volume infinitesimal. Ela é obtida quando aplicado a primeira lei da termodinâmica (conservação de energia) e a lei de Fourier a esse volume:

$$\rho \frac{d\hat{u}}{dt} + p(\nabla \cdot V) = \nabla \cdot (k \nabla T) + \Phi \quad (2.23)$$

onde  $\hat{u}$  representa a energia interna do sistema,  $p$  é a pressão,  $V$  é o campo vetorial de velocidades e  $\Phi$  é a função de dissipação viscosa, que representa o calor dissipado através do atrito interno do meio.

A equação 2.23 quando resolvida permite o conhecimento da distribuição de temperatura de um sistema, seja sólido, líquido ou gasoso, desde que as hipóteses estabelecidas sejam aplicáveis. Ela pode ser ainda mais simplificada se forem considerada as seguintes aproximações:

- Densidade  $\rho$  constante. Válido para escoamento incompressível ou em sólidos;
- Condutividade térmica  $k$  aproximadamente constante. Válido quando  $k$  varia pouco com a temperatura ou a temperatura não varia muito;
- Viscosidade constante. Válido para fluido newtoniano com pouca variação de temperatura;
- $d \cap u \approx c_v dT$ , onde  $c_v$  é o calor específico a volume constante. Nesse caso considera-se  $c_v$  aproximadamente constante por haver pouca variação de temperatura;

Nesse caso a equação reduz a:

$$\rho c_v \frac{dT}{dt} = k \nabla^2 T + \Phi$$

O termo  $dT/dt$  pode ser expandido:

$$\frac{dT}{dt} = \frac{\partial T}{\partial t} + u \frac{\partial T}{\partial x} + v \frac{\partial T}{\partial y} + w \frac{\partial T}{\partial z} = \frac{\partial T}{\partial t} + \mathbf{V} \cdot \nabla T$$

Resultando finalmente em:

$$\rho c_v \left( \frac{\partial T}{\partial t} + \mathbf{V} \cdot \nabla T \right) = k \nabla^2 T + \Phi \quad (2.24)$$

Que é a equação usada neste trabalho para modelar transferência de calor em fluidos. É importante notar que no caso dos sólidos em que não há escoamento ( $\mathbf{V}$  nulo), a equação reduz para a equação de difusividade térmica:

$$\rho c_v \frac{\partial T}{\partial t} = k \nabla^2 T \quad (2.25)$$

Que será a equação de preferência para transferência de calor em sólidos.

### 2.8.1 Difusão de calor unidimensional em regime permanente

Considerando a equação 2.25 e todas as hipóteses necessárias, para o caso em que a transferência de calor ocorre apenas em uma dimensão (devido a por exemplo as dimensões do objeto ou a isolamento térmico das faces), e em regime permanente, tem-se a seguinte equação:

$$k \frac{d^2 T}{dx^2} = 0 \quad (2.26)$$

### 2.8.2 Equações de Navier-Stokes para fluido newtoniano e incompressível

Para se determinar problemas de transferência de calor em que há transferência através de um meio fluido também é importante se determinar o campo de velocidades, uma vez que os deslocamentos das massas de fluido contribuem para a parcela advectiva da equação de energia ( $V \cdot \nabla T$ ).

Considere um fluido newtoniano e incompressível. O balanço da quantidade de movimento em um volume de controle infinitesimal fornece (FOX *et al.*, 2014, cap. 5, seção 4):

$$\begin{aligned} \rho \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} \right) &= \rho g_x - \frac{\partial p}{\partial x} + \mu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \\ \rho \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} \right) &= \rho g_x - \frac{\partial p}{\partial x} + \mu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \\ \rho \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} \right) &= \rho g_x - \frac{\partial p}{\partial x} + \mu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \end{aligned} \quad (2.27)$$

Para completar as equações é necessário considerar o balanço do fluxo de massa dentro do volume, o que resulta na equação da continuidade para fluidos incompressíveis:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \quad (2.28)$$

A equação 2.27 em conjunto com 2.28 são formas simplificadas das equações

de Navier-Stokes para escoamentos newtonianos e incompressíveis e sua solução equivale a determinar o campo de velocidades do escoamento.

### 3 METODOLOGIA

A metodologia consistiu em primeiro implementar e aprender a utilizar a biblioteca Modulus Sym. Após isso tomou-se dois problemas de transferência de calor para estudo de caso: o primeiro foi de uma barra, composta de três materiais e com condução unidimensional e em regime permanente. O segundo foi um problema tridimensional e em regime permanente de um dissipador de calor submetido a escoamento dentro um canal. Ambos os problemas foram tomados de exemplos prontos da biblioteca do Modulus, que vinham tanto com códigos quanto com hiperparâmetros prontos para treinamento da PINN, sem o uso de dados <sup>1</sup>. Isso facilitou o desenvolvimento do trabalho, sendo necessário apenas modificações menores no código e nos hiperparâmetros.

Para avaliar a eficácia de PINNs como modelos substitutos, criaram-se versões parametrizadas e não parametrizadas dos problemas 1 e 2. A versão parametrizada dispunha dos mesmos hiperparâmetros que a não parametrizada. O desenvolvimento dos modelos foi desenvolvido em um regime sem dados (não-supervisionado), apenas com as equações diferenciais do problema.

Nas próximas seções são dado mais detalhes de como foi feita a implementação do *Modulus*, os problemas avaliados e os testes feitos.

#### 3.1 BIBLIOTECA UTILIZADA: NVIDIA MODULUS SYM

O NVIDIA Modulus Sym é uma dentre as diversas bibliotecas de programação para implementação de PINNs. É uma biblioteca escrita em linguagem de programação Python de código aberto para construção e treinamento de modelos físicos com base em ML, com bastante foco em PINNs. Na data de escrita deste trabalho a versão corrente do Modulus é a 24.09.

O Modulus é construído em cima do PyTorch, uma consagrada biblioteca para DL. O Modulus propõe-se a fornecer uma interface amigável, incorporando métodos do estado da arte de DL, com robustez e integração com as *Graphical Processing Units* (GPUs) da NVIDIA. Entre as funcionalidades dele se encontram (NVIDIA MODULUS TEAM, 2023):

- Suporte a *constructive solid geometry*, ou seja, é possível construir a geometria do

---

<sup>1</sup> Os problemas utilizados e outros exemplos podem ser encontrados em: <https://github.com/NVIDIA/modulus-sym/tree/main/examples>

problema por meio de primitivas geométricas e operações *booleanas* entre elas;

- Suporte a *Tesselated Geometry*. Aceita arquivos .STL para gerar geometrias;
- Possibilidade de parametrizar equações diferenciais, de contorno e geometrias;
- Facilidade de implementação de equações diferenciais novas, além de já contar com diversas equações famosas para uso;
- Facilidade de integrar diversas equações para modelos multifísicos;
- Facilidade de configuração e experimentação de hiperparâmetros, por meio de uso da biblioteca Hydra;
- Integração com TensorBoard, um *software* para monitoramento do treinamento e resultados da rede neural;
- Permite a exportação das saídas para arquivos *Visualization Toolkit* (VTK), que permite realizar o pós-processamento em ferramentas como o *Paraview*.

Apesar do Modulus utilizar os algoritmos descritos na seção 2.7 para modelar PINNs internamente, a maneira como os *scripts* são construídos são bem diferentes do que os algoritmos sugerem. É importante, portanto, explicar alguns conceitos chaves do Modulus:

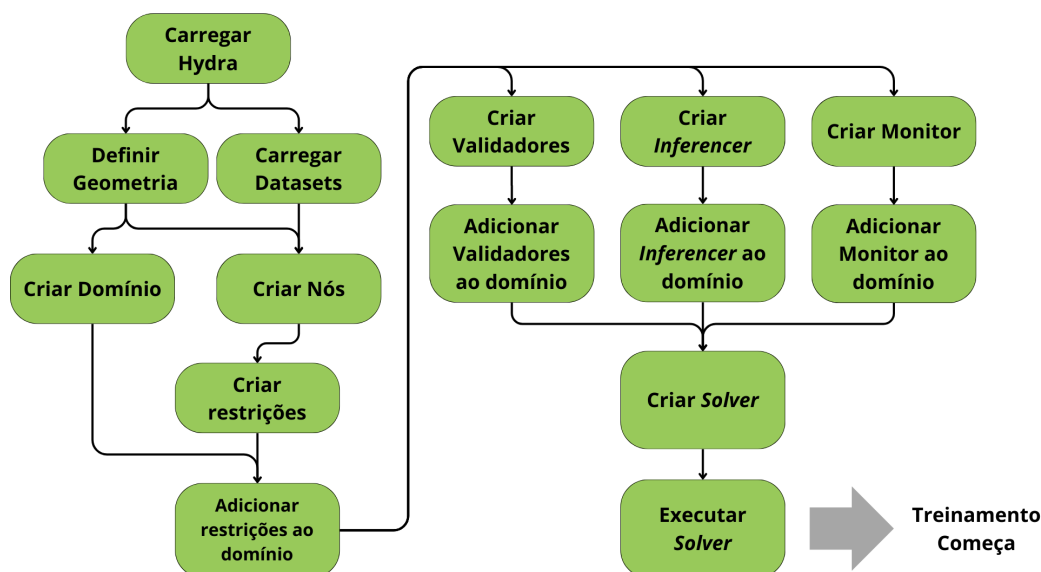
- **Hydra:** essa é uma biblioteca incluída no Modulus. Ela é responsável por carregar e gerenciar hiperparâmetros do modelo. Permite rápida alteração e controle dos hiperparâmetros;
- **Restrições:** ao invés de lidar diretamente com a criação da função de perda, o usuário cria restrições, que nada mais são que imposições aos valores das entradas/saídas da rede neural. As restrições são usadas para especificar quais equações, condições de contorno e/ou conjunto de dados de treinamento a rede neural deve aproximar. As restrições precisam ser alimentadas com o um grafo de nós;
- **Nós:** Os nós podem ser redes neurais ou equações diferenciais. O Modulus agrupa essas duas estruturas em um grafo e identifica automaticamente a interconexão entre diferentes redes e equações. Usa-se esse grafo para criação de restrições, computação das derivadas e, de forma geral, estabelecer coerência entre as entradas e saídas de cada nó;



- **Validadores:** são responsáveis por calcular o erro relativo da rede em relação a um valor de referência;
- **Inferencers:** permitem o cálculo de grandezas auxiliares a partir das saídas da rede no pós-processamento;
- **Monitor:** monitoram determinadas grandezas e as mostram durante o treinamento;
- **Domínio:** um objeto que gerencia restrições, validadores, inferenciadores e monitores;
- **Solver:** Responsável por executar o treinamento.

A Figura 7 mostra o fluxograma de um *script* do Modulus para criação e treinamento de uma PINN. Para entender melhor o funcionamento do Modulus recomenda-se checar o apêndice A que contém um exemplo didático de um script.

Figura 7 – Fluxograma de um script do Modulus.



Acima, o fluxograma de um típico script escrito no Modulus. O modulus primeiro carrega os hiperparâmetros e outras configurações através da biblioteca Hydra. Em seguida dados da geometria e para treinamento são carregados. Ocorre a criação dos nós, que são as próprias redes neurais. (2023).

### 3.2 O AMBIENTE DE DESENVOLVIMENTO

Há duas opções para instalação do Modulus:

1. Uso de *containers*: *Containers* nada mais são que processos que emulam um ambiente isolado do sistema computacional do *host*. Esses ambientes já dispõem de todas as dependências necessárias para desenvolvimento da aplicação, o que elimina a necessidade de instalar essas dependências manualmente, além de evita conflito entre versões diferentes que estejam instaladas (DOCKER INC, );
2. Instalar a partir do código fontes: consiste em baixar e instalar na máquina o Modulus e todas as dependências necessárias;

Optou-se pelo uso do desenvolvimento em *containers* que acelerou o trabalho e permitiu maior consistência no desenvolvimento.

Para treinamento das redes neurais foi utilizado um computador com 20 *Central Processing Units* (CPUs). Como não se dispunha de *Graphical Processing Units* (GPUs) e o treinamento de redes neurais é altamente mais eficiente quando operado nesse *chip* de computação, foi utilizado também o Google Colaboratory. O Google Colaboratory é um serviço de computação na nuvem que dispõe GPUs de alto desempenho a seus usuários, além de ter uma interface amigável para desenvolvimento de códigos. Apesar de otimizar o tempo de treinamento, o Google Colaboratory limita o tempo de uso das GPU's a seus usuários não pagantes e o custo para acesso prolongado é caro. Assim ambos os sistemas foram utilizados, as CPUs para testes e trabalhos menores e as GPUs do colaboratory para treinamentos mais difíceis.

Não obstante, outras ferramentas permitiram o melhor andamento do trabalho:

- **Docker**: *Software* para geração de *containers* e imagens de *containers*;
- **VSCode**: Um *software* de *Integrated Development Environment* (IDE), útil para o desenvolvimento ágil e organizado de grandes projetos de programação. Fornece uma interface e diversas ferramentas que facilita a escrita e gerenciamento de código;
- **Git**: *Software* de gerenciamento de versões de código, útil para controlar versões de código, trabalhar de maneira organizada em projetos de equipe;
- **Tensorboard**: *Software* para monitoramento e pós-processamento de redes neurais. O Modulus tem integração com este e gera gráficos automaticamente ao se executar o treinamento, permite acompanhar a acurácia do modelo;
- **Paraview**: *Software* para pós-processamento de dados, principalmente de simulações numéricas, de arquivos VTK. Vem com diversos filtros para tratamento e

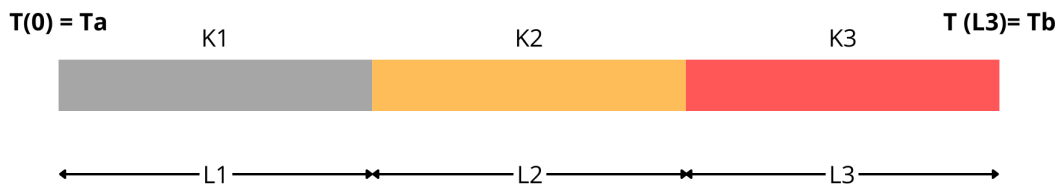
visualização de dados. Foi utilizado para conseguir visualizar os resultados do treinamento dos PINNs.

### 3.3 OS PROBLEMAS ABORDADOS

#### 3.3.1 Problema 1: Condução de calor unidimensional e de regime permanente

Nesse problema busca-se determinar a distribuição de temperaturas e fluxo de calor em uma barra composta de três materiais diferentes, Figura 8. A barra encontra-se isolada ao longo de seu comprimento e suas extremidades são mantidas a temperaturas constantes. Além disso foi dado tempo suficiente para sua temperatura estabilizar e alcançar o regime permanente. Nessas condições é válido o uso da Equação 2.26.

Figura 8 – Barra composta do problema 1.



Barra composta de três materiais com condutividades térmicas diferentes, K1, K2 e K3. Os comprimentos de cada porção da barra são, respectivamente, L1, L2 e L3. As temperaturas nas extremidades são Ta e Tb. Fonte: Autor.

A equação diferencial e as condições de contorno são as seguintes:

$$\begin{aligned}
 k_1 \frac{d^2 T_1}{dx^2} &= 0 \quad \text{Para: } 0 \leq x < L_1 \\
 k_2 \frac{d^2 T_2}{dx^2} &= 0 \quad \text{Para: } L_1 \leq x < L_1 + L_2 \\
 k_3 \frac{d^2 T_3}{dx^2} &= 0 \quad \text{Para: } L_1 + L_2 \leq x \leq L_1 + L_2 + L_3
 \end{aligned} \tag{3.1}$$

Com as seguintes condições de contorno:

Temperaturas fixas nas extremidades:  $T_1(0) = 0$

$$T_2(L_1 + L_2 + L_3) = 200$$

Temperatura igual nas interfaces:  $T_1(L_1) = T_2(L_1)$

$$T_2(L_1 + L_2) = T_3(L_1 + L_2)$$

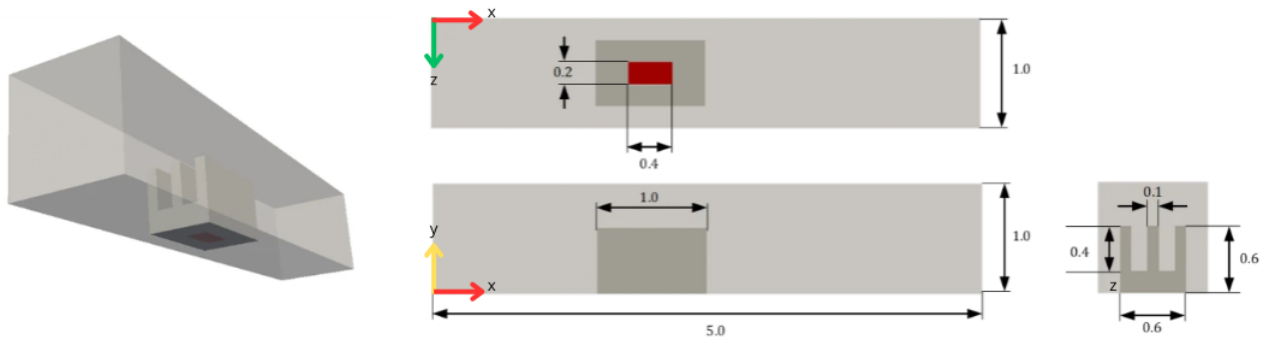
$$\text{Fluxo de calor constante: } k_1 \frac{dT_1}{dx} = k_2 \frac{dT_2}{dx} = k_3 \frac{dT_3}{dx}$$

Na versão parametrizada do modelo, parametrizaram-se os comprimentos  $L_1$ ,  $L_2$  e  $L_3$ , podendo variar de 1 a 25; e também as condutividades térmicas  $k_1$ ,  $k_2$  e  $k_3$ , podendo variar de 0,1 a 1. Para avaliar a acurácia do modelo, tanto parametrizado como não parametrizado, foi utilizada a solução analítica do problema.

