AGREGADOR DE DADOS JUNTO À ANÁLISE DE RESULTADOS DE REDES NEURAIS GUIADAS POR FÍSICA VIA FRAMEWORK MODULUS

Matheus Lima Scramignon

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Marta Lima de Queirós Mattoso

Rio de Janeiro
Dezembro de 2022

AGREGADOR DE DADOS JUNTO À ANÁLISE DE RESULTADOS DE REDES NEURAIS GUIADAS POR FÍSICA VIA FRAMEWORK MODULUS

Matheus Lima Scramignon

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientador: Marta Lima de Queirós Mattoso

Aprovada por: Prof. Marta Lima de Queirós Mattoso
Prof. Alvaro Luiz Gayoso de Azeredo Coutinho
Prof. Pedro Mario Cruz e Silva
Prof. Daniel Cardoso Moraes de Oliveira

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2022

Lima Scramignon, Matheus

Agregador de Dados junto à análise de resultados de Redes Neurais Guiadas por Física via framework Modulus/Matheus Lima Scramignon. – Rio de Janeiro: UFRJ/COPPE, 2022.

XII, 63 p.: il.; $29,7$cm.

Orientador: Marta Lima de Queirós Mattoso

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2022.

Referências Bibliográficas: p. 59 – 63.

1. Primeira palavra-chave. 2. Segunda palavra-chave. 3. Terceira palavra-chave. I. Lima de Queirós Mattoso, Marta. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha querida mãe, que para sempre viverá no meu coração.

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus pais por terem me dado o suporte necessário para que eu pudesse concluir esse trabalho. Agradeço ao meu pai pelo incentivo em continuar meus estudos e pelas palavras de apoio em momentos difíceis. Agradeço à minha mãe por sempre ter insistido na importância de minha educação desde cedo e que me ensinou a valorizar a busca constante por conhecimento. A eles, agradeço imensamente.

Agradeço ao meu irmão querido pela amizade e por me inspirar quanto à disciplina para continuar estudando.

Sou imensuravelmente grato à professora Marta Lima de Queiroz Mattoso, que além de ter me acolhido, me forneceu a melhor orientação que jamais poderia pedir. Do mesmo modo, agradeço também ao professor Álvaro Luiz Gayoso de Azeredo Coutinho, cujas sugestões e comentários fizeram toda a diferença no curso desse trabalho. Costumo dizer que dou muita sorte com orientadores e supervisores no geral. Dessa vez, felizmente, não foi diferente.

Agradeço à minha companheira nessa jornada de vida, Fernanda, que sempre fez questão de permanecer ao meu lado, de me aturar durante todo esse período e de me levantar em momentos de incerteza e ansiedade. Espero que eu seja capaz de poder retribuir à altura, servindo de apoio em momentos críticos mas também estando ao seu lado nos seus momentos de conquista. Você para sempre terá um lugar especial no meu coração.

Agradeço também aos meus colegas da ISDB Flowtech, em especial ao Marcelo Pasqualette e ao João Carneiro, que sempre serviram de inspiração para meu aprendizado contínuo e que deram todo o incentivo e condições necessárias para que eu continuasse meus estudos.

Agradeço à querida Maria Aparecida pelas conversas, receitas, brincadeiras, e por ter sido para mim praticamente uma terceira mãe.

Agradeço por fim ao Felipe, Jéssica, Otávio, Leonardo e ao Ismael, por terem me oferecido suas preciosas amizades durante todos esses anos turbulentos. Isso certamente fez toda a diferença para que eu tivesse forças para perseverar nesse trabalho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

AGREGADOR DE DADOS JUNTO À ANÁLISE DE RESULTADOS DE REDES NEURAIS GUIADAS POR FÍSICA VIA FRAMEWORK MODULUS

Matheus Lima Scramignon

Dezembro/2022

Orientador: Marta Lima de Queirós Mattoso

Programa: Engenharia de Sistemas e Computação

Recentemente, tem ganhado destaque no meio acadêmico o desenvolvimento de modelos de Redes Neurais Guiadas à Física (*PINN*), onde diferentes equações diferenciais parciais que descrevem o comportamento físico de um determinado sistema são acopladas à função de perda desses modelos. Tal família de modelos representa um novo paradigma para a solução de PDEs, tanto para problemas diretos quanto inversos, onde deseja-se estimar parâmetros físicos de um sistema. Diferentes *frameworks* que visam facilitar a produção e treinamento de modelos desse gênero são disponibilizados atualmente, sendo o *framework Modulus* um dos que têm ganhado destaque ultimamente. De qualquer modo, apesar tais pacotes facilitarem a construção de experimentos, é necessário que seja considerada uma estratégia de análise de resultados produzidos para diferentes problemas físicos colocados. Apresenta-se nessa dissertação a ferramenta aqui denominada de *Modulus Aggregator*, que foi desenvolvida com a finalidade de apoiar o especialista junto ao processo de análise de dados e configuração de hiperparâmetros de múltiplos modelos produzidos a partir de uma estratégia de agregação de resultados, complementando a ferramenta de visualização de dados *Tensorboard* e aproveitando-se da estrutura de diretório nativa de um experimento *Modulus*. Mostra-se que a utilização da ferramenta tem o potencial não só de dar suporte a análise de resultados, como também de servir para a automatização do processo de extração e filtragem de modelos em um cenário de grande volume de dados.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DATA AGGREGATOR WITH THE ANALYSIS OF RESULTS OF PHYSICS INFORMED NEURAL NETWORKS VIA MODULUS FRAMEWORK

Matheus Lima Scramignon

December/2022

Advisor: Marta Lima de Queirós Mattoso

Department: Systems Engineering and Computer Science

The development of Physics Informed Neural Networks (*PINN*) has been receiving considerable highlights in the academic field lately, where different partial derivative equations that describe the physical behavior of a certain system are incorporated in the loss functions of neural networks. This model family represents a new paradigm for the solutions of PDEs, for both forward and inverse problems where the objective is to estimate physical parameters of a system. Different frameworks that aim to facilitate the production and training of such models are being currently provided, and *Modulus* is one of the available frameworks that has been gaining ground recently. In any case, despite of the capability of these packages to assist the construction of experiments, it is important to consider a viable results analysis strategy for different physical problems being considered. In this work, it is presented the so called *Modulus Aggregator* tool which was developed with the objective of supporting the specialist in the data analysis process and the hyperparameter configuration of multiple models produced, with a results aggregation strategy, complementing the *Tensorboard* visualization tool and taking advantage of the native directory structure of a *Modulus* experiment. It is showed that the proper use of the developed tool adds not only the potential of assisting the results analysis process, but also the possibility of automating the extraction and filtering activities of trained models in a scenario of significant amount of data.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

1 Introdução **1**

2 Redes Neurais Guiadas à Física (*PINNs*) **6**

2.1 Definição 6

2.2 Aplicações 9

2.3 Gerenciamento de Dados 10

2.4 *Frameworks* de desenvolvimento 11

2.4.1 *DeepXDE* 12

2.4.2 *SciANN* 12

2.4.3 *NeuroDiffEQ* 13

2.4.4 *NVIDIA Modulus* 13

3 NVIDIA Modulus - Experimentos preliminares **15**

3.1 Análise da implementação de *PINNs* no *Modulus* 15

3.1.1 Definição do Experimento Preliminar de Escoamento em Seção Cilíndrica 16

3.1.2 Organização do Código 16

3.1.2.1 Pontos de definição da Equação Diferencial e nós da Rede Neural 18

3.1.2.2 Definição da Geometria do Problema 20

3.1.2.3 Definição das Restrições Físicas e de Contorno 20

3.1.2.4 Definição de Objetos Adicionais 21

3.1.2.5 Definição do Solver e Treinamento 23

3.1.3 Geração de Dados de Resultados 24

3.1.4 Análise de dados via *TensorBoard* 27

3.1.5 Fluxograma para definição de Experimento *Modulus* e geração de *PINNs* 29

4 Análise de escolha de modelos em *PINNs* com *Modulus* **32**
4.1 Geração e Análise de modelos com *Modulus* 32
4.1.1 Definição do Experimento Explorado de Equação de Onda em meio 1D 33
4.1.2 Execução do Experimento 33
4.1.2.1 Reestruturação do Código 33
4.1.2.2 Geração dos Modelos 35
4.1.2.3 Análise de Resultados 36
4.2 Limitações de Análise de Modelos 40

5 *Modulus Aggregator* - Agregador de dados de Experimentos *Modulus* **42**
5.1 Objetivo do desenvolvimento da ferramenta 42
5.2 Arquitetura de *Software* do *Modulus Aggregator* 43
5.3 Metodologia de Uso 45
5.4 Justificativas para abordagem de desenvolvimento do *Modulus Aggregator* 46

6 Avaliação Experimental **47**
6.1 Experimento Conduzido de Equação de Onda em domínio 2D 47
6.2 Utilização do *Modulus Aggregator* 49
6.3 Análise de Resultados 51

7 Conclusão **57**

Referências Bibliográficas **59**

Lista de Figuras

2.1 Quantidade de trabalhos relacionados a *PINNs* publicados anualmente, agregados via Scopus. 7

3.1 Diretório de Exemplos resolvidos com o pacote *Modulus*. 17
3.2 Conteúdo do diretório *cylinder*. 17
3.3 Declaração do método principal *run()*. 18
3.4 Código da definição dos nós da equação física e da rede neural. 18
3.5 Conteúdo do arquivo de configuração *config.yaml*. 19
3.6 Código da definição da geometria do problema físico proposto. 20
3.7 Figura ilustrativa da geometria do problema. 21
3.8 Códigos das Definições das Restrições Físicas e de Contorno. 22
3.9 Código da definição do objeto de validação. 22
3.10 Códigos das Definições dos objetos adicionais de monitoramento e inferência. 23
3.11 Código da definição da classe do Solver. 24
3.12 Configuração completa da *PINN* especificada em *.hydra/config/config.yaml*. 25
3.13 Continuidade esperada e predita para o interior da geometria via *Paraview*. 25
3.14 Arquivos de imagem relativos aos objetos de validação e inferência, para a variável de saída de pressão. 26
3.15 Monitoramento da evolução dos valores de queda de pressão ao longo do treinamento. 26
3.16 Gráficos da evolução dos erros de treinamento e *learning rate* via *TensorBoard*. 27
3.17 Gráficos da evolução dos erros de validação via *TensorBoard*. 28
3.18 Gráficos das grandezas de monitoramento via *TensorBoard*. 28
3.19 Mapas de contorno referentes ao objeto de inferência via *TensorBoard*. 28
3.20 Mapas de contorno referentes ao objeto de validação via *TensorBoard*. 29
3.21 Configuração da rede via *TensorBoard*. 29

3.22 Fluxograma de definição de um experimento *Modulus* e geração de *PINNs*. 31

4.1 Código da definição da Equação de Onda 1D. 34
4.2 Código principal do experimento 2 depois da reestruturação. 35
4.3 Comando utilizado para execução do experimento *wave_1d.py* 36
4.4 Evolução temporal dos erros de treinamento para os modelos gerados. 36
4.5 Formato do *DataFrame* extraído para análise. 37
4.6 Formato do *DataFrame* reorganizado para análise. 38
4.7 Tempo de treinamento de cada passo dos modelos treinados com os diferentes otimizadores. 39
4.8 Erros de treinamento para modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios.. 40

5.1 Arquitetura do *Modulus Aggregator* e sua integração com o *Modulus*. . 44
5.2 Comandos disponíveis no *Modulus Aggregator*. 45

6.1 Comando utilizado para execução do experimento *wave_2d.py* 48
6.2 Função referente à primeira parte do *script* utilizado. 49
6.3 Função referente à segunda parte do *script* utilizado. 51
6.4 Função referente à terceira parte do *script* utilizado. 52
6.5 Esquema de utilização do *Modulus Aggregator*. 52
6.6 Gráficos do erro de validação em c no *Tensorboard*. 53
6.7 Evolução temporal para erros de treinamento e validação dos modelos filtrados. 54
6.8 Erros de validação de u em função do *learning rate* para a arquitetura de 5 camadas com 128 neurônios e otimizador *adam*. 55
6.9 Erros de validação de u em função dos diferentes *otimizadores* para a arquitetura de 5 camadas com 128 neurônios e *learning rate* inicial igual a 0.001. 56

Lista de Tabelas

2.1 *Frameworks* de desenvolvimento de *PINNs*. 12

4.1 Tempos médios de treinamento de cada passo dos modelos treinados com os diferentes otimizadores. 38
4.2 Erros de treinamento no último passo para modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios. . . . 39
4.3 Valores de *Learning rate* para modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios. 39

6.1 Melhores modelos filtrados. 53

Capítulo 1

Introdução

O Aprendizado de Máquina (*Machine Learning*) é uma área da Inteligência Artificial que utiliza técnicas de estatística aplicadas a algoritmos computacionais com o objetivo de estimar estatísticamente funções complexas com o auxílio de dados GOODFELLOW *et al.* (2016). Tal estimativa, ou aprendizado, pode ser feito de diferentes maneiras, a depender da aplicação, mas sempre conta com a existência de um conjunto de dados (oriundos, por exemplo, de medidas efetuadas por sensores, registros computacionais e resultados de simulações), de um conjunto de hipóteses (que será utilizado para achar uma função que se aproxime da função alvo que se deseja modelar) e de um algoritmo de aprendizado que realizará o treinamento do modelo ABU-MOSTAFA *et al.* (2012). Dentro do paradigma de Aprendizado Supervisionado, os pontos coletados de um conjunto aleatório de entrada disponível $\mathbf{x}$ contém valores associados de saída $\mathbf{y}$, os quais devem ser utilizados dentro do processo de treinamento do modelo de maneira a aprender a prever $\mathbf{y}$ a partir de $\mathbf{x}$ a partir da estimativa de uma função de probabilidade $p(y|x)$ GOODFELLOW *et al.* (2016).

Existe uma gama de algoritmos de Aprendizado de Máquina, cujas características são aproveitadas dentro de diferentes domínios de aplicação. Nesse contexto, as Redes Neurais Artificiais podem ser formalmente definidas como pertencentes a uma família de algoritmos representadas por um sistema de grafo composto por elementos de processamento capazes de computar informações a partir de seu estado em função da entrada ECKMILLER e V.D. MALSBURG (1989). Tal família de modelos tem recebido cada vez mais atenção, principalmente por seu uso cada vez mais acentuado em aplicações de aprendizado profundo (*Deep Learning*).

Já o termo *Deep Learning* refere-se a um conjunto de técnicas de *Machine Learning*, cujo foco encontra-se no desenvolvimento de modelos baseados em Redes Neurais Artificiais que utilizam múltiplas camadas de unidades de processamento não lineares DENG (2014), principalmente junto a aplicações que produzem um vasto volume de dados. Atualmente, o desenvolvimento de novas técnicas e algorit-

mos de *Deep Learning*, aliado à maior capacidade disponível de processamento de grandes quantidades de dados tornou possível um relevante avanço em uma gama considerável de áreas, como visão computacional KRIZHEVSKY *et al.* (2012), reconhecimento de voz e fala ZHANG *et al.* (2021), e muitas outras.

Porém, para determinados tipos de aplicações científicas, como por exemplo dinâmica de fluidos, propagação de ondas e mecânica quântica, a quantidade limitada de dados de campo disponível somada ao alto custo de produção de dados sintéticos acaba por tornar proibitivo o treinamento de modelos robustos baseados puramente em dados. A partir desse tipo de problema, surge a necessidade de acoplar o conhecimento físico, representado por meio de equações diferenciais parciais, ao treinamento e geração de modelos, como uma estratégia de regularização RAISSI *et al.* (2019), devido ao volume reduzido de dados de campo disponível para domínios multidimensionais.

Nesse contexto, um campo de estudo que tem recebido cada vez mais atenção recentemente está relacionado ao desenvolvimento de modelos de Redes Neurais baseadas em Física, ou *Physics Informed Neural Networks* (*PINNs*) KARNIADAKIS *et al.* (2021). Tal família de modelos busca integrar o conhecimento físico de um sistema, determinado pelas PDEs que o modelam, dentro da composição da função de custo de uma rede neural. Desse modo, a rede busca minimizar as parcelas relativas a dados referentes às condições de contorno ou de dados de campo coletados em diferentes pontos do domínio e à física representada pelas PDEs durante seu treinamento. Tal estratégia consegue aproveitar dados ruidosos coletados em campo, bem como respeitar a física vigente dentro do sistema físico analisado.

De qualquer modo, existem questões relevantes e que devem ser levadas em consideração quando trata-se da implementação de *PINNs*. Para que exista um acoplamento eficiente das equações que modelam um certo sistema físico a ser estudado ao treinamento de uma rede neural, exige-se que o cientista tenha um entendimento relativamente aprofundado em relação a implementação de redes em algum tipo de arcabouço de desenvolvimento de *Deep Learning*, como por exemplo *TensorFlow* ABADI *et al.* ou *Pytorch* PASZKE *et al.* (2019). É possível encontrar implementações práticas de *PINNs* no trabalhos de RAISSI *et al.* (2019) e de THUEREY *et al.* (2021). Dessa forma, apenas o conhecimento das equações físicas referentes ao domínio do sistema não é o suficiente para que o desenvolvimento de uma *PINN* seja possível, ainda mais quando considera-se a necessidade de operacionalização de um esquema robusto e escalável de coleta de dados de proveniência para fins de suporte à análise de resultados junto à configuração de hiperparâmetros de múltiplos modelos.

Nesse sentido, existe atualmente uma iniciativa da NVIDIA que disponibiliza um *framework* chamado *Modulus* HENNIGH *et al.* (2021) (anteriormente chamado

Simnet). Esse *framework* tem o objetivo de dar suporte à construção de modelos informados por física, facilitando a definição de domínios físicos, bem como a definição de PDEs a serem acopladas durante o treinamento da rede, além de prover também um esquema nativo para a paralelização durante a geração de modelos, facilitando a escalabilidade de eventuais experimentos desenvolvidos.

Porém, quando em face de um experimento relacionado ao desenvolvimento de *PINNs*, é comum que o especialista tenha a necessidade de gerar não apenas um, mas vários modelos, a fim de que o cientista possa fazer análises comparativas entre estes e despender esforço maior em uma estratégia de treinamento que seja favorável a partir dos resultados oriundos de sua análise. Para facilitar essa análise por parte do desenvolvedor das *PINNs*, o próprio *framework Modulus* disponibiliza alternativas *built-in* para análise visual de resultados a partir do uso da ferramenta *TensorBoard* ABADI *et al.*.

No entanto, apesar de os recursos de visualização para a análise de dados, serem essenciais, análises complementares sobre dados escalares envolvendo agregações e dados de desempenho tornam-se cada vez mais importantes SPINNER *et al.* (2020). Porém, deixar esse monitoramento por conta do desenvolvedor da *PINN* é cansativo, gera arquivos de estruturas heterogêneas e por isso de difícil comparação, além de uma organização de arquivos por muitas vezes não intuitiva. Surge então a necessidade do desenvolvimento de um recurso que facilite a análise comparativa entre diferentes modelos gerados, de modo complementar à análise visual. De fato, resultados anteriores, desenvolvidos na COPPE/UFRJ SILVA *et al.* (2021), mostram a contribuição da análise de dados escalares agregados por meio da invocação de serviços de dados de proveniência com as soluções DNNProv KUNSTMANN *et al.* (2021) e KerasProv PINA *et al.* (2021b). Entretanto, além de não tratarem especificamente de PINNs, a adoção de tais abordagens fazem uso de sistemas de banco de dados relacionais e APIs demandando uma integração intrusiva com o *framework Modulus*.