### 3.3.2 Problema 2: Dissipador de calor tridimensional

O problema 2 consistia em um dissipador de calor imerso em um escoamento forçado dentro de um tubo, Figura 9. O dissipador está assentado sobre uma fonte de calor de  $0,2 \times 0,4 \text{ m}$  que gera um gradiente de temperatura de  $360 \text{ K/m}$  na direção normal. As paredes do tubo encontram-se isolados. A entrada encontra-se a temperatura de  $273,15 \text{ K}$  e a velocidade do escoamento é  $1 \text{ m/s}$ , uniforme por toda a seção. A pressão de saída encontra-se a  $0 \text{ Pa}$ . Esse mesmo problema, de forma bidimensional, foi primeiro apresentado por Cheung e See (2021).

Figura 9 – Geometria do problema 2



Dissipador de calor dentro de um canal com escoamento do problema 2. Em vermelho a fonte de calor sobre a qual está o dissipador. Dimensões em metros. Fonte: Adaptado de NVIDIA MODULUS TEAM (2023).

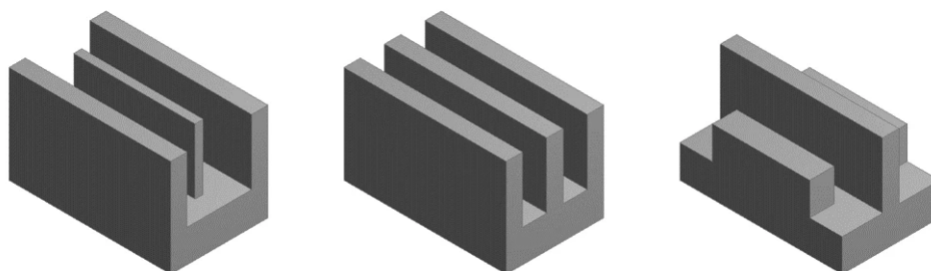
Para tal problema as Equações 2.24, 2.27 e 2.28 são aplicáveis.

Considera-se também que a dissipação viscosa por atrito,  $\Phi$  é desprezível e que a variação de temperatura não é suficiente para alterar a densidade ou escoamento do fluido. Desta forma as equações 2.27 e 2.28 ficam desacopladas da Equação 2.24.

O desacoplamento pode então ser usado para facilitar o treinamento. O modelo implantado utiliza duas redes neurais diferentes, uma para modelar o escoamento e outra a parte térmica. A rede do escoamento é treinada primeiro, e a rede térmica é treinada depois, usando os resultados da rede de escoamento.

A versão parametrizada do problema incluiu como parâmetros a espessura e altura das aletas do dissipador, Figura 10, e o intervalo de parâmetros encontra-se na Tabela 6. Para validação foram usados os resultados de uma simulação feita no *software* OpenFOAM<sup>2</sup>, com valores de parâmetros também especificados na Tabela 6.

Figura 10 – Parametrização da geometria do dissipador



Exemplos de configuração gerados pela parametrização das aletas do dissipador de calor. Fonte: NVIDIA MODULUS TEAM (2023).

Tabela 4 – Intervalo de variação dos parâmetros do problema 2

Parâmetro	Intervalo de variação (mm)
Espessura das aletas laterais	50 a 150
Altura das aletas laterais	0,0 a 600
Comprimento das aletas laterais	500 a 1000
Espessura da aleta central	50 a 150
Altura da aleta central	0,0 a 600
Comprimento da aleta central	500 a 1000

Nesta tabela consta os intervalos de parâmetros para a qual a rede foi treinada. Fonte: autor

<sup>2</sup> Dados dessa simulação fornecidos pela NVIDIA em <[https://catalog.ngc.nvidia.com/orgs/nvidia/teams/modulus/resources/modulus\\_sym\\_examples\\_supplemental\\_materials](https://catalog.ngc.nvidia.com/orgs/nvidia/teams/modulus/resources/modulus_sym_examples_supplemental_materials)>

## 4 RESULTADOS E DISCUSSÃO

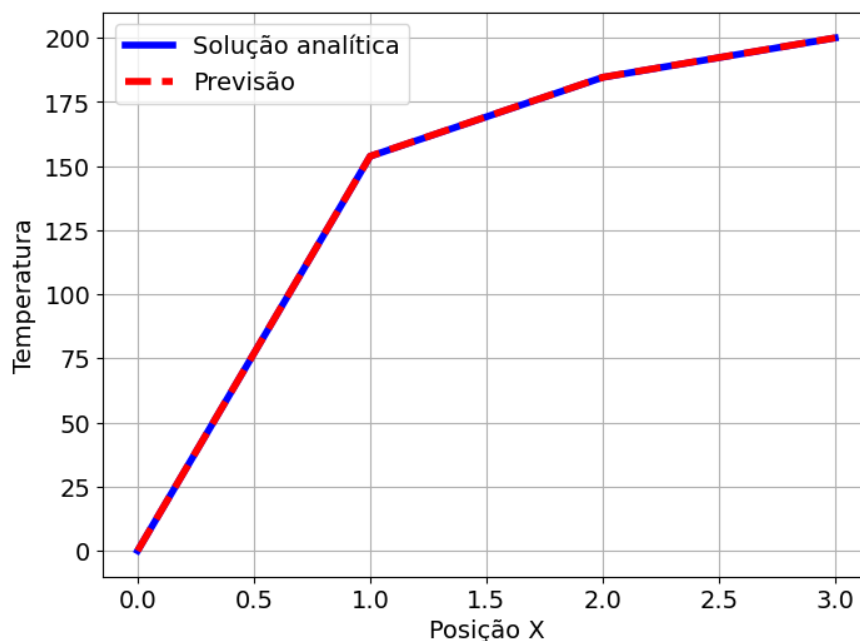
Os códigos utilizados para os problemas 1 e 2 encontram-se em <[https://github.com/hguthierry/diffusion\\_bar](https://github.com/hguthierry/diffusion_bar)> e <[https://github.com/hguthierry/three\\_fin\\_3d](https://github.com/hguthierry/three_fin_3d)>, respectivamente.

### 4.1 PROBLEMA 1: CONDUÇÃO DE CALOR UNIDIMENSIONAL EM REGIME PERMANENTE

Para modelar esse problema foram utilizadas 3 redes neurais iguais, uma para cada seção da barra. Os hiperparâmetros utilizados foram iguais para essas as redes e encontram-se na Tabela 9.

Vê-se que na Figura 11 o modelo atinge uma convergência quase perfeita, apesar de isso acontecer em um tempo ainda longo. Provavelmente o tempo ainda poderia ser reduzido drasticamente com ajustes nos hiperparâmetros.

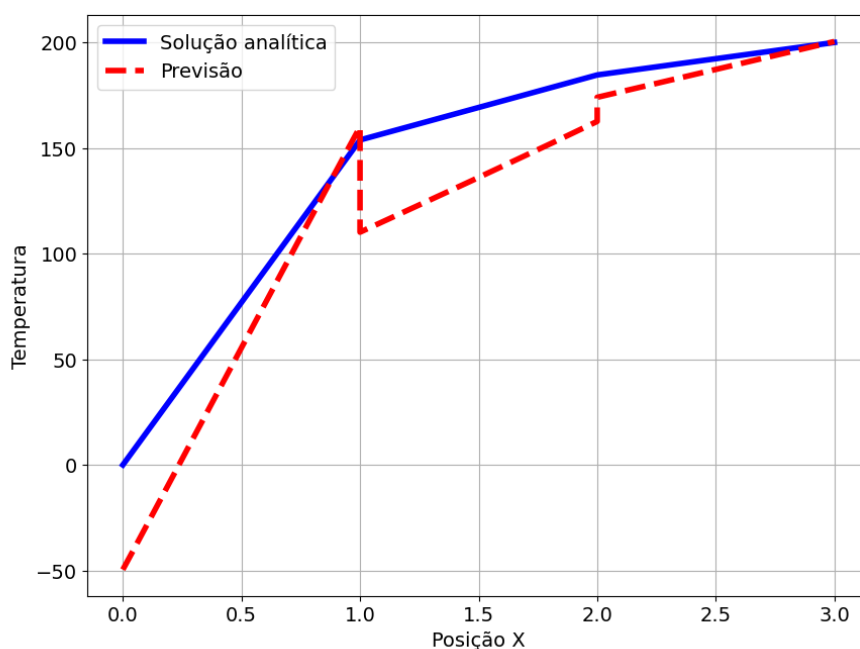
Figura 11 – Comparação solução analítica com previsão da PINN.