Assim, esta dissertação de mestrado tem como objetivo desenvolver uma solução de análise de dados que funcione como um agregador de dados aproveitando os recursos do *framework Modulus*, e que forneça ao especialista de *PINNs* funcionalidades que facilitem a análise comparativa de múltiplos modelos dentro de um determinado experimento. Este agregador já considera uma estrutura típica de organização de diretórios prevista para um experimento desenvolvido no *Modulus*, de maneira que o analista não tenha a necessidade de fazer grandes modificações em sua estrutura de arquivos. A ferramenta visa a oferecer ao usuário opções de exportação dos dados gerados em um formato conveniente para análises dos dados, bem como para a construção de consultas, levando em consideração a estrutura de *tags* geradas a partir de um experimento definido dentro do *framework Modulus*. Assim, a análise

comparativa de modelos em aspectos e parâmetros diversos que considerem dados de treinamento, monitoramento ou mesmo validação definidos dentro do escopo do código de um experimento científico é facilitada, principalmente em contextos onde o pós-processamento de dados obtidos a partir de modelos diversos gerados por diferentes conjuntos de hiperparâmetros torna-se custoso e impeditivo em determinados cenários de pesquisa.

As vantagens da integração da ferramenta em um ambiente de desenvolvimento de modelos *PINN* via *Modulus* são mostradas a partir de experimentos que busquem explorar as funcionalidades do *framework* em relação a desempenho, escalabilidade, apoio a análise de resultados e facilidade de modelagem física de um problema utilizando uma máquina de processamento de alto desempenho (PAD) no supercomputador SDumont - Sistema de Computação Santos Dumont (`https://sdumont.lncc.br`), a fim de que as características presentes no *Modulus* sejam devidamente exploradas de maneira a servirem como base para o desenvolvimento das funcionalidades da ferramenta aqui proposta.

Com isso, a dissertação se propõe a explorar alternativas que facilitem e organizem o fluxograma da etapa pós-processamento de dados via uso da ferramenta desenvolvida, principalmente em relação à análise e comparação com modelos gerados com diferentes conjuntos de hiperparâmetros. Embora o presente trabalho não proponha uma solução de proveniência para a captura de dados ao longo do treinamento como em PINA *et al.* (2021a), busca-se organizar um pacote onde o cientista seja capaz de extrair os dados produzidos de maneira a possibilitar a exploração de questões mais específicas em relação à performance dos diferentes modelos produzidos, aproveitando ao máximo a estrutura nativa oferecida pelo *framework* aqui examinado. A ideia é que, futuramente, uma vez consolidada a solução desenvolvida nessa dissertação, sejam oferecidas ao especialista alternativas tanto de armazenamento de arquivos como de proveniência junto ao uso de um Sistema de Gerenciamento de Banco de Dados (SGBD) para consultas ao longo do treinamento.

Além da presente introdução, essa dissertação encontra-se dividida em outros seis capítulos. O Capítulo 2 apresenta a definição formal de *PINN* considerada nesse estudo, bem como trabalhos já produzidos e *frameworks* relacionados a essa família de modelos. O Capítulo 3 faz a esquematização de como é dada a definição de um experimento para treinamento de uma *PINN* dentro do *Modulus* a partir das funcionalidades do *framework*. O Capítulo 4 apresenta o processo de análise de múltiplos modelos via *Modulus* e *Tensorboard* e suas limitações. O Capítulo 5 trata da ferramenta de agregação de dados proposta, detalhando o objetivo de seu desenvolvimento, sua arquitetura, metodologia de uso e atuais limitações do pacote. O Capítulo 6 discute o emprego da ferramenta em um experimento *Modulus*, bem como as vantagens de sua utilização junto ao processo de análise de resultados e

filtragem automática de modelos. Por fim, encerra-se com o Capítulo 7, onde são levantados tópicos de estudo a serem aprofundados em trabalhos futuros.

Capítulo 2

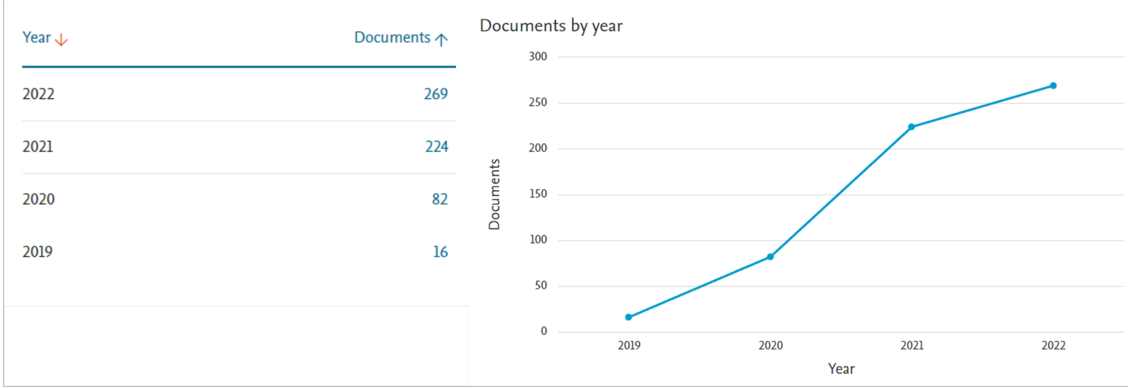
Redes Neurais Guiadas à Física (*PINNs*)

Esse capítulo tem como objetivo apresentar a fundamentação teórica geral do funcionamento das Redes Neurais Guiadas à Física (*PINNs*), estudos desenvolvidos relacionados e *frameworks* de desenvolvimento para tal família de modelos. Desse modo, apresenta-se, na Seção 2.1, a definição de *PINNs* a partir de equações que definem o problema a ser solucionado, na Seção 2.2, diversos trabalhos de pesquisa produzidos a partir de aplicações diversas com *PINNs*, na Seção 2.3, trabalhos relacionados à gerencia de dados para desenvolvimento de modelos de Aprendizado de Máquina e *PINNs*, e por fim, na Seção 2.4, *frameworks* de desenvolvimento de redes guiadas à física.

2.1 Definição

As Redes Neurais Guiadas à Física (*PINNs*) referem-se a uma família de modelos primeiramente apresentada em RAISSI *et al.* (2019), cujo objetivo é resolver, em última medida, problemas físicos caracterizados por equações diferenciais parciais (PDEs). Tal resolução funciona de modo a aproximar a solução de uma PDE a partir do acoplamento de termos residuais junto à função de perda de uma rede neural. Dessa maneira, é possível fazer com que tanto termos referentes às condições de contorno, de dados de campo coletados e de termos residuais relacionados às PDEs governantes do sistema sejam minimizados durante o treinamento do modelo CUOMO *et al.* (2022).

O acoplamento das equações diferenciais governantes de um sistema físico é a grande novidade inerente a essa nova família de modelos que tem cada vez mais recebido atenção da comunidade científica. Isso pode ser atestado vide o aumento do número de trabalhos publicados que foram filtrados com uma pesquisa feita via



Year	Documents
2022	269
2021	224
2020	82
2019	16

Figura 2.1: Quantidade de trabalhos relacionados a *PINNs* publicados anualmente, agregados via Scopus.

Scopus com os termos *TITLE-ABS-KEY (((physic* OR physical) W/2 (informed OR constrained) W/2 "neural network")).* A Figura 2.1 mostra um gráfico com a quantidade de trabalhos publicados anualmente que foram filtrados com tal pesquisa, de 2019 a 2022.

Além disso, as *PINNs* também podem ser vistas como uma estratégia de aprendizado não supervisionado quando trata-se da resolução de problemas que não apresentam dados rotulados de campo, ou de uma estratégia de aprendizado supervisionado quando existem dados rotulados a serem considerados dentro do problema CUOMO *et al.* (2022).

Apesar de existirem diferentes estratégias de acoplamento do conhecimento referente a um sistema como *prior* de um modelo guiado ou informado à física, como por exemplo via introdução de vieses observacionais (com técnicas de produção de dados sintéticos ou *data augmentation*) ou indutivos (com modificação da arquitetura do modelo para que haja a satisfação de determinadas restrições físicas, como a adição de camadas convolucionais que preservam a simetria e invariância) KARNIADAKIS *et al.* (2021), as *PINNs* introduzem tal conhecimento por meio de uma mudança no processo aprendizado do modelo em si, modificando explicitamente termos da equação de perda de uma rede neural, de maneira a adicionar as PDEs do problema como *soft constraints* na função objetivo a ser otimizada HUANG *et al.* (2022).

Para fins de formalização dos conceitos mencionados, são definidas as equações 2.1 e 2.2, que ilustram o formato geral do problema a ser resolvido por uma *PINN*, isto é.

$$\mathcal{L}(\mathbf{u}(t;\mathbf{x});\boldsymbol{\lambda}) = \mathbf{f}(t;\mathbf{x}) \qquad \mathbf{x} \in \Omega, t \in T \tag{2.1}$$

$$\mathcal{B}(\mathbf{u}(t;\mathbf{x})) = \mathbf{g}(t;\mathbf{x}) \qquad \mathbf{x} \in \partial\Omega \tag{2.2}$$

onde as equações 2.1 e 2.2 são definidas no domínio $\Omega \in \mathbb{R}^D$, com seu contorno

em $\partial\Omega$. Além disso, a solução $\mathbf{u}(t;\mathbf{x})$ depende de $t \in [0,T]$, que representa a variável temporal, e $\mathbf{x} = [x_1, x_2, ..., x_D]$, que constitui a variável espacial definida em Ω. Ademais, $\mathcal{L}$ simboliza o operador não linear diferencial característico do problema, $\boldsymbol{\lambda}$ os parâmetros físicos e $\mathbf{f}$ a função inerente aos dados do problema. Por fim, tem-se que $\mathcal{B}$ corresponde ao operador que indica condições iniciais e/ou de contorno (como por exemplo, condições de contorno de *Dirichlet*, *Robin* ou mesmo periódicas) relativas ao problema, e $\mathbf{g}$ a função especificada no contorno.

É interessante notar que tais equações podem vir a representar tanto problemas diretos como inversos. No primeiro caso, o objetivo de um modelo *PINN* seria de aproximar a solução $\mathbf{u}(t;\mathbf{x})$ a partir dos parâmetros físicos $\boldsymbol{\lambda}$ conhecidos do problema, dadas as condições iniciais e/ou de contorno. Já no último caso, o problema se coloca de maneira a serem estimados os parâmetros físicos $\boldsymbol{\lambda}$ a partir de dados coletados de $\mathbf{u}$ em diferentes pontos de $\mathbf{x}$ e t.

Assim, a rede neural treinada aproxima $\mathbf{u}$ a partir de um conjunto de parâmetros $\boldsymbol{\theta}$ obtidos ao longo do treinamento, de modo que $\mathbf{u}_\theta(t;\mathbf{x}) - \mathbf{u}(t;\mathbf{x}) = R(\mathbf{x}, t, \boldsymbol{\theta})$. Considera-se aqui que $\mathcal{R}$ representa o resíduo entre a solução aproximada e a real, o qual deve ser minimizado. Além disso, considera-se também que $\boldsymbol{\theta}$ representa tanto os parâmetros relacionados apenas ao modelo da rede neural no caso de um problema direto, como também os parâmetros $\boldsymbol{\lambda}$ a serem estimados quanto trata-se de um problema inverso.

Desse modo, é necessário que uma *PINN* minimize uma função de perda de formato equivalente à equação 2.3:

$$\mathbf{L}(\boldsymbol{\theta}) = \mathbf{w}_d \mathbf{L}_d(\boldsymbol{\theta}) + \mathbf{w}_u \mathbf{L}_u(\boldsymbol{\theta}) \tag{2.3}$$

onde $\mathbf{L}_u$ representa a função de perda relacionada a aproximação da PDE que busca minimizar $\mathcal{R}(\mathbf{x}, t, \boldsymbol{\theta})$, e $\mathbf{L}_u$ corresponde a uma função de perda relativa aos dados, sejam eles dados oriundos das condições iniciais, de contorno ou mesmo de eventuais dados coletados. Desse modo, o objetivo maior é achar $\boldsymbol{\theta}^* = \arg\min_{\mathbf{x}}(\mathbf{L}(\boldsymbol{\theta}))$, onde $\boldsymbol{\theta}^*$ minimiza a função de perda definida na equação 2.3. No caso de uma *PINN*, diferentes arquiteturas de redes neurais podem ser utilizadas para a minimização da equação geral de perda, como por exemplo redes *feed-foward*, convolucionais ou recorrentes.

Mesmo que não haja nenhuma garantia de que tal abordagem de aprendizado consiga convergir para um mínimo global como bem observado em KOLLMANNSBERGER *et al.* (2021), as *PINNs* têm demonstrado em diferentes estudos o seu potencial para a obtenção de soluções satisfatórias para aplicações diversas. A próxima seção traz mais detalhes de trabalhos do gênero.

2.2 Aplicações

O trabalho apresentado em RAISSI *et al.* (2019) inaugura oficialmente a proposta de aprendizado para obtenção de soluções de PDEs apresentadas na Seção 2.1. Nesse estudo, os autores apresentam resultados satisfatórios obtidos por intermédio de modelos *PINNs* treinados a partir da definição de diferentes problemas. Um deles foi relacionado à equação de *Schrodinger* em domíno contínuo, onde uma *PINN* foi treinada a fim de encontrar a solução para a PDE a partir da penalização de termos da função de perda relacionados aos dados de pontos reunidos, dados das condições de contorno bem como ao erro obtido da equação de *Schrodinger* nos pontos do domínio.

Um outro exemplo apresentado em RAISSI *et al.* (2019) foi uma aplicação para o desenvolvimento de uma solução para a equação de *Allen–Cahn* em regime discreto. Nesse caso, foi separado um conjunto de dados de treino em $t = 0.1s$ aleatoriamente amostrado ao longo do domínio espacial, e o objetivo foi prever a solução em $t = 0.9s$. Mesmo em um passo de tempo de $0.8s$, os resultados obtidos pela *PINN* treinada mostraram-se bastante satisfatórios a partir da comparação com resultados de refêrencia. Por fim, o trabalho de RAISSI *et al.* (2019) apresenta soluções de problemas inversos via o aprendizado de parâmetros para as equações de *Navier-Stokes* (em regime contínuo) e de *Korteweg–de Vries* (em regime discreto).

Já o trabalho de ALMAJID e ABU-AL-SAUD (2022) utiliza uma *PINN* para estimar o escoamento de gás em um meio poroso a partir da equação de *Buckley and Leverett* relacionada à teoria de fluxo fraccional bem como dados observados, a qual é comparada com uma rede neural treinada somente com dados observados. Os autores mostraram que o modelo *PINN* mostrou-se consideravelmente melhor do que uma rede neural convencional, principalmente quando colocada em regime de extrapolação.

O trabalho de KOVACS *et al.* (2022) mostra um tipo de uso de *PINNs* para a solução de problemas de autovalores junto à aproximação de um campo coercivo associado em materiais magnéticos. Além disso, o estudo apresentado em STIASNY *et al.* (2021) utiliza uma *PINN* para aplicação em sistemas de potência, utilizando-a para a estimativa dinâmica de parâmetros de estado.

Já em SUN *et al.* (2020), utiliza-se de uma rede neural profunda, ou *Deep Neural Network (DNN)*, guiada à física para a geração de um modelo para a predição das soluções paramétricas das equações de *Navier-Stokes* utilizadas em problemas de escoamento cardiovascular, de maneira a satisfazer tanto às equações físicas relativas à conservação de massa e de momento, como também as condições iniciais e de contorno, sem a utilização de dados rotulados.

Além disso, o trabalho de RAISSI *et al.* (2020) apresenta um novo *framework* de

extração de campos de velocidade e pressão a partir de imagens, chamado *Hidden Fluid Mechanics (HFM)*. Os autores mostram a eficiência do *HFM* a partir de um problema de dinâmica de fluidos em um aneurisma, mas também mostram que ele também pode ser aplicado para campos eletromagnéticos, por exemplo.

Em um outro estudo, apresentado em MAO *et al.* (2020), os autores procuraram explorar a utilização de *PINNs* a fim de aproximarem soluções modeladas pelas equações de *Euler* para escoamentos aerodinâmicos de alta velocidade. Foram examinados tanto problemas diretos como inversos, em domínios 1D e 2D. Os autores mostraram nesse trabalho uma superioridade das *PINNs* em relação a métodos tradicionais para os problemas inversos considerados.

2.3 Gerenciamento de Dados

Como visto na Figura 2.1, é crescente o interesse da comunidade científica em relação ao desenvolvimento de *PINNs* para a resolução de problemas oriundos de diferentes campos de estudo, como exposto com os trabalhos apresentados na Seção 2.2. Desse modo, é imperativo que o gerenciamento dos dados produzidos durante o treinamento de *PINNs* seja conduzido a fim de que tais dados possam ser utilizados de maneira sistemática com o objetivo de melhorar os resultados obtidos no processo de treinamento dos modelos e de facilitar o processo de análise de dados no geral. Nesse sentido, o crescente avanço no desenvolvimento de estudos e aplicações na área de *Deep Learning* tem ressaltado a importância da criação de técnicas e ferramentas que auxiliem no gerenciamento de dados gerados por modelos de maneira contínua e automática GHARIBI *et al.* (2019). Desse modo, existem diversos trabalhos publicados que se propõem a acrescentar soluções no âmbito dessa cada vez mais relevante demanda.

O trabalho de SILVA *et al.* (2018) apresenta a ferramenta DfAnalyzer, cujo funcionamento se dá de forma a implementar a extração e o registro de dados de proveniência de maneira a possibilitar a execução de consultas durante o treinamento de modelos. Sua arquitetura inclui componentes de extração e indexação bem como de análise de dados via módulos que disponibilizam interfaces para consultas SQL e para visualização de fluxo de dados para o usuário. Foi mostrado no trabalho, a partir de um caso de uso, um *overhead* muito pequeno oriundo do acoplamento da ferramenta.

Já o trabalho de PINA *et al.* (2021b) apresenta a ferramenta *open-source* Keras-Prov, que funciona como uma extensão da biblioteca Keras CHOLLET e OTHERS (2015) para o armazenamento e gerenciamento de dados de proveniência que são capturados durante o treinamento de modelos de aprendizado profundo. O armazenamento desses dados é feito através de um SGBD colunar conhecido, chamado

MonetDB IDREOS *et al.* (2012), de maneira a oferecer ao usuário a possibilidade de fazer consultas analíticas SQL com o apoio de uma interface gráfica intuitiva, que auxilia o especialista na seleção de modelos.

Nessa mesma linha, o trabalho de PINA (2020) apresenta o desenvolvimento da ferramenta CNNProv, cuja maior contribuição é disponibilizar uma solução capaz de efetuar a captura, armazenamento e disponibilização *online* para consulta do usuário de dados de proveniência obtidos durante o treinamento de redes neurais convolucionais. A ferramenta é testada a partir do treinamento de duas redes convolucionais: a AlexNet KRIZHEVSKY *et al.* (2012), (para a classificação de imagens de flores) e a DenseED HUANG *et al.* (2018) (para a análise de dados de imageamento sísmico). Para ambos os casos, a análise experimental mostra que o tempo de *overhead* medido com a utilização da CNNProv foi desprezível.

Já o trabalho de SILVA *et al.* (2021) apresenta uma solução de proveniência de dados desenvolvida para uma *PINN* treinada para um problema inverso governado pela equação fatorada de Eikonal, utilizando a solução DNNProv PINA *et al.* (2021a). Foi avaliada a qualidade de integração da solução de proveniência ao ambiente em que foi desenvolvido o modelo *PINN* via medição de *overhead*, bem como também foi estudado a conveniência geral da implantação de uma base de dados de proveniência, onde é possível fazer consultas inteligentes relacionadas a avaliação de diferentes modelos treinados com conjuntos distintos de hiperparâmetros. No entanto, apesar de apresentarem um avanço em experimentos com *PINNs*, esses trabalhos atuam sobre *frameworks* não voltados a *PINNs*, fazendo com que o usuário tenha que desenvolver componentes já disponíveis no Modulus.

2.4 *Frameworks* de desenvolvimento

Com o aumento de produções científicas relacionadas a *PINNs*, foram desenvolvidos também diversos pacotes de *software* a fim de facilitar a geração e o treinamento desse tipo de modelo. Em sua grande maioria, tais pacotes aproveitam a grande eficiência computacional dos métodos ja empregados de otimização e diferenciação automática via *backpropagation* de *frameworks* de *Deep Learning* popularmente conhecidos, como TensorFlow ABADI *et al.*, Keras CHOLLET e OTHERS (2015) e Pytorch PASZKE *et al.* (2019). Nesse sentido, dentre o conjunto de *frameworks* desenvolvidos para o desenvolvimento de *PINNs*, destacam-se, por exemplo, pacotes como *DeepXDE* LU *et al.* (2021), *SciANN* HAGHIGHAT e JUANES (2021), *NeuroDiffEQ* CHEN *et al.* (2020) e o *framework* explorado nesta dissertação *NVIDIA Modulus* HENNIGH *et al.* (2021)

O *framework Modulus* foi selecionado para o estudo desenvolvido neste trabalho por oferecer um conjunto de funcionalidades maior junto à definição de redes *PINNs*

(por exemplo, mais opções de arquitetura de rede e suporte para resolução de problemas definidos em domínios geométricos mais complexos), bem como por também dar suporte a tarefa de análise de resultados de maneira nativa via *Tensorboard*, como mostrado posteriormente nas Seções 3.1 e 4.1. A seguir são expostos alguns detalhes de cada um dos pacotes mencionados, além da Tabela 2.1, onde são resumidas algumas das funcionalidades disponibilizadas por cada um desses *frameworks*.

Tabela 2.1: *Frameworks* de desenvolvimento de *PINNs*.

Framework	**Restrição para condições iniciais/contorno**	**Suporte a Domínios 3D**	**Suporte a *CSG* e *TSG***	**Arquiteturas de rede disponíveis**
DeepXDE	*soft*	Sim	Não	*Feed-Foward*
SciANN	*soft*	Sim	Não	*Feed-Foward*
NeuroDiffEQ	*hard*	Não	Não	*Feed-Foward*
Modulus	*soft*	Sim	Sim	*Feed-Foward*, *DeepONet*, *SiReNs*, *Fourier Neural Operator*

2.4.1 *DeepXDE*

DeepXDE LU *et al.* (2021) é uma biblioteca *open source* escrita em *python* que tem como objetivo facilitar a declaração e o uso de modelos *PINN* a partir da ênfase na flexibilidade junto à definição de um problema. No *DeepXDE*, isso é realizado com a facilidade do uso de módulos e diretivas presentes no pacote, que permitem a definição de condições iniciais e/ou de contorno diversas, bem como também a definição de geometrias mais complexas a partir da técnica de geometria sólida construtiva, ou *constructive solid geometry* (CSG). Sua implementação conta com a técnica de refinamento adaptativo baseado em resíduo, ou *residual-based adaptive refinement* (*RAR*), cuja abordagem otimiza a distribuição de pontos ao longo do domínio durante o treinamento da *PINN* em localidades que tenham um residual mais elevado. Apesar da flexibilidade em definição de *PINNs*, existem limitações quanto ao tipo de rede disponibilizada pelo pacote, que são limitadas em redes do tipo *feed-foward* e redes neurais residuais.

2.4.2 *SciANN*

O *SciANN* HAGHIGHAT e JUANES (2021) é um pacote para definição e treinamento de redes *PINNs* implementado como um *wrapper* de alto nível da biblioteca Keras CHOLLET e OTHERS (2015). Seus módulos possibilitam que o usuário

defina de maneira particular a função de perda a ser minimizada pela *PINN*. Isso permite certa flexibilidade junto à definição do problema como um todo, mas também requer um maior nível de envolvimento do usuário junto a implementação do esquema de treinamento do modelo *PINN*. Porém, há também uma limitação junto aos tipos de arquitetura de rede oferecidas para definição no pacote, que disponibiliza apenas redes do tipo *feed-foward* tradicionais.

2.4.3 *NeuroDiffEQ*

NeuroDiffEQ CHEN *et al.* (2020) é uma biblioteca de implementação de *PINNs* que utiliza o pacote PyTorch PASZKE *et al.* (2019) como *backend*. Sua principal característica é possibilitar a definição estrita dos termos relativos às condições iniciais e/ou de contorno via própria construção da rede. Apesar dessa característica, tal biblioteca tem a limitação de não dar suporte à definição de condições de contorno e/ou iniciais mais gerais, bem como a definição de problemas é limitada apenas para domínios 2D.

2.4.4 *NVIDIA Modulus*

O *NVIDIA Modulus*, anteriormente conhecido como *SimNet* HENNIGH *et al.* (2021), é um *framework* de desenvolvimento de *PINNs* que busca oferecer tanto à indústria quanto à academia uma alternativa sólida para geração e treinamento de modelos de redes neurais guiadas por física. O *Modulus* se apresenta como um *toolset* completo no que tange à implementação de *PINNs*, oferecendo suporte junto à definição e parametrização de geometrias mais complexas via módulos de CSG e *Tessellated geometry* (TSG).

Além disso, o *NVIDIA Modulus* oferece também como funcionalidade a possibilidade de utilizar os pesos da equação de perda definidos na equação 2.3 como funções do espaço a partir de *Signed Distance Functions* (SDFs), obtendo ganhos de convergência de modo a atribuir valores menores aos pesos em áreas de descontinuidade do domínio.

Junto a isso, soma-se o fato de tal *framework* disponibilizar em seu pacote diversos tipos de arquitetura mais avançadas, como *Fourier Neural Operator*, *DeepONet* e *Sinusoidal Representation Networks* (*SiReNs*). Além do repertório disponível de tipos de arquiteturas de rede ser um diferencial notável em relação aos demais *frameworks* de *PINNs* apresentados, essas arquiteturas tem como objetivo principal oferecer ao usuário mais possibilidades junto a convergência de soluções menos enviesadas para baixas frequências.

O trabalho de HENNIGH *et al.* (2021), além de apresentar algumas das funcionalidades básicas e particularidades de implementação da ferramenta, apresenta

também alguns exemplos de casos de uso onde fez-se o uso do pacote para a resolução dos problemas colocados, justamente para fins de atestar não só a eficiência da biblioteca, mas também de sua grande capacidade para o treinamento de *PINNs* junto à aplicações mais complexas. Foram abordados problemas de multifísica como o uso de *PINNs* para a solução de equações de transferência de calor em geometrias complexas e parametrizadas de dissipadores e de escoamento 3D de sangue em um aneurisma intracranial. Apesar da complexidade dos problemas tratados, as soluções obtidas pelo *NVIDIA Modulus* não só apresentaram resultados satisfatoriamente próximos de *solvers* comerciais utilizados como referência, como também foram alcançadas em tempo significativamente menor do que o observado nos demais *solvers*, para alguns exemplos.

A documentação atual do *Modulus* pode ser acessada em `https://docs.nvidia.com/deeplearning/modulus/`, onde é possível acompanhar não só a utilização do *framework* para alguns exemplos, como também a documentação em si de cada um dos módulus definidos e disponíveis. Além de sua documentação, é possível também fazer o *download* de alguns exemplos de experimentos que utilizam o *framework Modulus* para o treinamento de *PINNs*. Tais exemplos podem ser utilizados de maneira a melhor entender o funcionamento básico do *framework*, tanto em termos de estrutura de código quanto também em termos de definição de módulos necessários junto à definição de um problema e do *solver* a ser treinado.

Capítulo 3

NVIDIA Modulus - Experimentos preliminares

Esse capítulo tem como objetivo avaliar os recursos de desenvolvimento de *PINNs* e de análise de resultados a partir do *Modulus*, bem como detalhar o processo de contrução de um experimento de geração de modelos *PINNs*. Para tanto, um experimento do banco de exemplos do *Modulus* foi executado utilizando-se do Supercomputador SDumont, e seus resultados foram analisados em ambiente *Desktop*. A Seção 3.1 esquematiza a implementação de *PINNs* no *Modulus*, a partir da análise das funcionalidades do *framework* expostas em um dos exemplos de experimentos disponibilizados.

3.1 Análise da implementação de *PINNs* no *Modulus*

A presente Seção tem como objetivo representar um primeiro esforço no sentido de compreender a dinâmica de implementação de Redes Neurais Guiadas a Física no *framework Modulus* a partir da análise preliminar de um dos experimentos do banco de exemplos disponibilizado em `https://developer.nvidia.com/modulus-downloads`, de maneira que tal dinâmica possa servir como base para um entendimento concreto relacionado à estrutura de um experimento definido no *Modulus*, tanto em termos de estrutura de diretórios, quanto em termos de construção de código para a definição de classes ou objetos necessários para que seja gerado um modelo *PINN*. Além disso, cabe também um esforço de compreensão sobre o processo de geração de dados das *PINNs* implementadas com o *Modulus*, de forma que tais dados possam também servir para futuros estudos de pós-processamento e análise de desempenho de diferentes configurações treinadas.

A partir da exploração do experimento investigado, a presente Seção produz

como resultado o entendimento das funcionalidades gerais básicas e estruturais para a criação de uma *PINN*, explorando como declarar diferentes objetos para a definição de uma rede que será treinada a partir de uma função de perda baseada tanto em equações físicas, quanto em condições de contorno, e também a estruturação de um *workflow* de definição e implementação de *PINNs* e de geração de dados a partir do caso analisado. Além disso, apresenta-se diversas funcionalidades já disponíveis no *framework* relativas a análise de dados via *TensorBoard*. Apesar de ficar claro que a definição de um modelo *PINN* torna-se facilitada via uso do *Modulus*, visto que existem classes específicas para a declaração de restrições físicas e também das equações a serem utilizadas para o treinamento dos modelos, faz-se necessária uma investigação mais aprofundada em relação a existência de funcionalidades nativas do *Modulus* de análise e pós-processamento de dados gerados em um experimento.

3.1.1 Definição do Experimento Preliminar de Escoamento em Seção Cilíndrica

Ao fazer o *download* do banco de exemplos, descompactar e abrir seu diretório, é possível ver uma coleção de pastas representando diferentes problemas resolvidos com a utilização do pacote *Modulus*, como mostra a Figura 3.1.

O experimento preliminar a ser explorado é relativo a um exemplo que trata de um problema clássico de escoamento de uma sessão de um cilindro em regime permanente, ou seja, sem dependência temporal. Tal exemplo é disponibilizado na pasta *cylinder*, onde encontra-se apenas um *script* em *python*, além de uma pasta chamada *openfoam*, que dispõe de um arquivo .csv que contém dados simulados do problema físico em questão. Tais dados são utilizados de modo a validar os resultados da rede definida dentro do *script* em python. Também encontra-se disponível uma pasta *conf*, que contém um arquivo chamado *config.yaml*, cujo propósito é o de armazenar informações devidamente formatadas sobre a arquitetura de rede a ser utilizada no treinamento da *PINN* definida no problema. Tal arquivo é operado a partir de uma extensão do pacote *Hydra*, definida no próprio *framework Modulus*, que tem o objetivo de gerenciar configurações definidas para aplicações mais complexas. Mais informações podem ser encontradas em `https://hydra.cc/`. A Figura 3.2 mostra o conteúdo da pasta *cylinder*.

3.1.2 Organização do Código

Ao analisar a estrutura do código somente, é possível notar primeiramente, após a declaração dos pacotes necessários, a existência de um único método principal chamado *run()*, como ilustra a Figura 3.3. Vê-se que tal módulo conta com um decorador em *python* representando a extensão do pacote *Hydra* feita dentro do

aneurysm	Pasta de arquivos
annular_ring	Pasta de arquivos
anti_derivative	Pasta de arquivos
bracket	Pasta de arquivos
chip_2d	Pasta de arquivos
cylinder	Pasta de arquivos
darcy	Pasta de arquivos
discontinuous_galerkin	Pasta de arquivos
fpga	Pasta de arquivos
fuselage_panel	Pasta de arquivos
helmholtz	Pasta de arquivos
ldc	Pasta de arquivos
limerock	Pasta de arquivos
ode_spring_mass	Pasta de arquivos
plane_displacement	Pasta de arquivos
seismic_wave	Pasta de arquivos
super_resolution	Pasta de arquivos
surface_pde	Pasta de arquivos
taylor_green	Pasta de arquivos
three_fin_2d	Pasta de arquivos
three_fin_3d	Pasta de arquivos
turbulent_channel	Pasta de arquivos
wave_equation	Pasta de arquivos
waveguide	Pasta de arquivos
.gitattributes	Arquivo Fonte Git ...
.gitignore	Arquivo Fonte Git ...
README.md	Arquivo Fonte Ma...

Figura 3.1: Diretório de Exemplos resolvidos com o pacote *Modulus*.

conf	Pasta de arquivos
openfoam	Pasta de arquivos
cylinder_2d.py	Arquivo Fonte Pyt...

Figura 3.2: Conteúdo do diretório *cylinder*.

framework Modulus responsável por importar as configurações da rede definidas na pasta *conf/* e no arquivo *config.yaml*.

Dentro do método *run()*, vê-se uma subdivisão clara, relacionada tanto à definição do problema em si (geometria e domínio do problema), quanto à definição de

```
@modulus.main(config_path="conf", config_name="config")
def run(cfg: ModulusConfig) -> None:...


if __name__ == "__main__":
    run()
```

Figura 3.3: Declaração do método principal *run()*.

classes que desempenham diferentes papeis junto à geração do modelo final. Desse modo, tais definições podem ser listadas como sendo as de:

- Pontos de definição da Equação Diferencial e nós da Rede Neural;
- Geometria do problema;
- Restrições Físicas e de Contorno;
- Objetos Adicionais;
- Solver e Treinamento;

3.1.2.1 Pontos de definição da Equação Diferencial e nós da Rede Neural

A Figura 3.4 expõe o trecho do código onde são definidos os nós referentes à equação física e à arquitetura da rede neural a ser empregada, bem como suas entradas (coordenadas espaciais x e y) e saídas (velocidade horizontal u, velocidade vertical v e pressão p).

```
# make list of nodes to unroll graph on
ns = NavierStokes(nu=0.02, rho=1.0, dim=2, time=False)
normal_dot_vel = NormalDotVec(["u", "v"])
flow_net = instantiate_arch(
    input_keys=[Key("x"), Key("y")],
    output_keys=[Key("u"), Key("v"), Key("p")],
    cfg=cfg.arch.fully_connected,
)
nodes = (
    ns.make_nodes()
    + normal_dot_vel.make_nodes()
    + [flow_net.make_node(name="flow_network", jit=cfg.jit)]
)
```

Figura 3.4: Código da definição dos nós da equação física e da rede neural.

Nesse caso, determina-se a equação a ser resolvida, que no caso, é o conjunto de equações de *Navier Stokes* que contém as equações de continuidade e momento a serem definidas como restrições físicas acopladas ao treinamento da rede. Além disso, é declarado o nó da rede neural a ser utilizada junto ao treinamento da *PINN*, cuja configuração é extraída do arquivo *config.yaml*, que define a arquitetura da rede. No presente caso, foi utilizada uma rede *fully connected*, com seis camadas e com 512 neurônios em cada (valores definidos por padrão para tal arquitetura). A Figura 3.5 mostra o arquivo de configuração *config.yaml*, onde além da arquitetura de rede, estão definidos outros parâmetros relativos por exemplo, à taxa de decaimento do *learning rate* (*scheduler*), bem como ao registro de resultados de treinamento e dos diferentes *batch sizes* utilizados ao longo do problema.

```yaml
defaults :
  - modulus_default
  - arch:
      - fully_connected
  - scheduler: tf_exponential_lr
  - optimizer: adam
  - loss: sum
  - _self_

scheduler:
  decay_rate: 0.95
  decay_steps: 2000

training:
  rec_results_freq : 1000
  rec_constraint_freq: 10000
  max_steps : 200000

batch_size:
  inlet: 640
  outlet: 640
  walls: 640
  no_slip: 256
  interior: 6400
```

Figura 3.5: Conteúdo do arquivo de configuração *config.yaml*.

3.1.2.2 Definição da Geometria do Problema

Após a definição da equação diferencial e da arquitetura de rede, o código segue com a definição da geometria do problema, a partir da definição das diferentes seções a serem utilizadas futuramente na construção das restrições físicas e de contorno. A Figura 3.6 mostra a definição de tais seções.

```
# simulation params
channel_length = (-10.0, 30.0)
channel_width = (-10.0, 10.0)
cylinder_center = (0.0, 0.0)
cylinder_radius = 0.5
inlet_vel = 1.0
# define sympy variables to parametrize domain curves
x, y = Symbol("x"), Symbol("y")

# define geometry
channel = Channel2D(
    (channel_length[0], channel_width[0]), (channel_length[1], channel_width[1])
)
inlet = Line(
    (channel_length[0], channel_width[0]),
    (channel_length[0], channel_width[1]),
    normal=1,
)
outlet = Line(
    (channel_length[1], channel_width[0]),
    (channel_length[1], channel_width[1]),
    normal=1,
)
wall_top = Line(
    (channel_length[1], channel_width[0]),
    (channel_length[1], channel_width[1]),
    normal=1,
)
cylinder = Circle(cylinder_center, cylinder_radius)
volume_geo = channel - cylinder
```

Figura 3.6: Código da definição da geometria do problema físico proposto.

Nesse caso, a geometria do domínio do problema conta com um retângulo de largura de 20 unidades e comprimento de 40 unidades, com uma sessão cilíndirica, representada por um círculo de raio de 0,5 unidades, posicionada no centro do sistema de coordenadas. A Figura 3.7 ilustra tal geometria.

3.1.2.3 Definição das Restrições Físicas e de Contorno

O código dispõe também de uma seção onde são definidas as restrições a serem consideradas no problema. Nessa seção, são definidas as equações a serem consideradas pela *PINN* no processo de treinamento, bem como os valores para as condições de contorno. Tais restrições seriam então acopladas ao processo de treinamento da rede de modo a criar uma função de perda que seja uma composição de uma parcela do

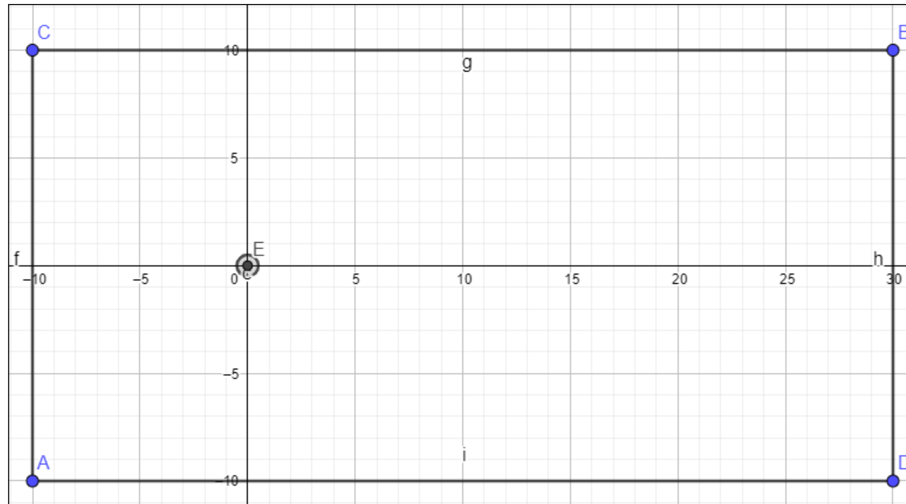


Figura 3.7: Figura ilustrativa da geometria do problema.

resíduo referente às equações físicas, bem como uma parcela relacionada às condições de contorno, como mostra a Equação 2.3, que seria então minimizada via *Modulus*.

No problema analisado, as condições de contorno são definidas nas diferentes fronteiras da geometria (tanto do retângulo, quanto da seção cilíndrica), e as equações físicas são definidas para os pontos do interior da geometria, onde esses obedecem às equações de continuidade e de momento nas direções de x e y. A Figura 3.8 traz o trecho do código onde a classe de treinamento da rede é definida. É possível observar nas Figuras 3.8a e 3.8b que primeiramente ocorre a definição do domínio do problema, e a partir disso, cada uma das restrições definidas é adicionada a este.

3.1.2.4 Definição de Objetos Adicionais

A partir da definição da equação física a ser observada, da geometria de rede e das restrições do domínio, é possível realizar o treinamento do modelo e então, gerar a *PINN* como saída do experimento. Porém, o *framework* do *Modulus* permite a declaração de objetos adicionais, que podem servir a diferentes propósitos junto ao processo de treinamento de um modelo.