Distribuição de temperatura ao longo do comprimento da barra, versão não parametrizada. Em azul, vê-se o resultado previsto para a solução analítica e em vermelho a previsão da rede. Houve bastante concordância entre os dois. Fonte: autor.

De posse do resultado tentou-se testar o desempenho do modelo parametrizado,

Figura 12 – Comparação solução analítica com previsão parametrizada da PINN.



Distribuição de temperatura ao longo do comprimento da barra, versão parametrizada. Em azul, vê-se o resultado previsto para a solução analítica e em vermelho a previsão da rede. Houve uma discordância relativamente alta entre os dois. Fonte: autor.

Figura 12. Entre um código e outro poucas modificações foram necessárias. Vê-se que a precisão alcançada em comparação com a solução analítica foi significativamente menor. Para plotar os valores das Figuras 11 e 12 utilizou-se os valores da Tabela 6.

Tabela 5 – Hiperparâmetros usados para treinamento do problema 1

Hiperparâmetro	Valor
Arquitetura	<i>Feedforward</i>
Quantidade de camadas ocultas	6
Neurônios por camada	20
Função de ativação	SiLu
Otimizador	ADAM
Taxa de aprendizado	1e-3

Fonte: autor.

A Figura 13 mostra o erro relativo L2 ao longo do treinamento. Comparando-se o treinamento parametrizado e não parametrizado vê-se que o primeiro aparenta ter de fato uma precisão inferior. Por outro lado esse modelo consegue através de um treinamento prever várias situações com parâmetros de comprimento de barra e condutividade possível. Se considerar o tempo de convergência, 15 vezes maior,

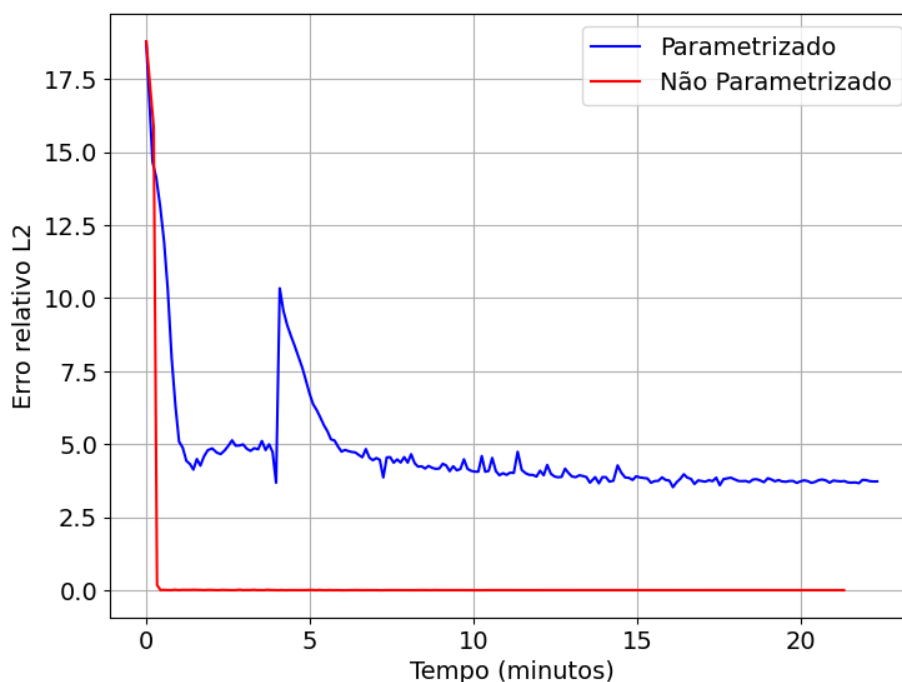
Tabela 6 – Parâmetros de validação problema 1

Parâmetro	Valor
$L_1$	1,0
$L_2$	5,0
$L_3$	10,0
$k_1$	1,0
$k_2$	1,0
$k_3$	1,0
$T_a$	0
$T_b$	200

Parâmetros usados na construção das Figuras 11 e 12 do problema 1. Fonte: autor.

pode-se dizer que houve um ganho de eficiência computacional.

Figura 13 – Comparação solução analítica com previsão parametrizada da PINN.



Erro relativo L2 dos dois modelos, parametrizado e não parametrizado. O segundo convergiu bem mais rápido e para um valor menor. Fonte: autor.

## 4.2 PROBLEMA 2: DISSIPADOR DE CALOR

Esse problema é significativamente mais complexo que o anterior e requereu mais neurônios, além de mais recursos computacionais.

O primeiro desafio enfrentado foi a quantidade de memória RAM (*Random Access*



*Memory*) exigida, que ultrapassava os 12,5 GB. Isso se deve ao tamanho elevado de *batch* especificado inicialmente pela NVIDIA.

Inicialmente o tempo entre cada iteração era 4,1 segundos. Sabidamente esse problema poderia requerer mais de 400 mil interações até ter uma precisão aceitável, o que levaria mais de 460 horas para resolver.

Tabela 7 – Configurações do computador local usado na pesquisa.

Configuração	Valor
Quantidade de CPU's	20
Modelo das CPU's	Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz

Fonte: autor.

#### 4.2.1 Paralelismo computacional

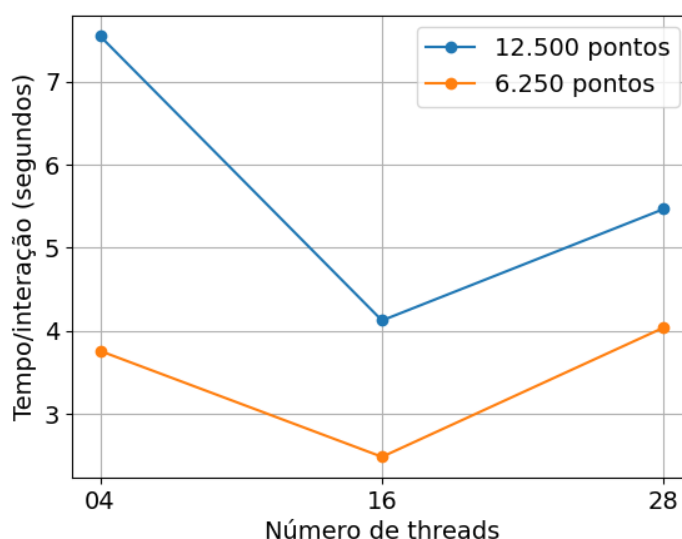
A premissa inicial era tentar utilizar apenas as CPUs disponíveis em laboratório (Tabela 7), visto que não se dispunha de GPU e o serviço de computação na nuvem pode ser caro, por este motivo buscou-se aproveitar a quantidade extra de CPUs dentro de um mesmo sistema, distribuindo a tarefa de treinamento entre as diversas CPUs simultaneamente, o que é chamado de paralelismo computacional.

Quando um processo computacional é segmentado ele é separado em *threads* que nada mais são subdivisões da tarefa principal. Tipicamente, um algoritmo pode se beneficiar da segmentação em *threads* só até certo ponto a partir da qual a eficiência começa a diminuir (PYTORCH CONTRIBUTORS, ).

Assim desejava-se saber o ponto ideal de *threads* para o modelo e computador em questão, além de se determinar se era viável executar o treinamento nele. Partiu-se para descobrir qual era o tamanho de *batch* ideal e se seria possível treinar a rede em tempo hábil. A Figura 14 mostra um estudo do tamanho de *batch* e número de *threads*.

Vê-se que o tamanho ideal de *threads* fica em um valor intermediário. Além disso, o impacto relativo da quantidade de *threads* é mais significativo em tamanhos de *batch* altos. Isso por que a cada interação a quantidade de cálculos paralelizáveis é maior. Adicionalmente, notou-se que em problemas com tamanho de *batch* muito pequeno quase não se notava diferença entre os tempos de computação.

Mostrou-se que, mesmo depois de melhorar o paralelismo computacional os tempos ainda continuaram longos: no melhor dos casos se gastava 2,4 segundos por interação.

Figura 14 – Estudo de *batch* e *threads* versus tempo

Esse gráfico mostra a variação no tempo de uma interação do treinamento para 12.500 pontos (azul) e 6.250 pontos (laranja), com o número de *Threads* variando ao longo do eixo horizontal. Fonte: autor.