No presente exemplo, o código segue com a declaração de um objeto de validação, o qual utiliza os dados da solução do problema, obtidos a partir do *solver open source* chamado *openfoam*. Tal objeto é então acrescentado ao domínio anteriormente definido, de maneira a avaliar, durante o treinamento, a validade dos dados gerados pelo modelo em comparação aos valores previamente reconhecidos como referência de resultados. Desse modo, os dados do *solver openfoam* acabam servindo como uma referência aos resultados gerados durante o treinamento, de modo que a diferença entre a referência e os resultados gerados é avaliada pelos objetos de validação, mas não fazem parte do treinamento do modelo em si. A Figura 3.9 mostra o trecho do código onde há a declaração do objeto de validação, bem como

```python
# make domain
domain = Domain()

# inlet
inlet = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=inlet,
    outvar={"u": 1, "v": 0},
    batch_size=cfg.batch_size.inlet,
)
domain.add_constraint(inlet, "inlet")

# outlet
outlet = PointwiseBoundaryConstraint(
    nodes=nodes, geometry=outlet, outvar={"p": 0},
    batch_size=cfg.batch_size.outlet
)
domain.add_constraint(outlet, "outlet")

# full slip (channel walls)
walls = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=channel,
    outvar={"u": 1, "v": 0},
    batch_size=cfg.batch_size.walls,
)
domain.add_constraint(walls, "walls")
```

(a) Restrições definidas em entrada, saída e deslizamento.

```python
# no slip
no_slip = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=cylinder,
    outvar={"u": 0, "v": 0},
    batch_size=cfg.batch_size.no_slip,
)
domain.add_constraint(no_slip, "no_slip")

# interior contraints
interior = PointwiseInteriorConstraint(
    nodes=nodes,
    geometry=volume_geo,
    outvar={"continuity": 0,
            "momentum_x": 0,
            "momentum_y": 0},
    batch_size=cfg.batch_size.interior,
    bounds={x: channel_length, y: channel_width},
)
domain.add_constraint(interior, "interior")
```

(b) Restrições definidas como sem deslizamento no interior da geometria.

Figura 3.8: Códigos das Definições das Restrições Físicas e de Contorno.

sua adição ao domínio do problema. Para o presente exemplo, foi feita uma adição ao parâmetro *plotter* junto à definição do objeto de validação.

```python
# Loading validation data from CSV
mapping = {"Points:0": "x", "Points:1": "y", "U:0": "u", "U:1": "v", "p": "p"}
openfoam_var = csv_to_dict(
    to_absolute_path("openfoam/cylinder_nu_0.020.csv"), mapping
)
openfoam_invar_numpy = {
    key: value for key, value in openfoam_var.items() if key in ["x", "y"]
}
openfoam_outvar_numpy = {
    key: value for key, value in openfoam_var.items() if key in ["u", "v", "p"]
}
openfoam_validator = PointwiseValidator(
    openfoam_invar_numpy, openfoam_outvar_numpy, nodes, plotter=ValidatorPlotter()
)
domain.add_validator(openfoam_validator)
```

Figura 3.9: Código da definição do objeto de validação.

Além disso, é possível definir objetos de monitoramento (cujo objetivo principal é permitir o acompanhamento de variáveis de interesse durante o treinamento) e de

inferência (que tem o papel de inferir o valor das variáveis de saída do modelo em pontos definidos do domínio do problema durante o treinamento). Apesar do código original referente ao exemplo estudado não conter a definição de tais objetos, é feita a sua adição dentro do experimento, como mostram as Figuras 3.10a e 3.10b.

```
# add monitor
global_monitor = PointwiseMonitor(
    cylinder.sample_interior(100, bounds={x: channel_length, y: channel_width}),
    output_names=["continuity", "momentum_x", "momentum_y", "p"],
    metrics={
        "mass_imbalance": lambda var: torch.sum(
            var["area"] * torch.abs(var["continuity"])
        ),
        "momentum_imbalance": lambda var: torch.sum(
            var["area"]
            * (torch.abs(var["momentum_x"]) + torch.abs(var["momentum_y"]))
        ),
        "pressure_drop": lambda var: torch.max(var['p']),
    },
    nodes=nodes,
)
domain.add_monitor(global_monitor)
```

(a) Definição do objeto de monitoramento.

```
# add inferencer data
grid_inference = PointwiseInferencer(
    openfoam_invar_numpy,
    ["u", "v", "p"],
    nodes,
    batch_size=1024,
    plotter=InferencerPlotter(),
)
domain.add_inferencer(grid_inference, "inf_data")
```

(b) Definição do objeto de inferência.

Figura 3.10: Códigos das Definições dos objetos adicionais de monitoramento e inferência.

3.1.2.5 Definição do Solver e Treinamento

Por fim, após as definições dos objetos adicionais de validação, monitoramento e inferência, define-se o *Solver* a partir do domínio do problema já estabelecido anteriormente, e das configurações de rede importadas via *config.yaml*, também já devidamente registradas. A Figura 3.11 mostra a definição do *Solver* e a chamada do método *solve()*, que inicia o treinamento da rede e conclui o método *run()*.

```
# make solver
slv = Solver(cfg, domain)

# start solver
slv.solve()
```

Figura 3.11: Código da definição da classe do Solver.

3.1.3 Geração de Dados de Resultados

Após o treinamento do modelo, uma série de dados e novos diretórios referentes a diferentes tipos de arquivos são gerados dentro da pasta */outputs/cylinder_2d*, de acordo com a frequência de registro de modelos especificada no parâmetro de configuração *'rec_results_freq'*. No caso do exemplo em análise, cinco diretórios são criados: um chamado *.hydra/*, que contém arquivos relativos à configuração completa da rede treinada bem como do pacote *Hydra*, e outros quatro relacionados às restrições definidas para o problema (diretório de nome *constraints/*) e aos objetos adicionais declarados e incorporados ao domínio do problema (objetos de validação, monitoramento e inferência, com diretórios de nome *validators/*, *monitors/* e *inferencers/*, respectivamente).

O diretório de nome *.hydra/*, como mencionado, contém um arquivo relacionado à configuração completa da *PINN* salva, de nome *config.yaml*. A Figura 3.12 mostra, de maneira resumida, o conteúdo do arquivo.

Já no diretório *constraints/*, são registrados os dados computados pela rede para cada uma das restrições adicionadas no domínio via método *add_constraint()*, em formato *.vtp* os quais podem ser lidos via software *Paraview*. Nesse caso, foram registrados os arquivos *inlet.vtp*, *interior.vtp*, *no_slip.vtp*, *outlet.vtp* e *walls.vtp*, os quais são referentes a cada uma das restrições adicionadas ao domínio como mostram as Figuras 3.8a e 3.8b. A Figura 3.13 mostra a continuidade esperada e predita para o interior da geometria mostrada no arquivo *interior.vtp*.

Já nas pastas *validators/* e *inferencers/*, nota-se uma distribuição semelhante de arquivos: um arquivo em formato *.vtp*, e outros três arquivos em formato *.png*, um referente a cada variável de saída do problema (velocidade horizontal u, velocidade vertical v e pressão p). O primeiro tipo de arquivo (*.vtp*) guarda as informações relativas às curvas de contorno para cada um dos objetos declarados. No caso do objeto de validação, encontram-se as curvas de contorno relativas aos valores verdadeiros e preditos pela rede para cada uma das grandezas de saída. Já no caso

```
training: <13 keys>
profiler: <4 keys>
network_dir: .
initialization_network_dir: ''
save_filetypes: vtk
summary_histograms: false
jit: true
device: ''
debug: false
run_mode: train
arch: <1 key>
loss: <2 keys>
optimizer: <7 keys>
scheduler: <4 keys>
batch_size: <5 keys>
custom: ???
```

Figura 3.12: Configuração completa da *PINN* especificada em *.hydra/config/config.yaml*.

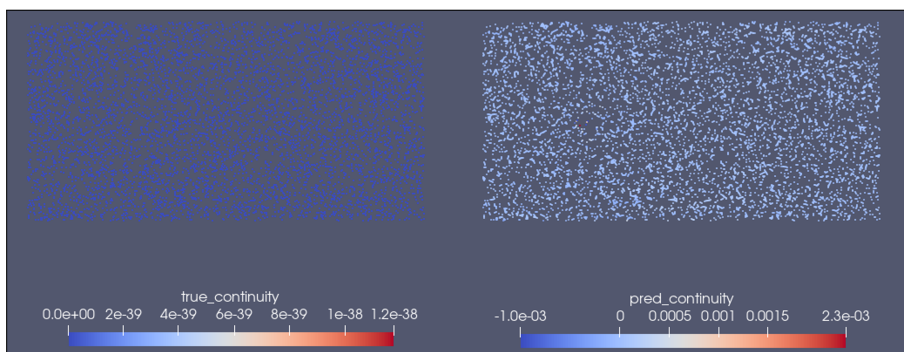
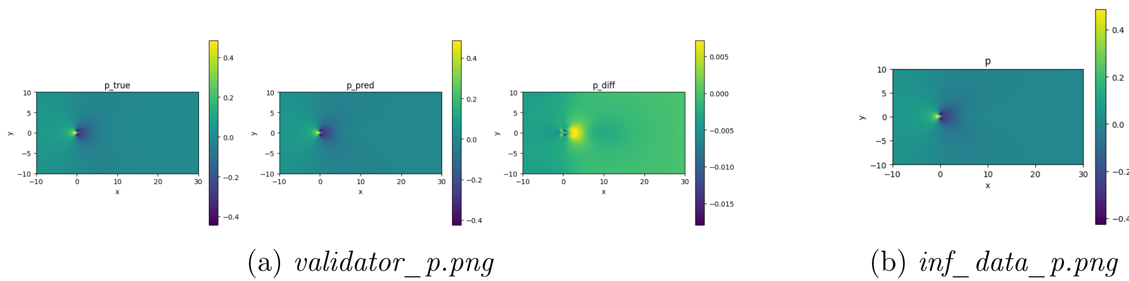


Figura 3.13: Continuidade esperada e predita para o interior da geometria via *Paraview*.

do objeto de inferência, apenas os valores preditos pela rede nos pontos especificados. Já os arquivos *.png* contém, no caso do objeto de validação, as curvas de contorno



(a) *validator_ p.png* (b) *inf_ data_ p.png*

Figura 3.14: Arquivos de imagem relativos aos objetos de validação e inferência, para a variável de saída de pressão.

verdadeiras do problema, as preditas pelo modelo e a diferença entre ambas em cada uma das imagens relacionadas às variáveis de saída. No caso do objeto de inferência, cada imagem contém apenas a curva de contorno predita pelo modelo para cada variável de saída. A Figura 3.14 mostra as imagens criadas para os objetos de validação e inferência relativos à variável de saída de pressão.

Por fim, dentro do diretório *monitors/* são encontrados arquivos .csv, trazendo os valores das diferentes métricas definidas para o objeto de monitoramento adicionado ao domínio. No presente caso, são criados três arquivos em formato .csv, chamados *mass_ imbalance.csv*, *momentum_ imbalance.csv* e *pressure_ drop.csv*, justamente o nome dado às métricas de monitoramento definidas, como mostra a Figura 3.10a. Dentro de cada uma dessas tabelas, são registrados os valores dessas diferentes métricas para cada passo registrado de treinamento. A Figura 3.15 mostra a evolução do valor da queda de pressão computada no arquivo *pressure_ drop.csv* Também é possível analisar tais valores via *TensorBoard*.

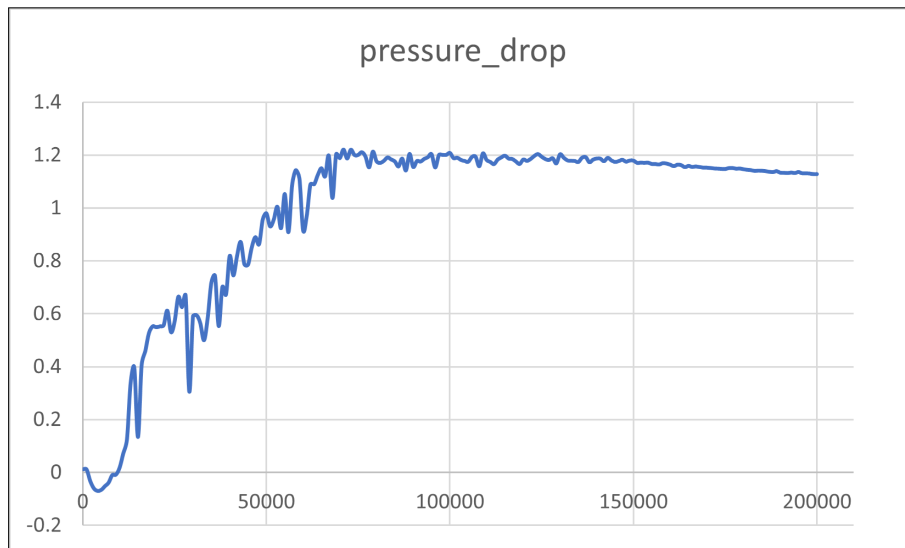


Figura 3.15: Monitoramento da evolução dos valores de queda de pressão ao longo do treinamento.

3.1.4 Análise de dados via *TensorBoard*

Os diferentes erros produzidos pela rede ao longo do treinamento são também analisados via *TensorBoard*, que é um *toolkit* para o monitoramento de métricas de modelos de *Machine Learning* e *Deep Learning*. Ao longo do treinamento, é possível acompanhar o erro total produzido pela rede, bem como os diferentes erros de treinamento, e métricas de validação e monitoramento. Mais detalhes sobre a ferramenta podem ser encontrados em `https://www.tensorflow.org/tensorboard/get_started`.

Ao rodar o *Tensorboard* é possível notar a presença de quatro abas: a primeira delas chamada *scalar*, cujos gráficos explicitam a evolução de métricas de erro de treinamento (para cada uma das restrições definidas) e de validação bem a evolução das métricas definidas de monitoramento; a segunda aba chamada *images*, que mostra a evolução dos mapas de contorno para os objetos de validação e inferência definidos; a terceira chamada *text*, que simplesmente mostra um texto resumido referente à configuração da rede treinada; e a quarta aba chamada *time series*, que basicamente reúne um compilado de todos os gráficos que evoluiram ao longo do treinamento. Cada uma dessas abas será explorada em ordem.

Ao acessar a primeira aba, nota-se a presença de três campos distintos, os quais contém gráficos de séries temporais. O campo *Train* é relacionado a evolução dos erros relativos às restrições definidas no domínio do problema. A Figura 3.16 mostra os gráficos presentes em tal campo, onde percebe-se a presença tanto do erro total (*loss_ aggregated*), quanto dos erros das restrições definidas, bem como também da evolução do *learning rate* ao longo do treinamento.

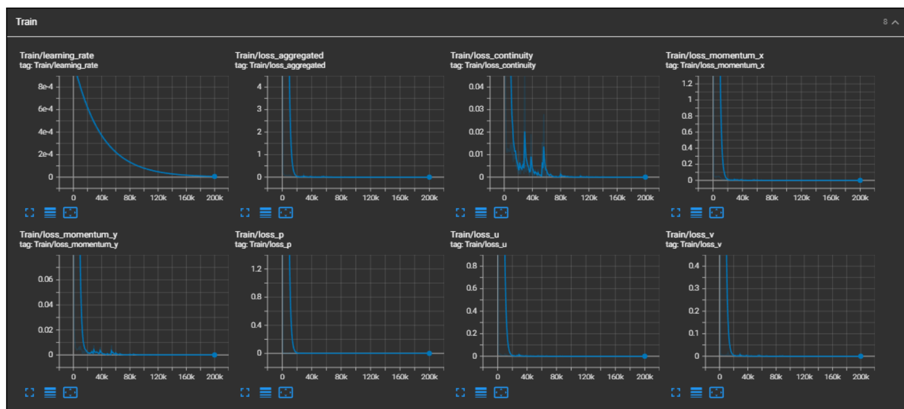


Figura 3.16: Gráficos da evolução dos erros de treinamento e *learning rate* via *TensorBoard*.

O segundo campo, chamado *Validators*, contém os gráficos referentes a evolução dos erros de validação, como mostra a Figura 3.17. Por fim, o terceiro campo chamado *Monitors* mostra a evolução das métricas definidas para o objeto de monitoramento adicionado ao domínio do problema, como ilustra a Figura 3.18.

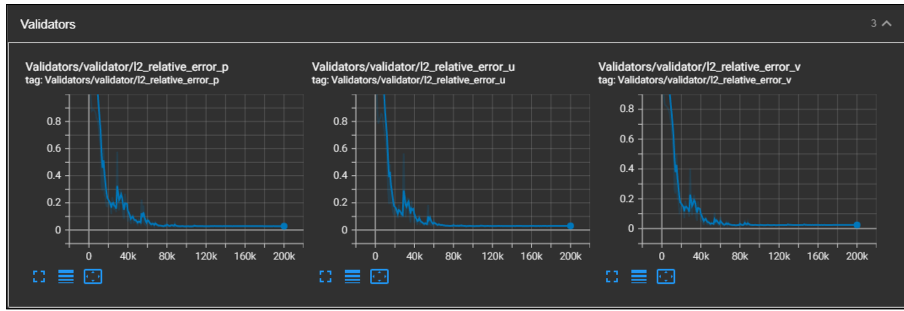


Figura 3.17: Gráficos da evolução dos erros de validação via *TensorBoard*.

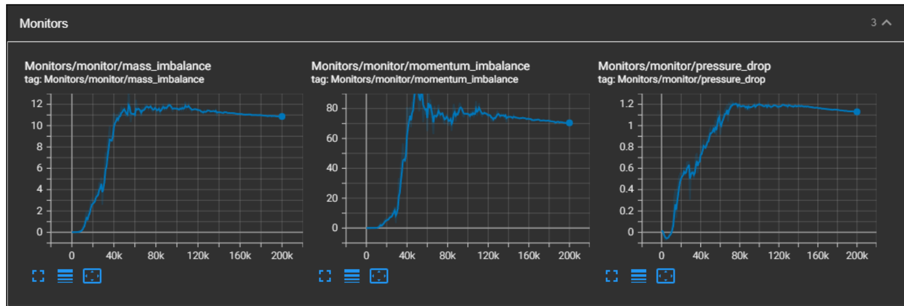


Figura 3.18: Gráficos das grandezas de monitoramento via *TensorBoard*.

Já dentro da segunda aba, chamada *Images*, é possível notar apenas dois campos distintos que mostram a evolução dos mapas de contorno de acordo com os objetos de validação e inferência definidos. Dentro do primeiro campo, chamado *Inferencers*, estão presentes os mapas de contorno para cada uma das variáveis de saída em função de x e y, como mostra a Figura 3.19.

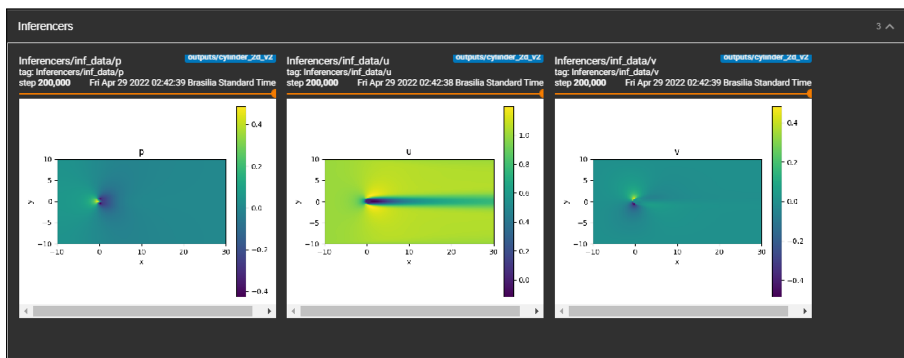


Figura 3.19: Mapas de contorno referentes ao objeto de inferência via *TensorBoard*.

Dentro do segundo campo da aba *Images*, chamado de *Validators*, encontram-se a evolução dos mapas de contorno relativos aos valores verdadeiros das variáveis de saída, os preditos pela rede e a diferença entre eles. A Figura 3.20 mostra tal campo. É importante frisar que a criação dos campos *Inferences* e *Validators* na aba *Images* foi implementada pelas funções *ValidatorPlotter()* e *InferencePlotter()*, utilizadas no parâmetro *'plotter'* dos objetos de validação e inferência respectivamente.

A terceira aba, chamada *Text*, mostra apenas um texto relacionado à configura-

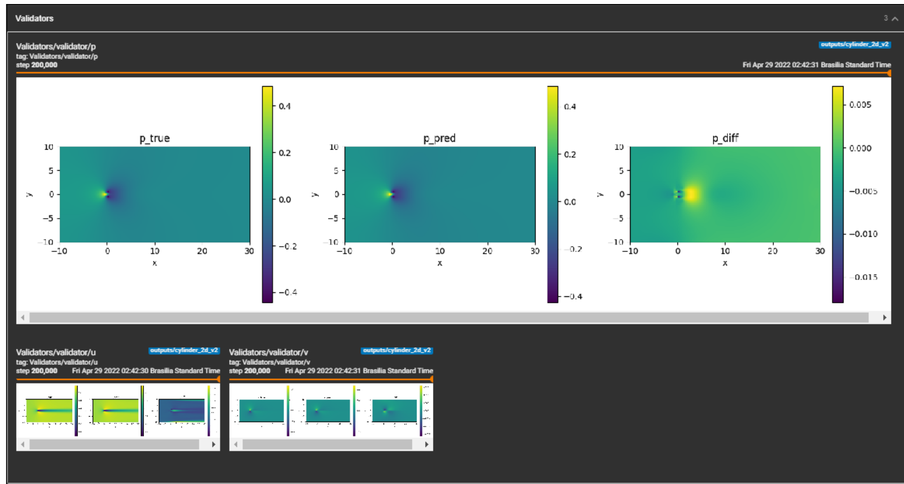


Figura 3.20: Mapas de contorno referentes ao objeto de validação via *TensorBoard*.

ção da rede treinada, como mostra a Figura 3.21. Por fim, a última aba, chamada *Time Series* apenas mostra um compilado geral de todos os gráficos que contém alguma evolução temporal, como os já apresentados das abas *Scalars* e *Images*.

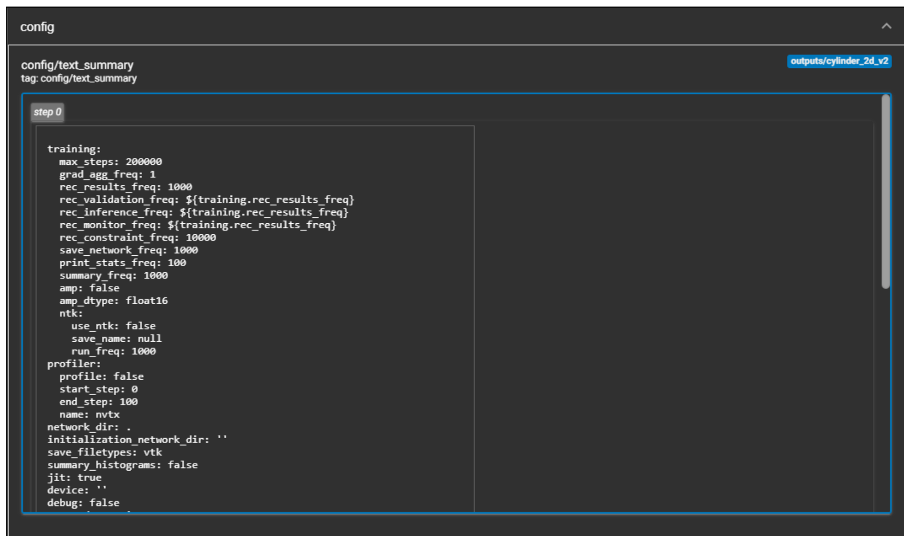


Figura 3.21: Configuração da rede via *TensorBoard*.

3.1.5 Fluxograma para definição de Experimento *Modulus* e geração de *PINNs*

A fim de facilitar o entendimento junto à definição de um experimento *Modulus*, propõe-se um fluxograma que sistematize a declaração dos diferentes objetos necessários à confecção de uma ou mais redes *PINNs*, ilustrando melhor cada etapa necessária a tal objetivo. A Figura 3.22 mostra o fluxograma de definição de um experimento no *Modulus* e geração de *PINNs*, onde as caixas retangulares referem-se aos processos de definição dos objetos mencionados anteriormente, o losango refere-

se a um processo de tomada de decisão, e o símbolo representando o *"Registro de Modelos e Resultados em diretórios"* é relativo à estrutura de dados de resultados geradas dentro do experimento.

A primeira etapa passa pela definição das equações diferenciais do problema a serem analisadas e da estrutura de rede a ser treinada. Define-se também a geometria do problema. Com essas duas primeiras etapas concluídas, devem ser então definidas as restrições físicas e de contorno que deverão ser respeitadas pela *PINN* a ser treinada.

É possível definir também alguns objetos adicionais, como de validação, monitoramento e inferência, os quais podem servir de suporte ao treinamento e geração da rede, mas são totalmente opcionais. Além disso, é possível definir múltiplos objetos de validação, monitoramento ou inferência. Em seguida, define-se o domínio do problema, com a adição das restrições e dos eventuais objetos adicionais.

Por fim, define-se o *Solver* a ser treinado, utilizando a configuração de rede importada via arquivo de configuração e também o domínio anteriormente definido. O treinamento da *PINN* gera então os dados aqui explorados, como o próprio *solver* (representado pela rede neural), os dados de configuração, de treinamento (referentes às restrições definidas) e de eventuais objetos adicionais. Esses dados podem ser analisados individualmente via *Paraview*, ou podem ser analisados através da ferramenta *TensorBoard*.

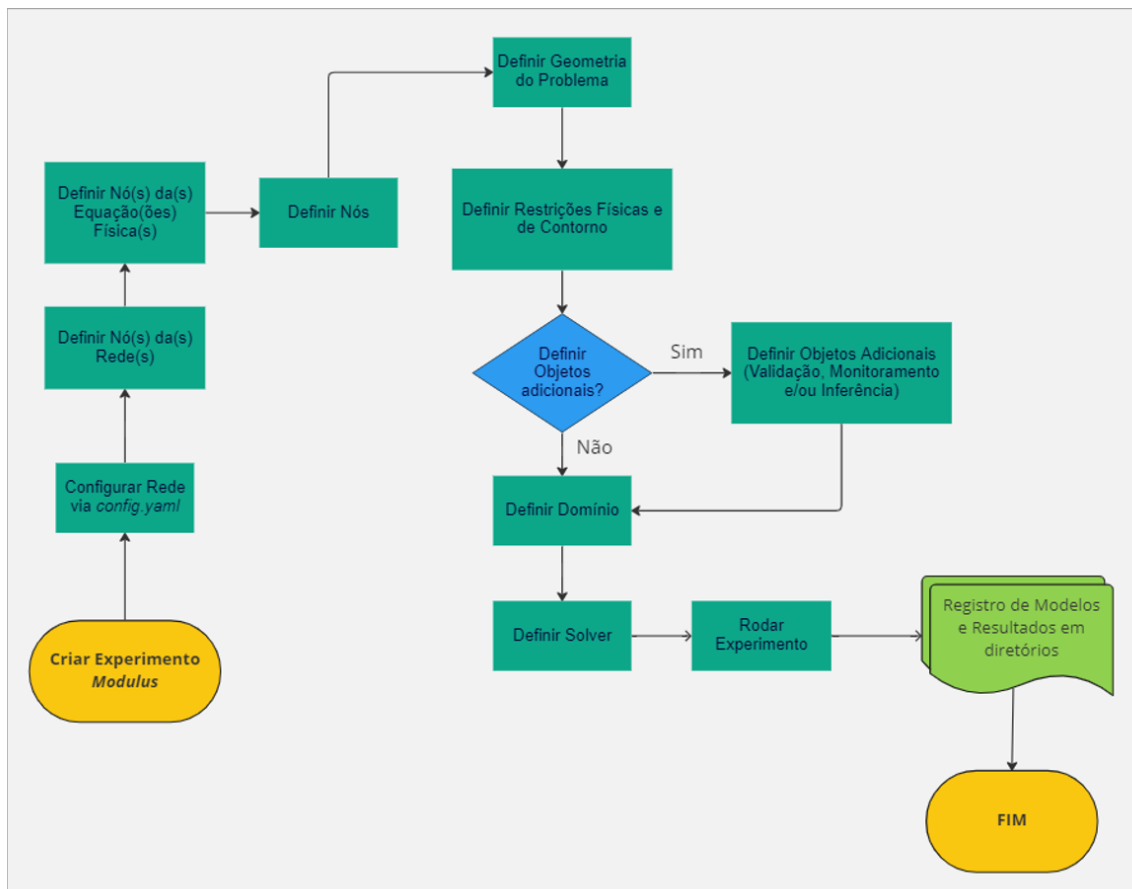


Figura 3.22: Fluxograma de definição de um experimento *Modulus* e geração de *PINNs*.

Capítulo 4

Análise de escolha de modelos em ***PINNs*** com ***Modulus***

Esse capítulo tem como objetivo apresentar uma análise mais criteriosa em relação à geração, comparação e escolha de modelos em *PINNs* dentro do *framework Modulus*, de maneira que sejam mapeadas eventuais limitações no que compete à análise de resultados de múltiplos modelos produzidos. O ambiente computacional de alto desempenho do supercomputador Santos Dumont foi utilizado para a execução do experimento analisado e seus resultados foram analisados em ambiente *Desktop*.

A Seção 4.1 apresenta e detalha o desenvolvimento, execução e análise de resultados de modelos a partir de um novo experimento examinado, ao passo que a Seção 4.2 mapeia as limitações relativas à analise de escolha de modelos *PINNs* com o *framework Modulus*.

4.1 Geração e Análise de modelos com ***Modulus***

A presente seção aplica os conhecimentos adquiridos durante a fase de exploração do *framework Modulus* a partir do experimento preliminar explorado na Seção 3.1, de maneira a realizar um novo experimento de geração de modelos a partir de um outro exemplo disponível no banco de exemplos do *Modulus*. Com isso, examina-se o uso das funcionalidades disponíveis tanto no próprio *Modulus* como no *Tensorboard* com o objetivo de reestruturar os dados computados no treinamento de diversos modelos a partir de uma busca de hiperparâmetros via modo *multirun*, produzindo-se como resultado uma análise de dados comparativa entre os diferentes modelos mais robusta e capaz de extrair mais conhecimento para o analista que produziu as diferentes *PINNs*.

4.1.1 Definição do Experimento Explorado de Equação de Onda em meio 1D

O novo exemplo explorado é o referente ao problema da equação de onda em um meio 1D, e pode ser encontrado na pasta *examples/wave_ equation*. Os códigos em *python* aqui analisados são os dos arquivos *wave_ 1d.py* e *wave_ equation.py*, sendo o primeiro referente à definição do modelo *PINN* em si, e o último relativo à definição da equação física a ser acoplada no treinamento do modelo.

No arquivo *wave_ 1d.py*, são definidos todos os nós e objetos necessários para o devido treinamento do modelo, um objeto de validação (uma vez que a solução de tal problema já é conhecida analiticamente e pode ser gerada), bem como uma instância da equação física a ser acoplada ao treinamento da *PINN*, relacionada justamente à própria equação de onda. Tal equação física é definida no arquivo *wave_ equation.py*

Nesse caso, a equação é definida apenas como ilustrativo de como podem ser definidas equações físicas que não necessariamente estejam disponíveis no *Modulus*, uma vez que já é disponibilizada no próprio *framework* uma equação mais genérica e universal para a equação de onda. A Figura 4.1 apresenta o código da equação definida e utilizada no exemplo.

Mais detalhes desse exemplo em específico podem ser vistos em `docs.nvidia.com/deeplearning/modulus/text/foundational/1d_wave_equation.html`, onde são apresentadas as particularidades do problema físico em si, bem como dos diferentes objetos definidos.

4.1.2 Execução do Experimento

Essa seção trata da execução do experimento analisado, e ela é dividida em 3 partes principais:

- Reestruturação do Código;
- Geração dos Modelos;
- Análise de Resultados;

4.1.2.1 Reestruturação do Código

Primeiramente, houve uma reestruturação do código do exemplo da equação de onda, de maneira a serem modularizados trechos de código referentes aos principais blocos definidos no *Workflow* de definição de uma *PINN*, confeccionado no Experimento 1.

No *Workflow*, como mostra a Figura 3.22, é possível notar diferentes blocos responsáveis pela definição e geração de um modelo via *framework Modulus*. Desses diferentes blocos, foram então modularizadas as partes relacionadas à definição da

```python
from sympy import Symbol, Function, Number
from modulus.pdes import PDES


class WaveEquation1D(PDES):
    """..."""

    name = "WaveEquation1D"

    def __init__(self, c=1.0):
        # coordinates
        x = Symbol("x")

        # time
        t = Symbol("t")

        # make input variables
        input_variables = {"x": x, "t": t}

        # make u function
        u = Function("u")(*input_variables)

        # wave speed coefficient
        if type(c) is str:
            c = Function(c)(*input_variables)
        elif type(c) in [float, int]:
            c = Number(c)

        # set equations
        self.equations = {}
        self.equations["wave_equation"] = u.diff(t, 2) - (c ** 2 * u.diff(x)).diff(x)
```

Figura 4.1: Código da definição da Equação de Onda 1D.

```python
from nodes import make_nodes
from constraints import define_contraints
from validation import define_validator


@modulus.main(config_path="conf", config_name="config")
def run(cfg: ModulusConfig) -> None:
    # make list of nodes to unroll graph on
    nodes = make_nodes(cfg=cfg)

    # make domain
    domain = Domain()

    # add constraints to solver
    IC, BC, interior = define_contraints(cfg=cfg, nodes=nodes)

    domain.add_constraint(IC, "IC")
    domain.add_constraint(BC, "BC")
    domain.add_constraint(interior, "interior")

    # add validation data
    validator = define_validator(cfg=cfg, nodes=nodes)
    domain.add_validator(validator)

    # make solver
    slv = Solver(cfg, domain)

    # start solver
    slv.solve()


if __name__ == "__main__":
    run()
```

Figura 4.2: Código principal do experimento 2 depois da reestruturação.

geometria do problema (arquivo *geometry.py*), dos nós da rede e da equação física (arquivo *nodes.py*), das restrições físicas e de contorno (arquivo *contraints.py*) e dos objetos adicionais (nesse caso, apenas um objeto de validação em um arquivo *validation.py*). Todos os módulos foram definidos em diferentes arquivos *python* no mesmo diretório do exemplo.

A definição do domínio e do *solver* foram feitas no código principal (arquivo *wave_1d.py*), que foi o responsável por invocar os módulos definidos em arquivos *python* separados. A Figura 4.2 mostra o código principal depois de sua reestruturação.

Com a devida modularização e reestruturação do código, executa-se o experimento de maneira a gerar modelos com diferentes conjuntos de hiperparâmetros para o presente exemplo.

4.1.2.2 Geração dos Modelos

O exemplo foi executado dentro do experimento com uma funcionalidade presente no pacote *Hydra*, que por consequência foi expandida para o *framework Modulus*, que permite a execução de múltiplas tarefas com diferentes configurações, chamada *multi-run*. Mais detalhes sobre tal funcionalidade podem ser encontrados em `hydra.cc/docs/tutorials/basic/running_your_app/multi-run/`. Tal funcionalidade é

```
$ python3 wave_1d.py -m arch.fully_connected.layer_size=128,256,512
arch.fully_connected.nr_layers=2,4,6 optimizer=sgd,adam,rmsprop
```

Figura 4.3: Comando utilizado para execução do experimento *wave_1d.py*

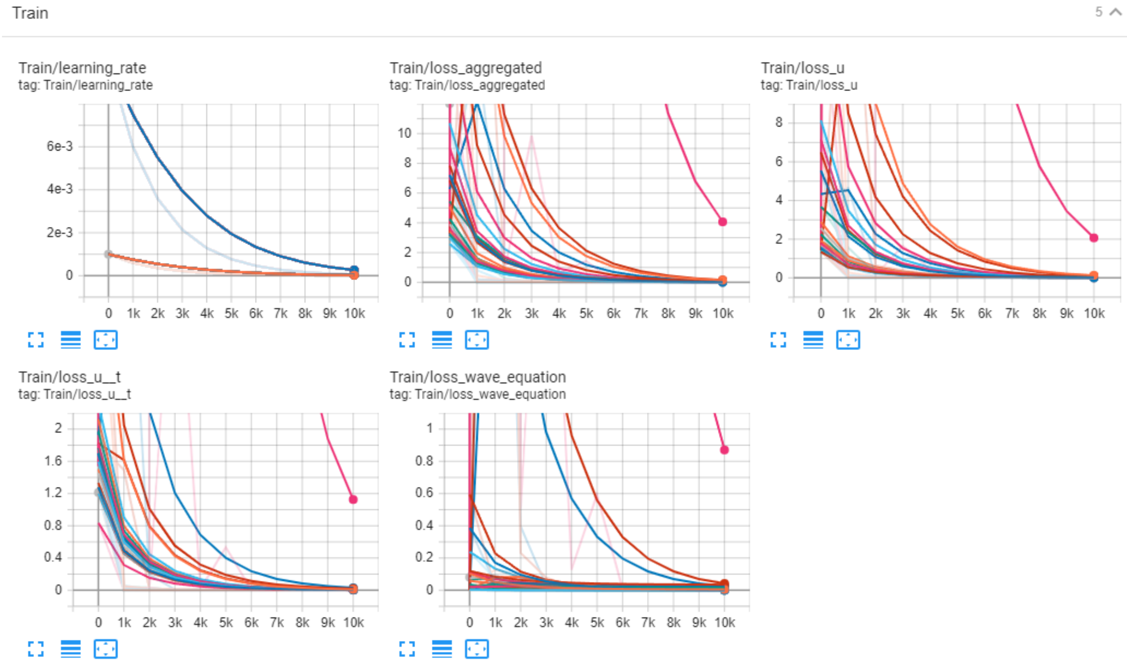


Figura 4.4: Evolução temporal dos erros de treinamento para os modelos gerados.

utilizada de modo a ser feita uma busca de hiperparâmetros na arquitetura definida.

Desse modo, são variados o valor do número de camadas (dois, quatro e seis), o valor do número de neurônios de cada camada (128, 256 e 512) e o otimizador utilizado (*adam*, *sgd* e *rmsprop*). Com essa rede de hiperparâmetros a serem variados, são gerados então 27 modelos, de acordo com um conjunto referente a cada combinação. O comando utilizado na *CLI* (*Command Line Interface*) é dado na Figura 4.3.

4.1.2.3 Análise de Resultados

No presente experimento, o *Tensorboard* foi novamente utilizado para fins de comparação dos diferentes modelos gerados. A Figura 4.4 mostra a evolução do *learning rate* e dos erros das diferentes variáveos de saída definidas nas restrições do modelo.

Os diferentes modelos treinados foram salvos em uma pasta chamada *multirun*, por terem sido gerados pelo modo *multirun* do pacote *Hydra*. É possível perceber, com a Figura 4.4, que a análise e comparação das diferentes arquiteturas dificulta-se consideravelmente quando feita de maneira somente visual. A quantidade de curvas, apesar de mostrarem a tendência geral dos erros, não respondem, de maneira mais imediata, perguntas relevantes quanto ao desempenho comparativo entre cada

index	run	tag	step	value	wall_time
0	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	0	0.0009994872380048037	1652372326.1148584
1	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	1000	0.0005984299350529909	1652372389.865612
2	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	2000	0.00035830208798870444	1652372454.7119982
3	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	3000	0.00021452869987115264	1652372519.246992
4	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	4000	0.00012844624870922416	1652372584.3691592
5	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	5000	7.690551865380257e-05	1652372649.8179238
6	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	6000	4.604617424774915e-05	1652372714.2123625
7	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	7000	2.7569545636652038e-05	1652372779.6483622
8	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	8000	1.650690501264762e-05	1652372843.7861035
9	arch.fully_connected.layer_size=128,arch.fully_connected.nr_layers=2,optimizer=adam	Train/learning_rate	9000	9.88329429674l951e-06	1652372909.6156545

Figura 4.5: Formato do *DataFrame* extraído para análise.

arquitetura.