#### 4.2.2 CPU x GPU

Nitidamente o treinamento com *cpu's* estava ineficiente, então cogitou-se executar o treinamento em GPU, usando o Google Colaboratory. A Tabela 8 mostra uma comparação entre tempos de interação. O tempo da GPU T4 foi cerca 10 vezes menor. Optou-se por utilizar o tamanho de *batch* de 12.500 pontos por estar mais próximo dos parâmetros fornecidos inicialmente.

Tabela 8 – Tempos por interação computador local *versus* GPU

Tamanho de <i>batch</i>	Computador local	GPU T4
6.250 pontos	2,4 s	0,19 s
12.500 pontos	4,1 s	0,33 s

Comparação do tempo médio por interação, computador local *versus* GPU T4. Fonte: autor.

#### 4.2.3 Análise do treinamento

As Figuras 15 a 17 mostram os resultados do treinamento. Vê-se que qualitativamente os modelos não-parametrizados e parametrizados concordam com a simulação numérica, apesar de uma precisão fina não ter sido alcançada. Os hiperparâmetros utilizados para as redes neurais encontram-se na Tabela ???. Os valores dos parâmetros usados para construção de ambas as Figuras 15 a 17 encontram-se na Tabela 10.

Tabela 9 – Hiperparâmetros usados para treinamento do problema 1

Hiperparâmetro	Valor
Arquitetura	<i>Feedforward</i>
Quantidade de camadas ocultas	6
Neurônios por camada	20
Função de ativação	SiLu
Otimizador	ADAM
Taxa de aprendizado	1e-3

Foram utilizadas três redes neurais, todas com a configuração acima. Fonte: autor

Tabela 10 – Parâmetros para validação problema 2

Parâmetro	Valor para validação (m)
Espessura das aletas laterais	100
Altura das aletas laterais	400
Comprimento das aletas laterais	1000
Espessura da aleta central	100
Altura da aleta central	400
Comprimento da aleta central	1000

Valores usados para validação e construção das Figuras 15 e 17. Fonte: autor.

Enquanto o treinamento do modelo não-parametrizado demorou cerca de 15 horas para ser completado, o parametrizado levou cerca 57 horas. Ambos usando os mesmos hiperparâmetros. Porém, nessa circunstância o parametrizado teve um custo-benefício melhor, uma vez que ele consegue simular com iguais níveis de precisão vários casos com parâmetros diferentes, efetivamente resolvendo várias problemas em um treinamento só. A Tabela 11 compara e tempos e quantidades de interação.

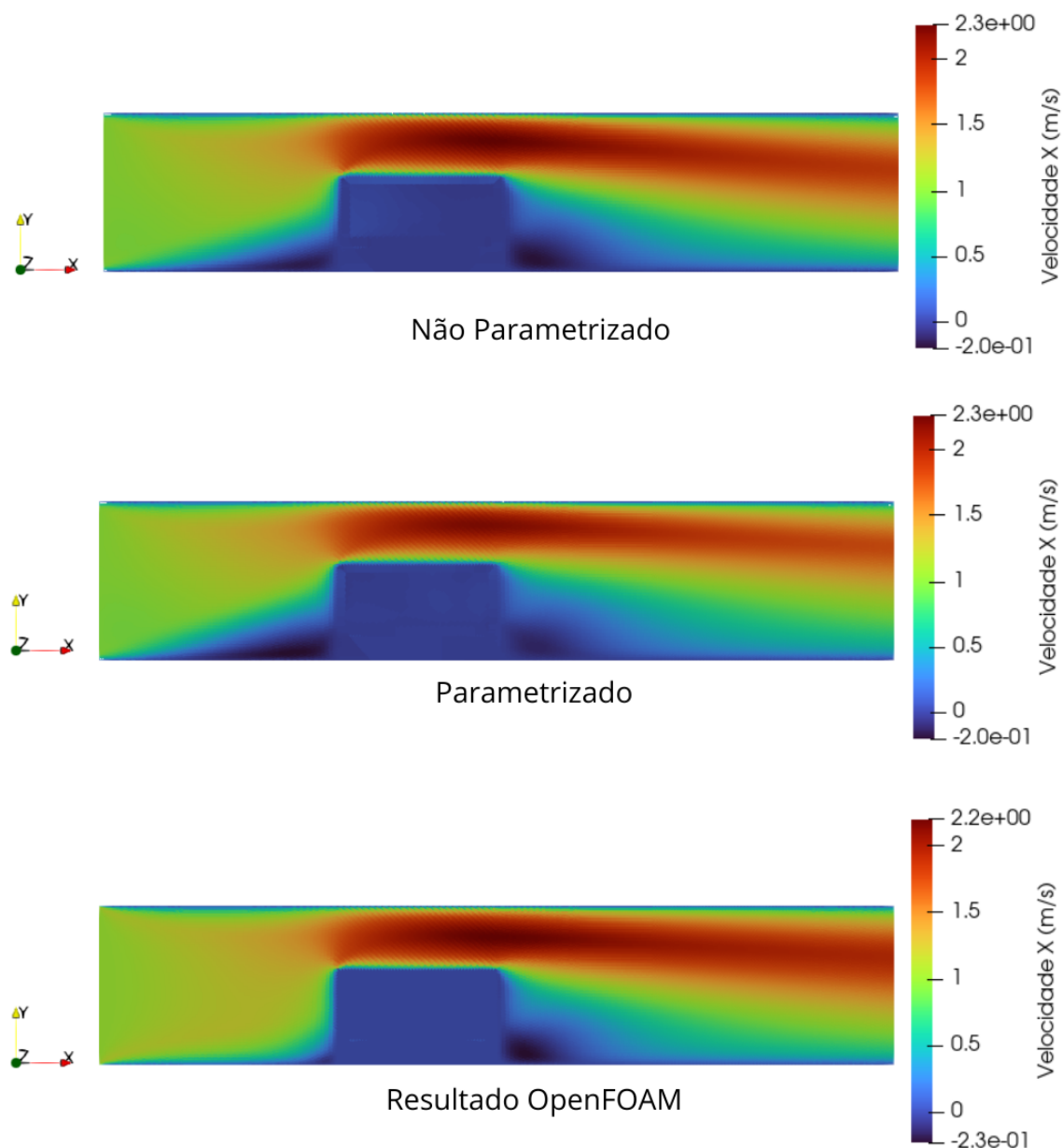
Tabela 11 – Tempos de treinamento problema 2

	Não parametrizado	Parametrizado
Quantidade de passos	260 mil	800 mil
Tempo de treinamento	15 horas	57 horas

Comparação tempo e quantidade de passos dos modelos não parametrizados e parametrizados. Fonte: autor.

A diferença na velocidade de convergência também pode ser vista nas Figuras 18 e 19 que mostram as evoluções da função de perda ao longo das interações.

Uma comparação das precisões alcançadas para diferentes saídas é feita na Figura 20. Pode-se ver que tal qual o problema 1 o parametrizado alcança precisões iguais ou melhores em menor tempo. Pode-se ver também que os erros foram maiores para saídas que têm gradientes menores como as velocidades em  $y$  e  $w$ .