Desse modo, faz-se necessária uma etapa de pós-processamento dos dados gerados de maneira que seja possível analisar comparativamente cada uma das arquiteturas geradas, para cada conjunto de hiperparâmetros definido. Para isso, é utilizada uma funcionalidade (ainda em fase experimental) do *Tensorboard*, que permite exportar os dados gerados em um formato *pandas Dataframe*, amplamente utilizado na linguagem *python*. Com isso, é possível fazer uma análise *post-hoc* mais detalhada dos dados. Mais detalhes da funcionalidade aqui mencionada podem ser vistos em `www.tensorflow.org/tensorboard/dataframe_api`

Para a extração dos dados em formato *pandas Dataframe*, primeiramente o experimento teve de ser carregado na plataforma *TensorBoard.dev*, que basicamente tem a função de servir como *host* de experimentos de *Machine Learning* compatíveis com *Tensorboard*. O experimento aqui descrito pode ser acessado a partir da url `www.tensorboard.dev/experiment/HkOSXbqnTyqC6U5AekNn4A`. Deve-se notar que, uma vez que o experimento é carregado na plataforma, os dados são tornados públicos e podem ser acessados por qualquer pessoa com o *link* de acesso.

Já com o experimento devidamente carregado, é possível então, via *python*, acessar os dados dos erros bem como do objeto de validação definido. Depois de extraído, o *DataFrame* apresenta um formato semelhante ao mostrado na Figura 4.5, com as colunas *run* (referente a cada um dos modelos gerados com os diferentes valores de hiperparâmetros), *tag* (relacionada ao nome do erro ou métrica computada), *step* (referente ao passo computado), *value* (relativo ao valor da *tag* computada) e *wall_time* (referente apenas às *timestamps* de cada passo). É possível simplificar o formato do *DataFrame*, de maneira a criar novas colunas para cada *tag* existente. Após fazer isso, o *DataFrame* é reorganizado de acordo com a Figura 4.6 tornando-o próprio para análises mais detalhadas.

Após a reestruturação do *Dataframe*, foi possível responder a algumas perguntas, como por exemplo:

1. qual o tempo de treinamento de cada passo, para cada uma das arquiteturas com 6 camadas e 512 neurônios?

2. qual a melhor arquitetura para cada tipo de erro de treinamento computada

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 287 entries, 0 to 1666
Data columns (total 9 columns):
 #   Column                                  Non-Null Count  Dtype
---  ------                                  --------------  -----
 0   run                                     287 non-null    object
 1   step                                    287 non-null    int64
 2   wall_time                               287 non-null    float64
 3   Train/learning_rate                     287 non-null    float64
 4   Train/loss_aggregated                   287 non-null    float64
 5   Train/loss_u                            287 non-null    float64
 6   Train/loss_u__t                         287 non-null    float64
 7   Train/loss_wave_equation                287 non-null    float64
 8   Validators/validator/l2_relative_error_u  287 non-null    float64
dtypes: float64(7), int64(1), object(1)
memory usage: 22.4+ KB
```

Figura 4.6: Formato do *DataFrame* reorganizado para análise.

Tabela 4.1: Tempos médios de treinamento de cada passo dos modelos treinados com os diferentes otimizadores.

Otimizador	**Tempo Médio de Treinamento por passo (s)**
Adam	80.24
SGD	80.17
RMSProp	79.96

para o último passo?

3. qual o valor final de *learning rate* para a melhor arquitetura em relação à perda total?

Para a primeira pergunta, gera-se um gráfico referente ao tempo de treinamento para cada passo computado dos modelos gerados com 6 camadas escondidas e com 512 neurônios em cada camada. Dessa maneira, são comparados os tempos médios de treinamento relacionados a cada otimizador para essas arquiteturas. A Figura 4.7 mostra o tempo de treinamento para os modelos treinados com os diferentes otimizadores para as arquiteturas mencionadas enquanto que a Tabela 4.1 mostra os tempos médios de treinamento para cada passo, que mostra que, dentro das arquiteturas consideradas, o otimizador RMSProp foi o que obteve menor tempo médio dentre o conjunto de otimizadores, porém sem apresentar uma grande diferença entre os demais.

Já para a segunda pergunta, verifica-se que a arquitetura que produziu o menor erro para as diferentes *tags* de erro de treinamento foi a referente ao modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios em cada camada. A Tabela 4.2 mostra os valores para cada uma das *tags* de treinamento

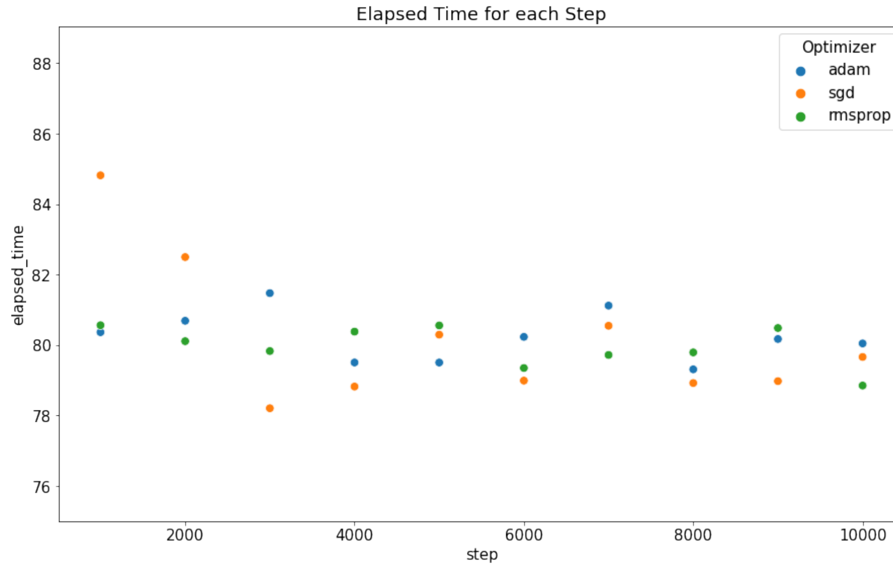


Figura 4.7: Tempo de treinamento de cada passo dos modelos treinados com os diferentes otimizadores.

Tabela 4.2: Erros de treinamento no último passo para modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios.

***Tag* de Treino**	**Valor**
Train/loss_aggregated	1.016E-06
Train/loss_u	4.361E-07
Train/loss_u__t	1.062E-08
Train/loss_wave_equation	5.682E-07

computados para essa arquitetura, ao passo que a Figura 4.8 mostra os gráficos dos erros de treinamento produzidos por esse modelo.

Finalmente, para a terceira pergunta, foi extraído o valor de *learning rate* do modelo referente à arquitetura exposta na Tabela 4.2 a partir de uma simples filtragem no *DataFrame* extraído. A Tabela 4.3 mostra os valores inicial e final de *Learning rate* para o modelo.

Tabela 4.3: Valores de *Learning rate* para modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios.

***Learning rate* inicial**	***Learning rate* final**
1.000E-03	5.918E-06

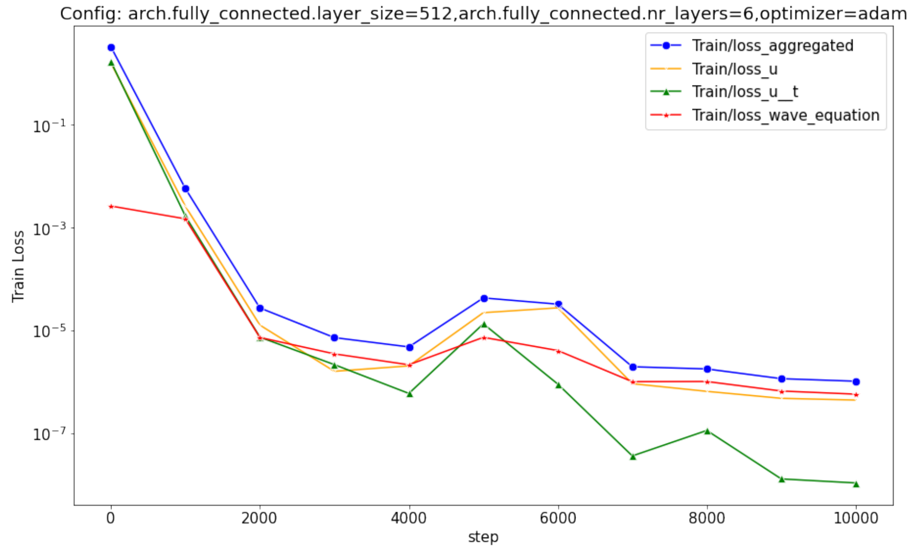


Figura 4.8: Erros de treinamento para modelo treinado com o otimizador *Adam*, com seis camadas escondidas e 512 neurônios..

4.2 Limitações de Análise de Modelos

A condução dos experimentos previamente desenvolvidos produziram resultados diversos que possibilitaram a compreensão das principais estruturas junto à geração, treinamento e análise de modelos informados à física produzidos dentro do *framework Modulus*. Constatou-se a grande flexibilidade a partir da qual experimentos e modelos que descrevam um sistema físico complexo podem ser devidamente estruturados a partir de um fluxograma como o da Figura 3.22 de maneira organizada em um código *python*, bem como também foram destacadas funcionalidades que permitem a geração de diversos modelos com diferentes conjuntos de hiperparâmetros, a partir da extensão do modo *multirun Hydra*.

Porém, também é possível mapear algumas limitações identificadas a partir da execução tanto do experimento preliminar (Seção 3.1) quanto do segundo experimento (seção 4.1), principalmente no que tange a análise de dados. São elas:

- Apesar de a análise visual pelo TensorBoard auxiliar o usuário em vários sentidos, ainda há uma carência de funcionalidades que consigam comparar quantitativamente os modelos gerados.

- A API disponibilizada pelo pacote *TensorBoard* que extrai os dados dos escalares em um *pandas dataframe* tem o empecilho de ser uma funcionalidade ainda experimental. Além disso, há também a necessidade de que haja o *upload* do experimento na plataforma *TensorBoard.dev*, expondo os dados publicamente, o que nem sempre é aceitável em um contexto de pesquisa e desenvolvimento.

Essas limitações foram categorizadas principalmente a partir da necessidade de

análise dos diversos modelos gerados no experimento 2 (seção 4.1). Houve dificuldade de serem agregados os resultados em um formato onde fosse facilitada a análise comparativa dos diferentes modelos gerados. Foi necessária a utilização de uma API experimental do *Tensorboard*, cujo maior problema encontra-se no fato de que é necessário publicizar os resultados gerados a partir do *upload* do experimento na plataforma *TensorBoard.dev*. Além disso, por se tratar de uma API ainda em fase experimental, foram encontrados alguns empecilhos no uso de tal ferramenta, no que tange a extração do *pandas dataframe* em formato expandido para a análise dos resultados. Seria ideal o uso de uma ferramenta já integrada ao *Modulus* e que pudesse fazer tal extração de modo facilitado e rápido, sem a necessidade de *upload* de resultados em uma plataforma separada, bem como que também pudesse aproveitar características inerentes de organização dos resultados gerados via *Modulus* para facilitar uma posterior agregação de dados, de maneira personalizada pelo analista.

Capítulo 5

Modulus Aggregator - Agregador de dados de Experimentos *Modulus*

Esse capítulo tem como objetivo apresentar a ferramenta desenvolvida nessa dissertação, cujo principal objetivo é fazer uma agregação de dados pertinentes produzidos junto ao treinamento de modelos *PINNs* gerados pelo *Modulus*, tendo em vista limitações mapeadas nas funcionalidades de análise de dados oferecidas pelo *framework Modulus*.

Desse modo, são detalhados os objetivos do agregador de dados desenvolvido bem como as vantagens da utilização de tal ferramenta em um ambiente integrado ao *framework Modulus* no sentido de dar suporte à etapa de pós-processamento e análise e comparação de modelos. Também são apontadas algumas limitações da própria ferramenta, de maneira que sejam estabelecidos pontos para avanço e futuros trabalhos.

As seções a seguir descrevem objetivo do desenvolvimento da ferramenta a partir dos experimentos desenvolvidos (Seção 5.1), a arquitetura de *software* da ferramenta e sua integração com o *Modulus* (Seção 5.2), metodologia de uso (Seção 5.3) e, por fim, limitações do pacote (Seção 5.4).

5.1 Objetivo do desenvolvimento da ferramenta

A partir das limitações das funcionalidades presentes relativas ao processo de processamento e análise de dados de resultados dos experimentos gerados a partir de um experimento *Modulus*, reconhece-se a necessidade do desenvolvimento de uma ferramenta que consiga operar de maneira integrada ao *framework Modulus* e que ofereça ao usuário analista funcionalidades que facilitem o processo de comparação de múltiplos modelos gerados em um experimento.

Desse modo, a presente dissertação tem como principal objetivo contribuir com a

disponibilização de uma ferramenta desse gênero, que proveja funcionalidades capazes de atenuar as dificuldades identificadas junto ao processo de análise de resultados de modelos gerados no *Modulus*. O desenvolvimento de uma ferramenta já integrada ao *framework* e que pudesse fazer a extração de dados relativos à múltiplos modelos de modo facilitado e rápido, funcionando como um agregador de dados, sem a necessidade de *upload* dos resultados em uma plataforma separada, bem como que também pudesse aproveitar características inerentes de organização dos resultados gerados via *Modulus*, contribuindo consideravelmente para facilitar o processo de agregação e de análise de dados, de maneira personalizada pelo analista.

A ferramenta nada mais é do que um pacote escrito em *python*, que pode ser instalado em um ambiente de desenvolvimento *Modulus*, de agregação e exportação dos dados relativos a diferentes modelos gerados dentro de um repositório de experimentos. A esse pacote foi dado o nome de *Modulus Aggregator*, e seu código fonte está inteiramente disponível em `https://github.com/mthlimao/modulus_aggregator`.

5.2 Arquitetura de *Software* do *Modulus Aggregator*

A ferramenta desenvolvida tem sua arquitetura de *software* dividida em quatro módulos principais, responsáveis pelas seguintes tarefas:

- Captura dos resultados dos múltiplos modelos treinados
- Pós-processamento dos resultados
- Agregação dos resultados
- Extração dos dados dos resultados em formato que facilite a análise do especialista

A Figura 5.1 ilustra a arquitetura de *software* do *Modulus Aggregator*, bem como sua relação com o *framework Modulus* e sua estrutura de diretórios de múltiplos modelos gerada após a execução de um experimento.

Após a execução de um experimento, especialmente quando são gerados múltiplos modelos *PINNs*, o próprio *framework Modulus* organiza os diversos arquivos relacionados aos diferentes modelos treinados em uma estrutura de diretórios característica por si só. Como foi acompanhado na Seção 4.1.2, os modelos foram salvos em um diretório de nome *multirun*, pelo experimento ter sido executado nesse tipo de modo. Porém, por padrão, os modelos gerados em um experimento são salvos em uma pasta chamada *outputs*, dentro do diretório de trabalho do experimento.

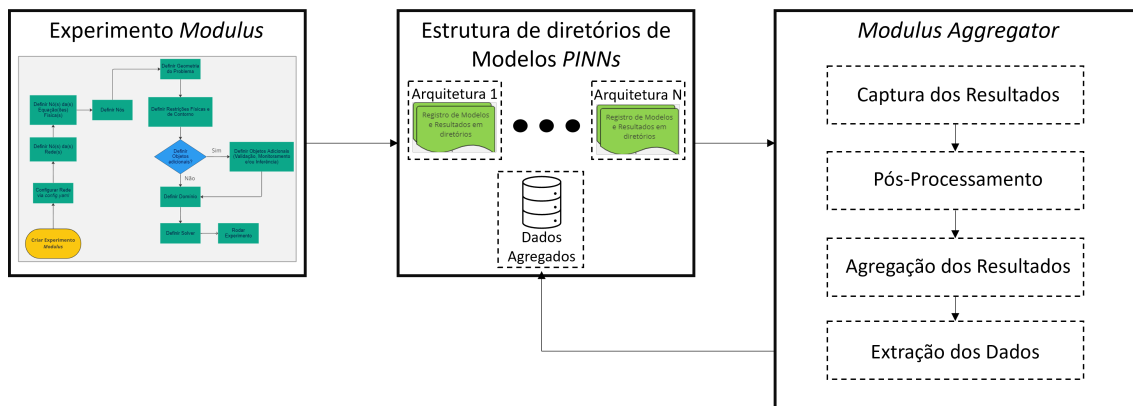


Figura 5.1: Arquitetura do *Modulus Aggregator* e sua integração com o *Modulus*.

A partir dessa estrutura de diretórios, os modelos são salvos em pastas individuais, onde são armazenados todos os arquivos referentes àquele modelo em particular.

Com isso, o *Modulus Aggregator* a partir de seu módulo de captura de resultados, faz o acesso à tal estrutura de diretórios onde os modelos estão armazenados, e realiza a captura dos resultados de todos os diferentes modelos. Em seguida, o módulo de pós-processamento da ferramenta é invocado, onde ocorre a organização dos diferentes tipos de resultados. Após isso, é feita a agregação dos resultados referentes aos diferentes modelos em uma estrutura que possa ser devidamente exportada. Finalmente, ocorre então a extração dos dados referentes ao resultados, de maneira que os dados já agregados são armazenados dentro da estrutura de diretórios gerada pelo próprio *Modulus*.

A estrutura na qual os dados são salvos facilita o processo de análise e comparação de modelos pelo especialista, aproveitando a própria estrutura de diretórios criada pelo *framework*. Além disso, o processo de extração de dados pode ser realizado de maneira customizada pelo especialista, de maneira que é possível extrair os dados referentes a resultados apenas de treinamento e/ou de monitoramento e/ou de validação.

É importante destacar que a ferramenta desenvolvida tem o objetivo de facilitar a análise e comparação de resultados de múltiplos modelos após o treinamento e de servir como uma ferramenta adicional junto a este processo. Em um cenário de treinamento de apenas um modelo, que não é muito comum em um contexto de ajuste e otimização de hiperparâmetros, a análise de dados pode ser conduzida via *TensorBoard*, que já integra o *framework*. Além disso, não há a captura de dados de proveniência durante o processo de treinamento dos modelos *PINNs* definidos em um experimento *Modulus*

```
(base) λ modagg
Usage: modagg [OPTIONS] COMMAND [ARGS]...

  Main command of modulus_aggregator package.
  This package aims to be a data aggregator for Modulus experiments.

  It's CLI package which allows the user to export data saved in tags (currently, just tensors)
  and save it in a python friendly format (.csv).

  This data aggregation procedure makes it possible for the specialist to conduct deeper analysis,
  specially in a scenario with multiple trained models.

Options:
  --help  Show this message and exit.

Commands:
  about   General Package Info and Contacts.
  export  Group of commands 'export'.
  show    This command basically shows to the user all registered tags in...
```

Figura 5.2: Comandos disponíveis no *Modulus Aggregator*.

5.3 Metodologia de Uso

A utilização do *Modulus Aggregator* se dá de maneira bastante facilitada, sem necessidade de intervenção direta no código dos experimentos definidos no *Modulus*. Seu uso se dá a partir de chamadas de comandos via interface de linha de comando, ou *Command Line Interface* (CLI). Desse modo, para que o especialista possa fazer o devido uso da ferramenta, é necessário fazer sua instalação, preferencialmente no mesmo ambiente de desenvolvimento onde o *Modulus* está instalado. A Figura 5.2 mostra os comandos atualmente disponíveis, depois de feita sua instalação.

É possível consultar maiores detalhes sobre o processo de instalação, uso dos comando disponíveis e também sobre os diferentes formatos de saída em que são extraídos os dados dos resultados de múltiplos modelos no repositório do *Modulus Aggregator* disponibilizado no *Github*.

O conjunto de comandos disponibilizados pelo *Modulus Aggregator* permite ao usuário analista exportar os resultados referentes a cada modelo gerado em um experimento em arquivos de saída *.csv*, de formato propício para carregamento em um *pandas dataframe* em *python*. Os arquivos de saída podem ser exportados em formato padrão ou em *wide-form*, de acordo com o comando, de maneira que os *dataframes* carregados em *python* tenham um formato semelhante ao exibido nas Figuras 4.5 e 4.6.

Dessa maneira, o fluxo para a devida utilização da ferramenta está também ilustrado na Figura 5.1, onde é necessário o especialista primeiramente defina e execute o experimento *Modulus* de acordo com os detalhes estudados na Seção 3.1. Após a devida execução do experimento e geração dos modelos treinados na estrutura de diretórios padronizada pelo *framework*, executa-se as rotinas de extração de dados via *Modulus Aggregator*, quando então os resultados são agregados e exportados dentro do diretório de referência onde estão armazenados os modelos.

Vale ressaltar que processo de carregamento dos resultados agregados (via *pandas Dataframe*, por exemplo) de maneira a realizar a posterior de análise de modelos

e eventual escolha de conjunto de hiperparâmetros a partir de alguma métrica que julgada razoável para o experimento conduzido fica a cargo do especialista.

5.4 Justificativas para abordagem de desenvolvimento do *Modulus Aggregator*

Como anteriormente mencionado, a ferramenta aqui apresentada não constitui uma solução de gerenciamento de dados de proveniência a exemplo das apresentadas em PINA *et al.* (2021b) e PINA (2020), uma vez que não executa a captura desses dados durante o treinamento dos modelos, tampouco os armazena em uma base do tipo SGBD.

Uma das principais razões pelas quais uma solução desse gênero não foi desenvolvida decorre, por exemplo, da necessidade de efetuação de uma série de modificações dentro do código fonte do *framework Modulus*. Tais modificações poderiam caracterizar-se demasiadamente intrusivas para que um especialista pudesse desenvolver seus experimentos ao mesmo tempo em que pudesse instrumentar o código para a devida captura dos dados de proveniência, eventualmente tirando o foco principal daquele que está interessado no treinamento de modelos *PINNs*. Ao mesmo tempo, mesmo que fosse lançada uma extensão do pacote *Modulus* que oferecesse funcionalidades de captura de dados de proveniência, tal pacote correria o risco de tornar-se rapidamente desatualizado a medida em que novas versões do *framework* fossem disponibilizadas.

Além disso, a partir da exploração dos diversos recursos do *framework* que foi feita principalmente na Seções 3.1 e 4.1.2 para a identificação de apoio adicional para análise de resultados, a necessidade da utilização de um SGBD em experimentos definidos via *Modulus* poderia causar dificuldades à definição e ao *deployment* dos modelos para o especialista.

Desse modo, opta-se para o desenvolvimento de uma solução simples e pouco intrusiva que consiga servir como uma ferramenta adicional junto a capacidade de análise de resultados do especialista, de maneira que seja possibilitada a exploração e comparação da performance dos diferentes modelos produzidos, de modo a aproveitar a estrutura nativa de um repositório de experimentos no *Modulus*. Uma vez validada e consolidada a solução aqui proposta, a ideia é que futuramente sejam também pensadas ferramentas que possam prover funcionalidades de extração de dados de proveniência em tempo real, o que se coloca como um ponto a ser explorado em estudos futuros.

Capítulo 6

Avaliação Experimental

Esse capítulo tem como principal objetivo detalhar o processo de avaliação experimental da ferramenta de agregação de dados de resultados oriundos de *PINNs* treinadas via *framework Modulus*. O presente capítulo ilustra de que modo a utilização da ferramenta pode ajudar o analista de dados a melhor orientar sua análise de parâmetros em um cenário de múltiplos modelos e de considerável volume de dados, principalmente quando o *Modulus Aggregator* é utilizado em conjunto com a ferramenta *Tensorboard*. Além disso, mostra-se também a possibilidade de automatização do processo de extração e filtragem de modelos com a ferramenta desenvolvida, auxiliando o analista no processo de configuração de hiperparâmetros de *PINNs*. O supercomputador Santos Dumont foi utilizado para a execução do experimento aqui analisado e seus resultados foram analisados localmente em ambiente *Desktop*.

A Seção 6.1 analisa o experimento *Modulus* conduzido no presente capítulo, enquanto que a Seção 6.2 detalha a utilização da ferramenta dentro do contexto do experimento *Modulus* produzido. Por fim, a Seção 6.3 expõe a análise comparativa de modelos produzida.

6.1 Experimento Conduzido de Equação de Onda em domínio 2D

O experimento conduzido no presente capítulo refere-se a um exemplo de propagação de ondas sísmicas em um domínio 2D retangular de $2km$ de largura e $2km$ de comprimento, com uma fonte *Ricker*, cuja descrição é devidamente detalhada em `https://docs.nvidia.com/deeplearning/modulus/user_guide/foundational/2d_wave_equation.html`. Nesse exemplo, são definidas tanto as equações de contorno para as extremidades do domínio (a fim simular a propagação de onda em um meio infinito em um domínio finito sem que haja reflexões) quanto a equação de onda para os pontos do interior do domínio.

```
$ python3 wave_2d.py -m arch.fully_connected.layer_size=128,256,512
arch.fully_connected.nr_layers=2,3,4,5,6 optimizer=sgd,adam,rmsprop
optimizer.lr=0.001,0.0003,0.0001,0.00003,0.00001
```

Figura 6.1: Comando utilizado para execução do experimento *wave_ 2d.py*

O objetivo do problema é prever os valores do campo de pressão $u(\mathbf{x}, t)$ (onde $\mathbf{x} = (x, y)$) e também da velocidade da onda no meio c. Para isso, são treinadas duas redes que têm como variáveis de saída os valores de u e c. Além das restrições impostas pelas equações de onda e de contorno definidas, também utiliza-se no exemplo uma série de dados obtidos através do simulador *Devito* para treinamento (para 4 passos de tempo de $100ms$ a $300ms$) e validação (para 13 passos de tempo de $350ms$ a *950ms*) das *PINNs* geradas. Os dados simulados são fornecidos no próprio diretório do exemplo e são dispostos em formato *.npz*.

Nesse experimento, foi empregada a mesma técnica de geração de modelos utilizada no experimento descrito na Seção 4.1.2. No caso, o modo *multirun* foi novamente utilizado para o treinamento dos modelos, com a diferença de que um maior número de parâmetros foi utilizado junto à geração das *PÌNNs*. Além do valor do número de camadas (2, 3, 4, 5 e 6), do tamanho de cada camada (128, 256 e 512) e dos otimizadores utilizados (*adam*, *sgd* e *rmsprop*), foram variados também os valores iniciais de *learning rate* ($1e-05$, $3e-05$, 0.0001, 0.0003 e 0.001). Dessa maneira, foram então gerados 225 modelos, com um espaço de armazenamento de cerca de $8,78$Gb. Por fim, o experimento demorou aproximadamente 120 horas para ser concluído. O comando para a execução do experimento no modo *multirun* é dado na Figura 6.1.

É interessante observar que apesar de o presente experimento não necessariamente refletir um cenário real, onde é gerado uma quantidade por vezes muito maior de dados a depender do experimento, ainda assim gerencia-se aqui um volume razoável de arquivos e modelos, de maneira que torna-se viável a avaliação experimental do agregador de dados em complemento às ferramentas de visualização de resultados apresentadas nas Seções anteriores. Por exemplo, caso o problema de propagação de ondas sísmicas tratado fosse dado em um domínio 3D, como estudado em BARBOSA, o tamanho de armazenamento dos arquivos de saída gerados poderia chegar a uma ordem de grandeza de dezenas de Terabytes.

Dessa maneira, o desafio colocado à ferramenta desenvolvida refere-se tanto a sua capacidade de dar suporte ao analista à medida que seja capaz de reduzir o espaço de análise de resultados (reduzindo por consequência o universo de arquivos a serem analisados) de acordo com métricas de interesse estabelecidas, bem como a sua capacidade de complementar a análise padrão de visualização de dados no *Tensorboard*. Todo o experimento foi executado em um ambiente de alto processamento

no supercomputador *Santos Dumont*. Não foram utilizadas técnicas de paralelização nesse caso.

Porém, para a devida utilização do *Modulus Aggregator* no ambiente computacional aqui citado, foi necessário que a ferramenta fosse instalada na imagem oficial do *Modulus* disponibilizada para *download*. Uma vez instalado o agregador via *pip*, a nova imagem extraída foi então carregada no *Santos Dumont* via *Singularity*, e o experimento foi executado, bem como a extração dos resultados efetuada por um *script* escrito em *python*, o que é detalhado na próxima Seção.

6.2 Utilização do *Modulus Aggregator*

A utilização da ferramenta de agregação de dados foi dada em um contexto de filtragem de modelos a partir de determinadas métricas pré fixadas de interesse a fim de redução da quantidade de dados a ser analisada. Nesse cenário, utiliza-se um simples *script* escrito em *python* com o objetivo de definir tais métricas, fazer a filtragem dos modelos, e por fim, salvar os melhores modelos para análise.

Como o experimento foi executado a partir de um ambiente de alto desempenho computacional do *Santos Dumont*, foi necessário fazer o download dos modelos a fim de que a análise de dados via *Tensorboard* fosse possibilitada. Com essa necessidade colocada, o *script* implementado em *python* foi utilizado de maneira a ser executado logo após a execução do experimento, ou seja, da criação dos modelos.

A primeira parte do *script* foi referente à geração da base de dados de resultados agregadas utilizando o diretório *multirun* gerado após o treinamento dos modelos. Desse modo, foi possível extrair uma base de dados de resultados apta a ser utilizada junto à filtragem de dados. A Figura 6.2 mostra a função referente à primeira parte do *script*, onde há a extração de dados com o agregador e o carregamentos dos resultados em um *pandas dataframe*.

```
def agregar_resultados(models_path):
    cmd = ['modagg', 'export', 'tensors', '-mp', f'{models_path.
        as_posix()}']
    subprocess.run(cmd)

    df = pd.read_csv(models_path / 'multirun_tensors.csv', sep=';')

    return df
```

Figura 6.2: Função referente à primeira parte do *script* utilizado.

A segunda parte do código foi escrita com a finalidade de fixar as métricas a serem analisadas, bem como filtrar aqueles modelos que tiveram melhor desempenho em

cada uma destas. O processo de filtragem foi implementado de maneira a selecionar os modelos que apresentaram menor erro para o último passo de treinamento para cada uma das métricas determinadas. Outros parâmetros de filtragem poderiam ser utilizados (por exemplo, filtrando-se o menor erro médio), porém, o objetivo foi apenas destacar o potencial de automatização de um experimento com o agregador. Para o problema em questão, foram escolhidas as seguintes métricas para fins de filtragem de dados:

- *Train/loss_aggregated*
- *Train/loss_c*
- *Train/loss_open_boundary*
- *Train/loss_u*
- *Train/loss_wave_equation*
- *Validators/VAL_0012/l2_relative_error_u*
- *Validators/Velocity/l2_relative_error_c*

As métricas referentes aos erros de treino são relativas aos erros computados para a função de perda agregada, para a perda em relação à velocidade no meio c, para a perda computada na equação de contorno, para a perda em relação ao campo de pressão u e para a perda computada para a equação de onda (respectivamente *Train/loss_aggregated*, *Train/loss_c*, *Train/loss_open_boundary*, *Train/loss_u* e *Train/loss_wave_equation*). Já as métricas dos erros de validação fixadas correspondem ao erro relativo do campo de pressão u para o último instante de tempo simulado bem como ao erro relativo da velocidade no meio c (respectivamente *Validators/VAL_0012/l2_relative_error_u* e *Validators/Velocity/l2_relative_error_c*). A Figura 6.3 mostra a função referente à segunda parte do *script* implementado, onde há a filtragem dos modelos a partir do menor erro para o último passo de treinamento.

Com os modelos devidamente filtrados, prossegue-se para a terceira e última parte do *script*, onde apenas os dados dos modelos filtrados são copiados para uma nova pasta chamada *multirun_filtered* no repositório do experimento, onde tem então seus resultados novamente extraídos via *Modulus Aggregator*. Com esse diretório de menor volume de armazenamento devidamente criado, foi então feito o *download* de tais dados para uma máquina local, de maneira a efetuar a análise de resultados pelo *Tensorboard*. A Figura , onde os melhores modelos são copiados para a pasta *multirun_filtered* e cujos resultados são extraídos via *Modulus Aggregator*. Já Figura 6.5 ilustra a utilização do *Modulus Aggregator* no experimento.

```
def filtrar_melhores_modelos(df, metrics):
    df_comp_models = df[df['step'] == 40000]
    best_models_metrics = {}

    for metric in metrics:
        df_metric = df_comp_models[df_comp_models['tag'] == metric]
        df_min_val = df_metric[df_metric['value'] == df_metric['value
            '].min()]
        best_models_metrics[metric] = df_min_val['run'].values[0]

    best_models = np.unique([model for model in best_models_metrics.
        values()]).tolist()

    return best_models, best_models_metrics
```

Figura 6.3: Função referente à segunda parte do *script* utilizado.

O processo de filtragem descrito foi capaz de reduzir um espaço de análise de 225 modelos, para um espaço de apenas cinco modelos no total, uma vez que apenas cinco modelos foram selecionados a partir das sete métricas selecionadas, significando uma redução considerável de um total de *8,78*Gb de armazenamento de arquivos dos modelos gerados no diretório *multirun* para um total de apenas *70,5*Mb de armazenamento no diretório *multirun_filtered*, com apenas 206 arquivos (de 9953 arquivos anteriormente) e 20 pastas (de 900 pastas anteriormente). Isso representa uma redução de aproximadamente $99,2\%$ do volume de armazenamento a ser analisado localmente. Esse tipo de utilização do agregador mostra a capacidade da ferramenta de reduzir o espaço de estudo do cientista de dados em um determinado experimento *Modulus*, desde que observada a possibilidade de serem estabelecidas métricas para filtragem de dados *a priori*.

A Figura 6.6 mostra a diferença visual observada no *Tensorboard* em cenários com e sem filtragem de modelos para a evolução temporal do erro de validação relativo em c. É possível notar a dificuldade imposta à análise dos resultados em um cenário de múltiplos modelos, principalmente quando feita exclusivamente no *Tensorboard*. Uma vez feita a filtragem, tal atividade torna-se mais intuitiva, possibilitando o analista a investigar tendências para cada um dos modelos selecionados. A análise de dados para o presente experimento é devidamente detalhada na próxima Seção.

6.3 Análise de Resultados

Um processo de análise de resultados junto aos modelos filtrados foi conduzido a fim de que maiores informações sobre os hiperparâmetros utilizados pudessem ser obtidas com o propósito de direcionar o treinamento das *PINNs*, reduzindo-se a

```
def agregar_resultados_filtrados(models_path, best_models):
    models_filtered_path = models_path.parent / 'multirun_filtered'
    if not models_filtered_path.exists():
        models_filtered_path.mkdir()

    for model in best_models:
        shutil.copytree((models_path / model).as_posix(), (
            models_filtered_path / model).as_posix(), symlinks=True)

    shutil.copy2((models_path / "multirun.yaml").as_posix(), (
        models_filtered_path / "multirun.yaml").as_posix())