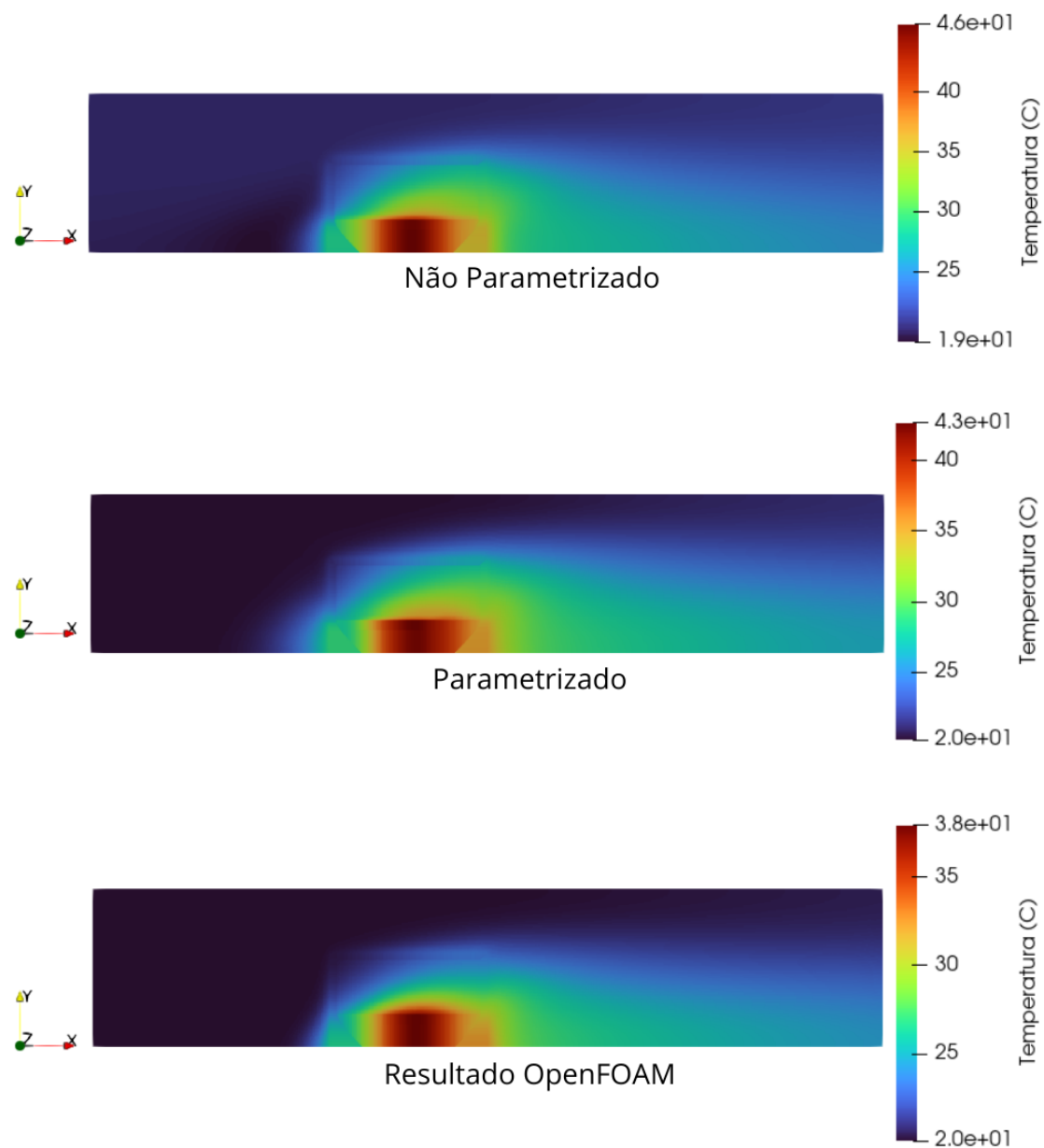
Figura 15 – Comparação soluções do campo de escoamento na direção  $X$ .

Nessa figura vê-se as soluções do campo de escoamento na direção  $X$ . Os modelos não parametrizado, parametrizado e a simulação do OpenFoam concordaram bem.

Fonte: autor.

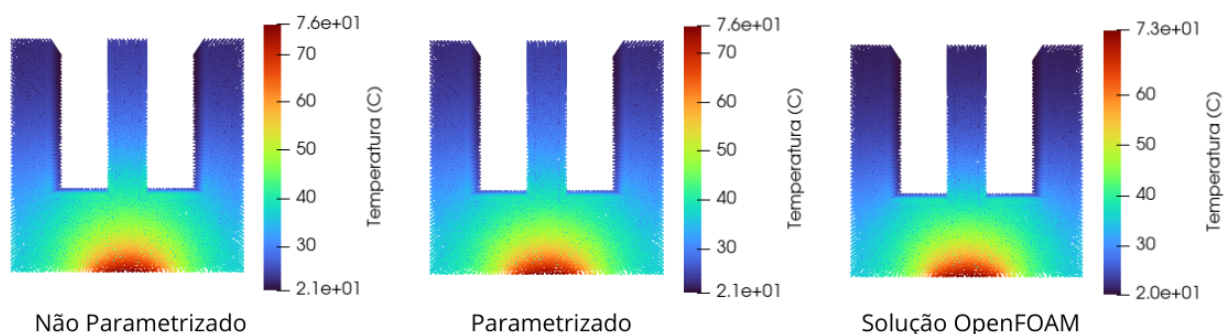
Outras observações podem ser feitas acerca dos dados da Figura 20. A métrica utilizada, o erro relativo L2, parece sugerir resultados muito piores do que os evidenciados, sugerindo penalizar muito pequenos desvios. Cuidado então deve ser tomado ao usar essa métrica, de acordo com a aplicação em mãos.

Figura 16 – Comparação soluções do campo de temperaturas no canal.



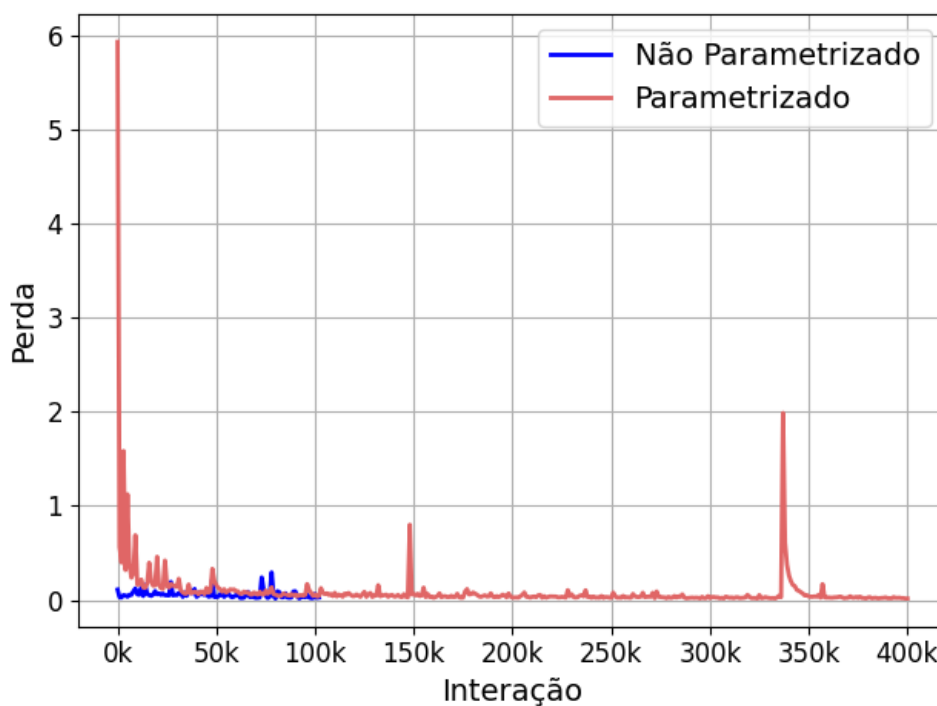
Nessa figura vê-se as soluções do campo de temperaturas. Os modelos não parametrizado, parametrizado e a simulação do OpenFoam concordaram apenas ligeiramente. Fonte: autor.

Figura 17 – Comparação soluções do campo de temperaturas no dissipador de calor.



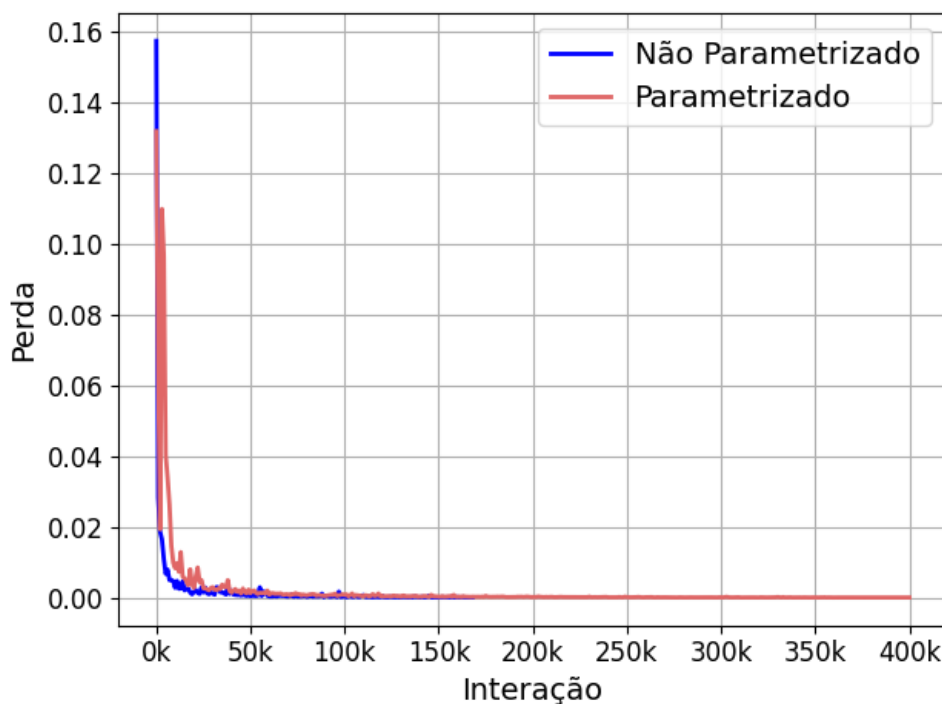
Nessa figura vê-se as soluções do campo de temperaturas. Os modelos não parametrizado, parametrizado e a simulação do OpenFoam concordaram ligeiramente.  
Fonte: autor.

Figura 18 – Treinamento problema 2 (Parte fluida).



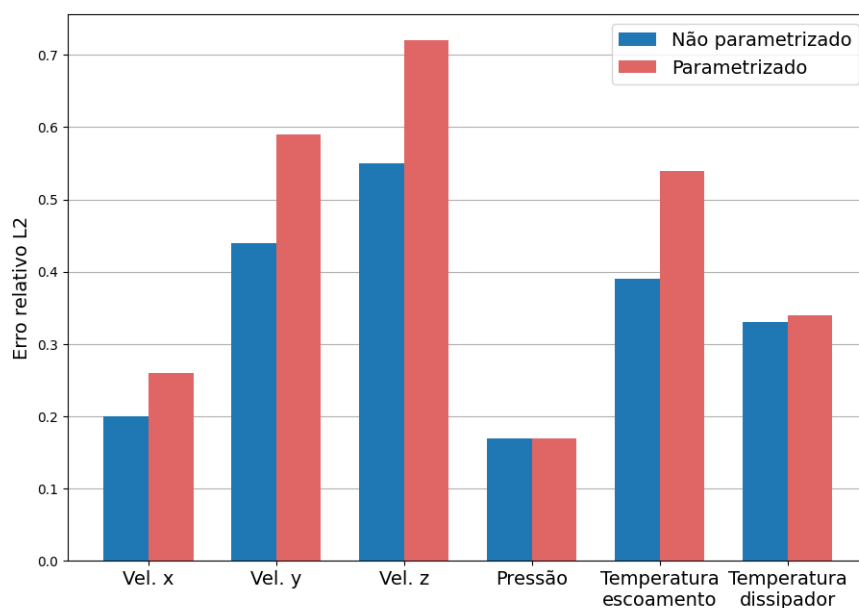
Nessa figura vê-se a variação da função de perda ao longo das interações do treinamento. O modelo não parametrizado (azul) convergiu mais rápido que o parametrizado (rosa). Fonte: autor.

Figura 19 – Treinamento problema 2 (Parte térmica).



Nessa figura vê-se a variação da função de perda ao longo das interações do treinamento. O modelo não parametrizado (azul) convergiu mais rápido que o parametrizado (rosa). Fonte: autor.

Figura 20 – Comparação da precisão entre modelo parametrizado e não parametrizado.



Comparação dos erros relativos de diferentes grandezas previstas pelas redes. Os erros foram altos, especialmente para o modelo parametrizado. Fonte: autor.

## 5 CONCLUSÕES

Conseguiu-se no trabalho implementar e desenvolver PINNs utilizando a biblioteca Modulus. A biblioteca mostrou-se um recurso valioso para este e outros estudos que desejem explorar simulação através de redes neurais, fornecendo uma solução robusta, completa e de fácil aprendizado.

Os problemas de calor foram resolvidos com relativa precisão: qualitativamente aproximaram-se do resultado desejado, mas uma precisão fina não foi alcançada. Em particular a precisão dos modelos parametrizados foi pior. Entretanto, uma vantagem dos modelos parametrizados foi o custo-benefício de tempo uma vez que conseguem aproximar solução qualquer combinação de parâmetros dentro do intervalo de treinamento, com um tempo de treinamento apenas 3 a 4 vezes maior.

Entre as maiores dificuldades do trabalho pode-se listar a disponibilidade de recursos computacionais, afinal, verificou-se ser inviável conduzir treinamentos complexos em CPUs, sendo necessário recorrer ao uso de GPUs pago do Google Colaboratory. As GPUs, portanto, se mostraram imprescindíveis para o treinamento, enquanto que as CPUs podiam ser usadas para pequenos testes. Outra dificuldade esteve em selecionar hiperparâmetros: Viu-se que eles influenciam fortemente a qualidade do treinamento, porém não há referenciais teóricos e metodológicos bons atualmente para se definir a escolha para um problema em específico. A seleção destes é bastante empírica.

Conclui-se por meio deste estudo que PINNs e a biblioteca Modulus Sym em particular, têm bastantes desafios a resolver, mas se continuarem a serem desenvolvidas, têm potencial para contribuir bastante para a simulação computacional. O treinamento de modelos parametrizados mostrou-se particularmente interessante.



## 6 SUGESTÕES PARA FUTUROS ESTUDOS

Ao se buscar as pesquisas na área de PINNs nota-se uma grande abstenção de informação sobre os tempos de computação gastos em cada problema resolvido, Isso dificulta uma comparação, ainda que qualitativa com os métodos tradicionais. Portanto, como sugestão para próximos estudos fica a de estabelecer um comparativo, ainda que mais qualitativo, entre os tempos de computação de métodos tradicionais e PINNs.

É importante também se deseja-se explorar problemas com algum grau de complexidade um estudo extensivo de hiperparâmetros para obter melhor acurácia e melhores tempos.

Por último, há diversas outras variações de PINNs, arquiteturas de redes neurais e abordagens diferentes para simulação através de redes neurais, muitas delas já implementadas no Modulus. Sugere-se explorar essas outras ferramentas a fim de explorar mais o potencial do DL em simulações.

## REFERÊNCIAS BIBLIOGRÁFICAS

- AGGARWAL, C. C. *Neural Networks and Deep Learning: A textbook*. Yorktown Heights, EUA: Springer, 2016.
- BERGMAN, T. L.; LAVINE, A. S.; INCROPERA, F. P.; DEWITT, D. P. *Fundamentos de transferência de calor e massa*. 7. ed. Rio de Janeiro: LTC, 2014.
- ÇENGEL, Y. A.; GHAJAR, A. J. *Transferência de calor e massa: uma abordagem prática*. 4. ed. Porto Alegre: AMGH, 2012.
- CHEUNG, K.; SEE, S. Recent advance in machine learning for partial differential equation. *CCF Transactions on High Performance Computing*, v. 3, n. 4, p. 298–310, 2021. Acesso em 04 de outubro de 2024. Disponível em: <<https://doi.org/10.1007/s42514-021-00076-7>>.
- CUOMO, S.; COLA, V. D.; GIAMPAOLO, F.; ROZZA, G.; RAISSI, M.; PICCIALI, F. Scientific machine learning through physics-informed neural networks: Where we are and what's next. *J.Sci Comput*, v. 92, n. 88, 2022. Acesso em 04 de outubro de 2024. Disponível em: <<https://arxiv.org/pdf/2201.05624>>.
- DOCKER INC. *What is a container?* Acesso em 20 de novembro de 2024. Disponível em: <<https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/>>.
- FOX, R. W.; MCDONALD, A. T.; PRITCHARD, P. J. *Introdução à Mecânica dos Fluidos*. 8. ed. Rio de Janeiro: LTC, 2014.
- GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. Cambridge, EUA: MIT Press, 2016.
- GUO, K.; YANG, Z.; YU, C.; BUEHLER, M. J. Artificial intelligence and machine learning in design of mechanical materials. *Materials Horizons*, v. 8, n. 4, p. 1153–1172, 2021. Acesso em 04 de outubro de 2024. Disponível em: <<http://dx.doi.org/10.1039/D0MH01451F>>.
- HORNIK, K.; STINCHCOMBE, M.; WHITE, H. Multilayer feedforward networks are universal approximators. *Neural Networks*, v. 2, n. 5, p. 359–366, 1989. Acesso em 04 de outubro de 2024. Disponível em: <[https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8)>.
- KARNIADAKIS, G. E.; KEVREKIDIS, I. G.; LU, L.; PERDIKARIS, P.; WANG, S.; YANG, L. Physics-informed machine learning. *J.Sci Comput*, v. 3, p. 422–440, 2021. Acesso em 04 de outubro de 2024. Disponível em: <<https://doi.org/10.1038/s42254-021-00314-5>>.
- NVIDIA MODULUS TEAM. *NVIDIA Modulus Sym Documentation*. 2023. Acesso em 20 de novembro de 2024. Disponível em: <<https://docs.nvidia.com/deeplearning/modulus/modulus-sym/index.html>>.
- PYTORCH CONTRIBUTORS. *Pytorch Documentation: Cpu threading and torchscript inference*. Acesso em 03 de outubro de 2024. Disponível em: <[https://pytorch.org/docs/stable/notes/cpu\\_threading\\_torchscript\\_inference.html](https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html)>.

RAISSI, M.; PERDIKARIS, P.; KARNIADAKIS, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, v. 378, p. 686–707, 2019. Acesso em 04 de outubro de 2024. Disponível em: <<https://doi.org/10.1016/j.jcp.2018.10.045>>.

SALEHI, H.; BURGUEÑO, R. Emerging artificial intelligence methods in structural engineering. *Engineering Structures*, v. 171, p. 170–189, 2018. Acesso em 04 de outubro de 2024. Disponível em: <<https://doi.org/10.1016/j.engstruct.2018.05.084>>.

SILVA, E. *Aplicação de técnicas de redes neurais na simulação de escoamento de gás natural enriquecido por hidrogênio verde*. 2023. Trabalho de Conclusão de Curso (TCC) – Universidade Federal de Pernambuco, Recife. Disponível em: <<https://repositorio.ufpe.br/handle/123456789/52620>>.

VASILEV, I.; SLATER, D.; GIANMARIO, S.; ROELANTS, P.; ZOCCA, V. *Python Deep Learning*. 2. ed. Birmingham, RU: Packt Publishing Ltd., 2019.

WHITE, F. M. *Mecânica dos Fluídos*. 6. ed. Porto Alegre: AMGH, 2011.

## APÊNDICE A – EXEMPLO DIDÁTICO UTILIZANDO MODULUS

Para fins didáticos é apresentado o código escrito em Python com a biblioteca Modulus, similar ao problema 1, mas com apenas uma barra com extremidades a temperatura 0 e 100 graus e condutividade térmica 1.

Inicialmente é necessário carregar as bibliotecas e módulos necessários (código A.1).

```

1 import numpy as np
2 import modulus.sym
3
4 from sympy import Symbol, Eq, Function
5 from modulus.sym.hydra import instantiate_arch, ModulusConfig
6 from modulus.sym.solver import Solver
7 from modulus.sym.domain import Domain
8 from modulus.sym.geometry.primitives_1d import Line1D
9 from modulus.sym.domain.constraint import (
10     PointwiseBoundaryConstraint,
11     PointwiseInteriorConstraint,
12 )
13 from modulus.sym.domain.validator import PointwiseValidator
14 from modulus.sym.key import Key
15 from modulus.sym.eq.pde import PDE
16 from modulus.sym.geometry import Parameterization

```

Código A.1 – Importa bibliotecas

As equações diferenciais do problema são descritas por objetos do tipo PDE. O modulus já vem com várias equações comuns definidas, mas é possível criar uma personalizada (Código A.2):

```

1 class Diffusion(PDE):
2     def __init__(self, T="T", K=1):
3
4         # coordinates
5         x = Symbol("x")
6         T = Function(T)(x)
7
8         # set equations

```

```

9         self.equations = {}
10        self.equations["diffusion"] = -(K * T.diff(x)).diff(x)

```

Código A.2 – Define uma PDE

Para esses objetos estarem bem definidos basta especificar o atributo `self.equations` que deve ser um dicionário contendo expressões simbólicas da biblioteca `sympy`.

O código principal é encapsulado dentro da função `run`, Código A.3. O argumento dessa função é um objeto do tipo `cfg` que contém todos os hiperparâmetros e configurações adicionais do *script*.

```

1 def run(cfg):
2     # Início do código
3
4     # Meio do código
5
6     # Fim do código

```

Código A.3 – Função principal do script

Em primeiro lugar é definido a geometria, Código A.4. A geometria pode ser construído através de arquivos `.STL` ou usando primitivas geométricas. Aqui usa-se a segunda opção. O Modulus vem com várias primitivas geométricas e é possível operações booleanas entre elas.

```

1     L = Symbol("L")
2     if cfg.custom.parameterized:
3         pr = Parameterization({L:(1,10)})
4     else:
5         pr = Parameterization({L:1.0})
6
7     barra = Line1D(0, L, parameterization=pr)

```

Código A.4 – Definição da geometria da barra

Para a barra usa-se a geometria `Line1D`. Os argumentos obrigatórios são o ponto inicial ( $x = 0$ ) e final ( $x = L$ ). Se a geometria for parametrizada é necessário incluir o argumento `parameterization`. O objeto `Parameterization` recebe um dicionário em que as chaves são `Symbols` e o argumento é uma tupla descrevendo os limites de variação

do respectivo parâmetro. Nesse caso, parametrizou-se o comprimento da barra de 1 a 10.

Em seguida, a equação do problema é instanciada, código A.5.

```
1      condutividade = 10
2      diff_u = Diffusion(T="u", K=condutividade)
```

Código A.5 – Instanciamento da classe Diffusion(PDE)

No código A.6 a rede neural propriamente dita é criada:

```
1      if cfg.custom.parameterized:
2          input_keys = [Key("x"), Key("L")]
3      else:
4          input_keys=[Key("x")]
5
6      diff_net_u = instantiate_arch(
7          input_keys=input_keys,
8          output_keys=[Key("u")],
9          cfg=cfg.arch.fully_connected,
10     )
```

Código A.6 – Instanciamento da rede neural

O argumento `input_keys` recebe as entradas da rede neural, `output_keys` são as saídas da rede neural (temperatura). Observe que quando a configuração parametrizado está ativa o comprimento  $L$  também é uma entrada. O argumento `cfg` recebe as configurações da arquitetura da rede (tipo, quantidade de camadas, neurônios por camada, etc).

O Modulus trabalha com estrutura de grafos. Os nós do grafo podem ser equações ou redes neurais. Usa-se o grafo criado para estabelecer coerência entre entradas e saídas do modelo e também computar as derivadas necessárias no treinamento. A interligação entre os nós é feita de forma automática, desde que os nomes das entradas e saídas mantenham-se coerentes ao longo do código. O código A.7 mostra a criação do grafo de nós.

```
1      nodes = (
2          diff_u.make_nodes()
```

```

3         + [diff_net_u.make_node(name="u_network")]
4     )

```

Código A.7 – Instanciamento do grafo de nós

O Modulus não trabalha diretamente com a definição da função de perda, mas a constrói a partir de restrições. As restrições podem especificar: aderência às equações no interior, condições de contorno, condições iniciais ou necessidade da rede aderir a um conjunto de dados rotulados. O Código A.8 estabelece restrições a partir das condições de contorno, usando restrições do tipo `PointwiseBoundaryConstraint`.

```

1     # Condição contorno direita, temperatura constante 0 C
2     extremidade_dir = PointwiseBoundaryConstraint(
3         nodes=nodes,
4         geometry=barra,
5         outvar={"u": 0},
6         batch_size=5,
7         criteria=Eq(x, L),
8         parameterization=pr
9     )
10    domain.add_constraint(extremidade_dir, "extremidade_dir")
11
12    # Condição contorno esquerda, temperatura constante 100 C
13    extremidade_esq = PointwiseBoundaryConstraint(
14        nodes=nodes,
15        geometry=barra,
16        outvar={"u": 100},
17        batch_size=5,
18        criteria=Eq(x, 0),
19        parameterization=pr
20    )
21    domain.add_constraint(extremidade_esq, "extremidade_esq")

```

Código A.8 – Instanciamento das condições de contorno através de restrições

Os argumentos a serem especificados são: `nodes` especifica o grafo a ser usado; `geometry` especifica sobre qual geometria está se aplicando a restrição (o programa identifica automaticamente o contorno destas); `outvar` é um dicionário especificando o valor das saídas no contorno; `batch_size` é a quantidade de pontos de colocação no contorno; `criteria` restringe o contorno usando uma equação simbólica e `parameterization` especifica o intervalo de parâmetros.

De forma similar é necessário estabelecer a restrição de aderência a equação diferencial no interior do problema, usando `PointwiseInteriorConstraint`, Código A.9. "diffusion" refere-se ao nome da equação diferencial, previamente estabelecida.

```

1      # interior
2      interior_u = PointwiseInteriorConstraint(
3          nodes=nodes,
4          geometry=barra,
5          outvar={"diffusion": 0},
6          batch_size=20,
7          parameterization=pr
8      )
9      domain.add_constraint(interior_u, "interior_u")

```

Código A.9 – Aplicação das equações diferenciais no interior

É necessário injetar as restrições em um objeto da classe `Domain()`, Código A.11.

```

1      domain = Domain()
2      domain.add_constraint(extremidade_dir, "extremidade_dir")
3      domain.add_constraint(extremidade_esq, "extremidade_esq")
4      domain.add_constraint(interior_u, "interior_u")

```

Código A.10 – Criação do domínio e injeção das restrições

Para comparação de precisão o Modulus dispõe de objetos chamados validadores, que são responsáveis por gerar dados para comparação do erro e pós-processamento. Há vários tipos de validadores, aqui usa-se o tipo `PointWiseValidator`. O argumento `invar` contém os pontos para verificação, `true_outvar` os valores das saídas esperadas nesses pontos.

```

1      val = PointwiseValidator(nodes=nodes, invar=invar_numpy,
2                                true_outvar=outvar_numpy)
3      domain.add_validator(val, name="Val1")

```

Código A.11 – Criação de validadores

Por fim, inclui-se o comando de execução do treinamento, Código A.12

```

1      # make solver

```



```
2     slv = Solver(cfg, domain)
3     slv.solve()
```

Código A.12 – Comando de execução