    cmd_filtered = ['modagg', 'export', 'tensors', '-mp', f'{
        models_filtered_path.as_posix()}']
    subprocess.run(cmd_filtered)
```

Figura 6.4: Função referente à terceira parte do *script* utilizado.

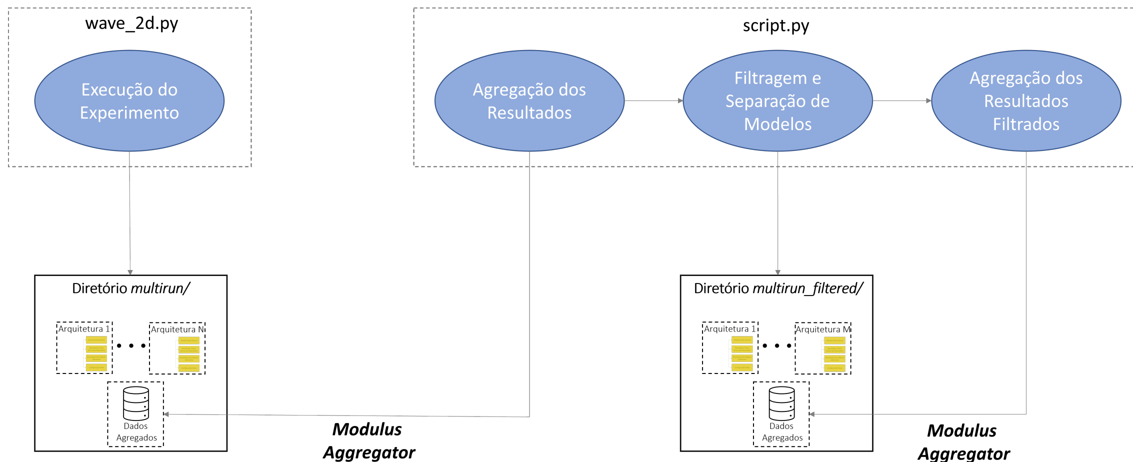
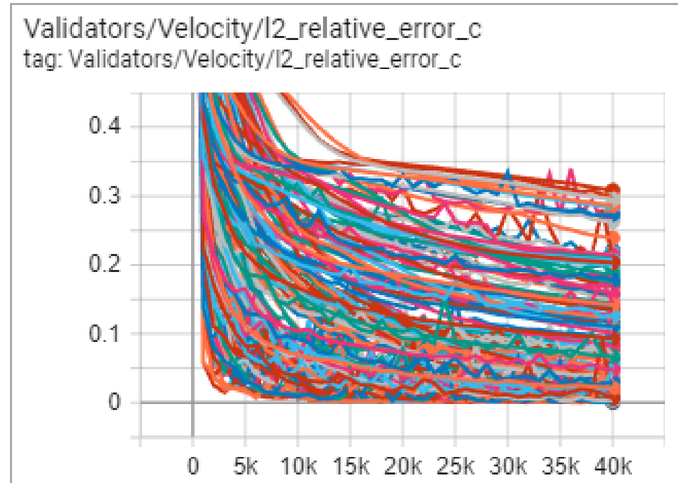


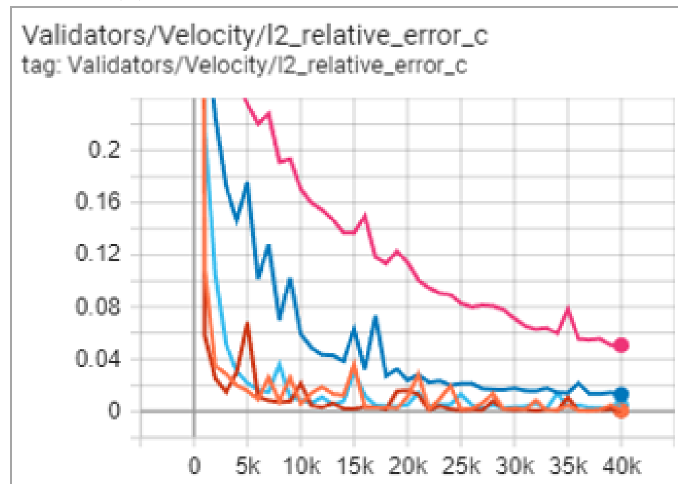
Figura 6.5: Esquema de utilização do *Modulus Aggregator*.

dimensionalidade do problema em questão. Para tal, analisa-se primeiramente a arquitetura bem como os hiperparâmetros dos modelos filtrados. A Tabela 6.1 mostra os modelos, seus hiperparâmetros associados (com os valores de tamanho da camada, número de camadas, *learning rate* inicial e otimizador, respectivamente) bem como as métricas que filtraram cada um desses modelos. Todo o estudo dos resultados foi efetuado a partir das bases de dados produzidas pela ferramenta *Modulus Aggregator*, armazenadas nos diretórios *multirun* e *multirun_filtered*, como ilustrado na Figura 6.5.

Nota-se na Tabela 6.1 que, primeiramente, dois dos cinco modelos selecionados (A e C) conseguiram o melhor desempenho para duas métricas simultaneamente. Além disso, três dos cinco modelos utilizaram o otimizador *adam* no treinamento. A partir disso, é possível fazer uma análise de resultados via *Tensorboard*. A Fi-



(a) Gráfico sem filtragem de modelos.



(b) Gráfico com filtragem de modelos.

Figura 6.6: Gráficos do erro de validação em c no *Tensorboard*.

Tabela 6.1: Melhores modelos filtrados.

	Hiperparâmetros	**Melhor Métrica**
A	(128, 5, 0.001, *adam*)	*Train/loss_aggregated*, *Train/loss_u*
B	(128, 6, 0.0001, *sgd*)	*Train/loss_open_boundary*
C	(256, 6, 0.001, *adam*)	*Train/loss_c*, *Validators/Velocity/l2_relative_error_c*
D	(512, 5, 0.0003, *adam*)	*Validators/VAL_0012/l2_relative_error_u*
E	(512, 5, 0.00003, *sgd*)	*Train/loss_wave_equation*

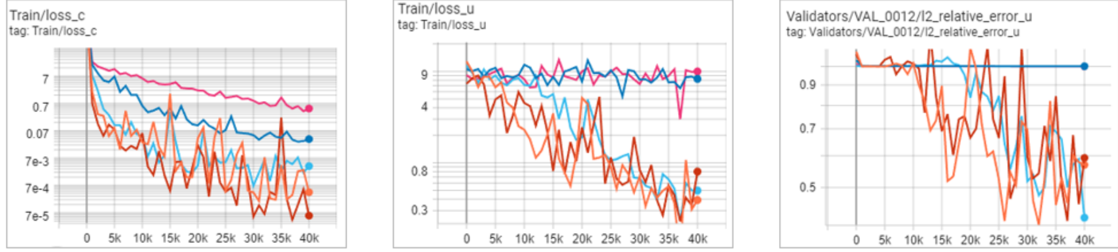


Figura 6.7: Evolução temporal para erros de treinamento e validação dos modelos filtrados.

gura 6.7 mostra a evolução temporal para erros de treinamento (*Train/loss_ c* e *Train/loss_ u*) e validação (*Validators/Velocity/l2_ relative_ error_ c*).

Percebe-se na Figura 6.7, principalmente para os gráficos dos erros de *Train/-loss_ u* e *Validators/Velocity/l2_ relative_ error_ c*, que três dos cinco modelos apresentam melhor desempenho, inclusive tendência de queda nos erros referentes a u. Tal comportamento também se verifica para todas as demais métricas, inclusive as de validação, excetuando-se os erros para *Train/loss_ open_ boundary* e *Train/-loss_ wave_ equation*.

Esses três modelos são justamente aqueles treinados com o otimizador *adam* (modelos A, C e D da Tabela 6.1). É possível notar que esses três modelos apresentam tamanhos (128, 256, 512) e números de camada (5 e 6) diversos. Desse modo, a fim de manter a maior simplicidade possível de um modelo, opta-se pela arquitetura mais simples, nesse caso, a de 5 camadas com 128 neurônios a fim de observar a influência de outros hiperparâmetros nas métricas de validação.

Nesse sentido, foi conduzida uma investigação dos erros de validação relativos a u para o último passo de treinamento. Tais métricas referem-se aos erros de validação medidos para os diferentes instantes de tempo simulados no *software Devito* e utilizados como referência para a averiguação da qualidade das *PINNs*. Tal estudo foi feito com o propósito de avaliar quais valores de *learning rate* poderiam ser utilizados junto ao ajuste de tal hiperparâmetro junto ao treinamento de novos modelos.

A Figura 6.8 mostra a variação dos erros de validação relativos de u para o último passo de treinamento em função dos demais valores de *learning rate*, fixando-se a arquitetura mais simples observada dos melhores modelos filtrados (5 camadas com 128 neurônios) com o otimizador *adam*. O gráfico produzido indica claramente que valores de *learning rate* iguais ou menores que 0.0001 acarretam uma performance pior para os erros de validação em relação à u para tal arquitetura e otimizador filtrados. Isso pode indicar ao analista que valores maiores de *learning rate* sejam empregados junto ao treinamento de novas *PINNs* via *Modulus*.

O mesmo tipo de análise foi feita variando-se os otimizadores e fixando-se a

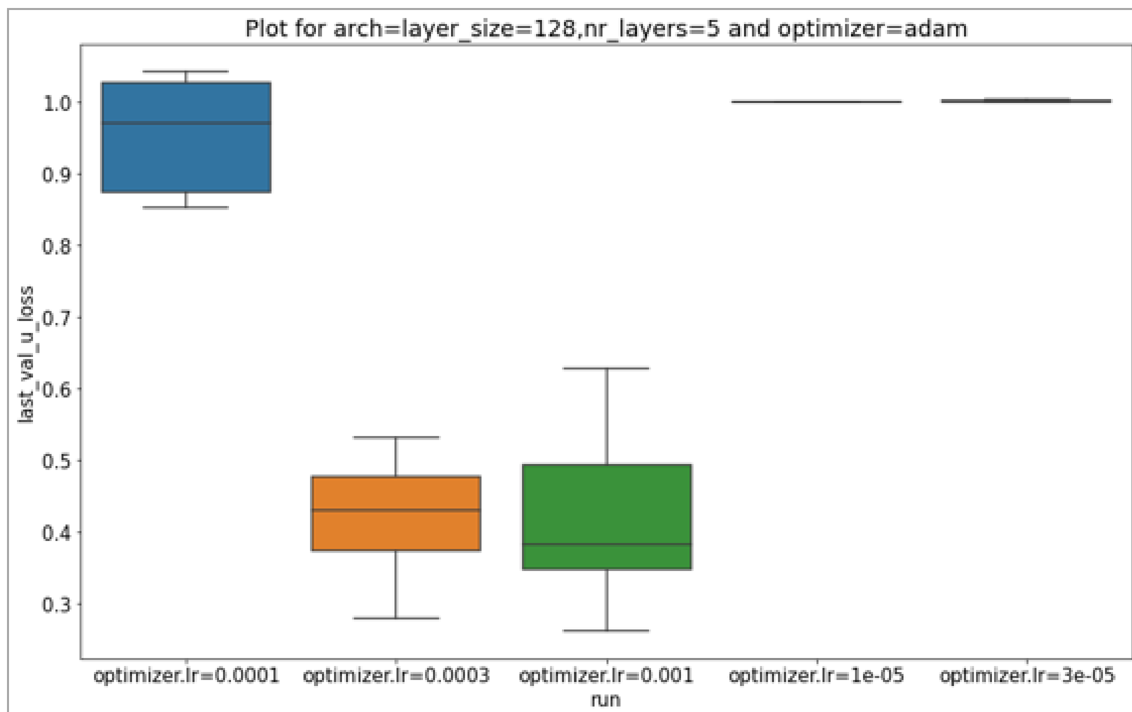


Figura 6.8: Erros de validação de u em função do *learning rate* para a arquitetura de 5 camadas com 128 neurônios e otimizador *adam*.

arquitetura e o valor inicial de *learning rate*. Nesse caso, foram somente analisados os otimizadores *adam* e *rmsprop* pois, para tal subconjunto de hiperparâmetros (5 camadas, 128 neurônios e *learning rate* inicial igual a 0.001), o modelo treinado com o otimizador *sgd* não foi concluído por conta de um erro de convergência.

A Figura 6.9 ilustra o mesmo tipo de gráfico produzido com os erros de validação relativos à u para o último passo de treinamento em função dos diferentes otimizadores utilizados com os modelos gerados, fixada a arquitetura mais simples de rede (5 camadas com 128 neurônios) e valor inicial de *learning rate* igual a 0.001. Também é possível observar de modo claro que, para essa arquitetura, o otimizador *adam* também oferece melhor desempenho. Em um cenário de geração de novos modelos, a opção pela continuidade da utilização do otimizador *adam* apresenta-se como justificada.

Em suma, foi possível verificar como a utilização da ferramenta *Modulus Aggregator* pode ser benéfica em um cenário de análise de resultados, principalmente quando feita complementarmente ao *Tensorboard*. Mostrou-se na presente Seção como tal processo de análise pode ser conduzido de maneira a auxiliar o analista no processo de tomada de decisão relativo à geração de novos modelos via *Modulus*, em um determinado experimento.

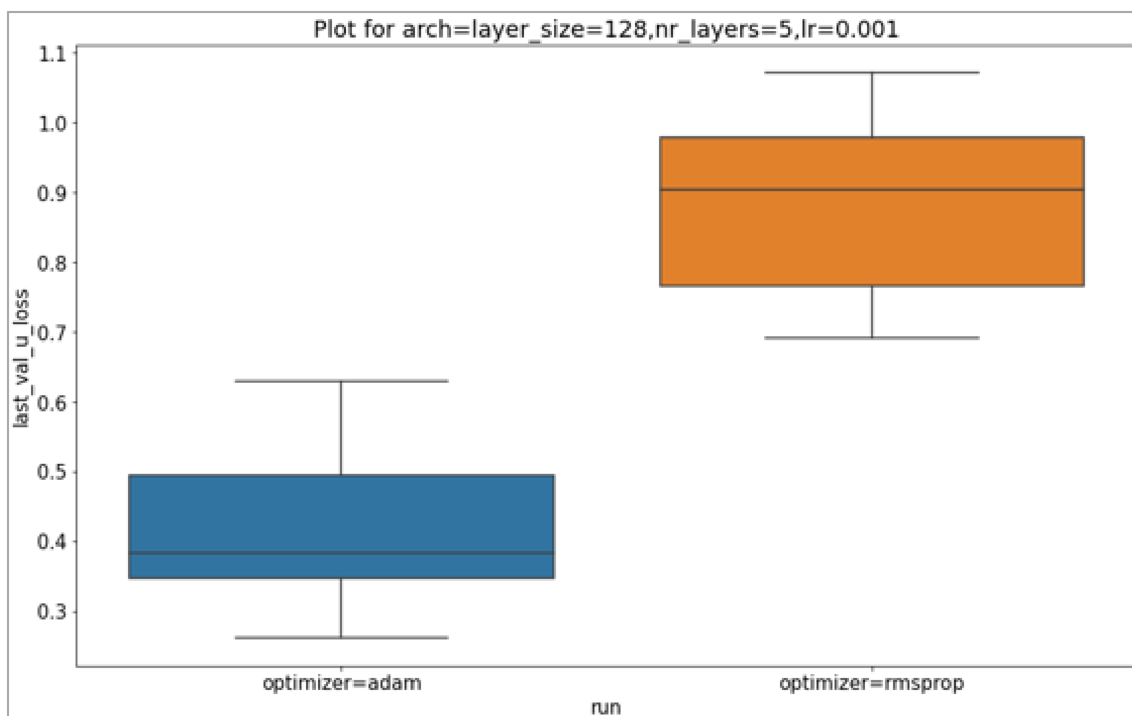


Figura 6.9: Erros de validação de u em função dos diferentes *otimizadores* para a arquitetura de 5 camadas com 128 neurônios e *learning rate* inicial igual a 0.001.

Capítulo 7

Conclusão

A ferramenta de agregação de dados desenvolvida e apresentada nessa dissertação tem como objetivo principal dar suporte no analista de dados ao processo de configuração de hiperparâmetros em um determinado experimento *Modulus*. Em um cenário de desenvolvimento de *PINNs*, uma estratégia de agregação de resultados, especialmente quando são treinados muitos modelos, faz-se necessária junto ao devido gerenciamento dos dados referentes ao desempenho das redes para métricas específicas definidas. Nesse contexto, a ferramenta *Modulus Aggregator* destaca-se por ser capaz de fazer a agregação dos resultados gerados pelas *PINNs* treinadas de maneira a aproveitar a estrutura de diretórios nativa de um experimento *Modulus* e sem a necessidade do *upload* público de um experimento na plataforma do *Tensorboard*, como é requerido para o uso da *dataframe API* do *Tensorflow*.

O *Modulus Aggregator* foi desenvolvido como uma biblioteca a ser instalada no ambiente de desenvolvimento nativo do *framework Modulus*, e funciona como um pacote de instruções de interface de linha de comando que possibilitam ao usuário fazer a extração e a agregação de resultados de modelos treinados em um determinado experimento *Modulus*. Tal processo é efetuado primeiramente com a captura dos resultados salvos nos diretórios referentes a cada modelo, seguido de um pós-processamento dos resultados capturados. Com os resultados devidamente pós-processados, a agregação de dados é feita, juntamente com sua extração e armazenamento da base gerada no diretório de experimentos especificado.

Na presente dissertação, foi mostrada a dificuldade inerente ao processo de análise de resultados de modelos a partir exclusivamente da ferramenta *Tensorboard*, especialmente quando trata-se de um conjunto de múltiplos modelos treinados. Desse modo, a ferramenta de agregação desenvolvida foi capaz de auxiliar nesse processo reduzindo o espaço de análise de hiperparâmetros a partir da agregação dos resultados seguida da filtragem de modelos em função de métricas pré fixadas, mostrando-se assim não só o potencial de redução do espaço de análise de modelos e o armazenamento físico de dados, conforme mostram os experimentos, mas também a

plausibilidade da automatização de um procedimento desse gênero, o que não seria possível com o uso exclusivo da *dataframe API* do *Tensorflow*, dada a necessidade de publicização do experimento que, além de ser um procedimento de complicada automatização, é simplesmente inviável em determinados cenários de pesquisa e desenvolvimento científico.

Ainda existem limitações a serem enfrentadas a fim de que a ferramenta apresente ainda mais flexibilidade junto ao processo de extração de dados. Por exemplo, é perfeitamente razoável que seja desenvolvida algum tipo de funcionalidade nativa do *Modulus Aggregator* que possibilite a filtragem automática dos modelos a partir de um conjunto de métricas utilizadas como parâmetro de entrada das instruções. Além disso, ainda não é possível que seja feita a captura de dados de proveniência em tempo real por conta da natureza pouco intrusiva da solução. Porém, não descartam-se estudos que utilizem a ferramenta de maneira a executar a extração de dados de maneira automática durante a geração das *PINNs*. Tal estratégia poderia apresentar-se como interessante em um cenário que se utilize de paralelização para o treinamento de diversas redes de maneira simultânea. Por fim, a captura de metadados referentes à performance do procedimento de extração em si, como tempo de execução e utilização de recursos computacionais como CPU e memória podem enriquecer ainda mais uma possível investigação a ser realizada em âmbito técnico pelo especialista.

Referências Bibliográficas

ABADI, M., BARHAM, P., CHEN, J., et al., "TensorFlow: A system for large-scale machine learning", p. 21.

ABU-MOSTAFA, Y. S., MAGDON-ISMAIL, M., LIN, H.-T., 2012, *Learning From Data*. AMLBook.

ALMAJID, M. M., ABU-AL-SAUD, M. O., 2022, "Prediction of porous media fluid flow using physics informed neural networks", *Prediction of porous media fluid flow using physics informed neural networks*, v. 208 (jan.), pp. 109205. ISSN: 09204105. doi: 10.1016/j.petrol.2021.109205. Disponível em: <`https://linkinghub.elsevier.com/retrieve/pii/S0920410521008597`>.

BARBOSA, C. H. S., "Multi-GPU 3-D Reverse Time Migration with Minimum I/O", p. 14.

CHEN, F., SONDAK, D., PROTOPAPAS, P., et al., 2020, "NeuroDiffEq: A Python package for solving differential equations with neural networks", *Journal of Open Source Software*, v. 5, n. 46 (fev.), pp. 1931. ISSN: 2475-9066. doi: 10.21105/joss.01931. Disponível em: <`https://joss.theoj.org/papers/10.21105/joss.01931`>.

CHOLLET, F., OTHERS, 2015. "Keras". Disponível em: <`https://github.com/fchollet/keras`>. Publisher: GitHub.

CUOMO, S., DI COLA, V. S., GIAMPAOLO, F., et al., 2022. "Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next". jun. Disponível em: <`http://arxiv.org/abs/2201.05624`>. arXiv:2201.05624 [physics].

DENG, L., 2014, "Deep Learning: Methods and Applications", *Foundations and Trends® in Signal Processing*, v. 7, n. 3-4, pp. 197–387. ISSN: 1932-8346, 1932-8354. doi: 10.1561/2000000039. Disponível em: <`http://nowpublishers.com/articles/foundations-and-trends-in-signal-processing/SIG-039`>.

Eckmiller, R., v.d. Malsburg, C. (Eds.), 1989, *Neural Computers*. Berlin, Heidelberg, Springer Berlin Heidelberg. ISBN: 978-3-540-50892-2 978-3-642-83740-1. doi: 10.1007/978-3-642-83740-1. Disponível em: <`http://link.springer.com/10.1007/978-3-642-83740-1`>.

GHARIBI, G., WALUNJ, V., ALANAZI, R., et al., 2019, "Automated Management of Deep Learning Experiments". In: *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning - DEEM'19*, pp. 1–4, Amsterdam, Netherlands. ACM Press. ISBN: 978-1-4503-6797-4. doi: 10.1145/3329486.3329495. Disponível em: <`http://dl.acm.org/citation.cfm?doid=3329486.3329495`>.

GOODFELLOW, I., BENGIO, Y., COURVILLE, A., 2016, *Deep Learning*. MIT Press.

HAGHIGHAT, E., JUANES, R., 2021, "SciANN: A Keras/Tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks", *Computer Methods in Applied Mechanics and Engineering*, v. 373 (jan.), pp. 113552. ISSN: 00457825. doi: 10.1016/j.cma.2020.113552. Disponível em: <`http://arxiv.org/abs/2005.08803`>. arXiv:2005.08803 [cs].

HENNIGH, O., NARASIMHAN, S., NABIAN, M., et al., 2021, "NVIDIA SimNet™: An AI-Accelerated Multi-Physics Simulation Framework", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 12746 LNCS, pp. 447–461. ISSN: 0302-9743. doi: 10.1007/978-3-030-77977-1_36. ISBN: 9783030779764.

HUANG, G., LIU, Z., VAN DER MAATEN, L., et al., 2018. "Densely Connected Convolutional Networks". jan. Disponível em: <`http://arxiv.org/abs/1608.06993`>. arXiv:1608.06993 [cs].

HUANG, S., FENG, W., TANG, C., et al., 2022. "Partial Differential Equations Meet Deep Neural Networks: A Survey". out. Disponível em: <`http://arxiv.org/abs/2211.05567`>. arXiv:2211.05567 [cs].

IDREOS, S., GROFFEN, F., NES, N., et al., 2012, "MonetDB: Two Decades of Research in Column-oriented Database Architectures." *IEEE Data Eng. Bull.*, v. 35, n. 1, pp. 40–45. Disponível em: <`http://dblp.uni-trier.de/db/journals/debu/debu35.html#IdreosGNMMK12`>.

KARNIADAKIS, G. E., KEVREKIDIS, I. G., LU, L., et al., 2021, "Physics-informed machine learning", *Nature Reviews Physics*, v. 3, n. 6 (jun.), pp. 422–440. ISSN: 2522-5820. doi: 10.1038/s42254-021-00314-5. Disponível em: <http://www.nature.com/articles/s42254-021-00314-5>.

KOLLMANNSBERGER, S., D'ANGELLA, D., JOKEIT, M., et al., 2021, *Deep Learning in Computational Mechanics: An Introductory Course*, v. 977, *Studies in Computational Intelligence*. Cham, Springer International Publishing. ISBN: 978-3-030-76586-6 978-3-030-76587-3. doi: 10.1007/978-3-030-76587-3. Disponível em: <https://link.springer.com/10.1007/978-3-030-76587-3>.

KOVACS, A., EXL, L., KORNELL, A., et al., 2022, "Conditional physics informed neural networks", *Communications in Nonlinear Science and Numerical Simulation*, v. 104 (jan.), pp. 106041. ISSN: 10075704. doi: 10.1016/j.cnsns.2021.106041. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1007570421003531>.

KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E., 2012, "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*, v. 25. Curran Associates, Inc. Disponível em: <https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html>.

KUNSTMANN, L., PINA, D., SILVA, F., et al., 2021, "Online Deep Learning Hyperparameter Tuning based on Provenance Analysis", *Journal of Information and Data Management*, v. 12, n. 5 (nov.), pp. 396–414. ISSN: 2178-7107, 2178-7107. doi: 10.5753/jidm.2021.1924. Disponível em: <https://sol.sbc.org.br/journals/index.php/jidm/article/view/1924>.

LU, L., MENG, X., MAO, Z., et al., 2021, "DeepXDE: A deep learning library for solving differential equations", *SIAM Review*, v. 63, n. 1, pp. 208–228. ISSN: 0036-1445. doi: 10.1137/19M1274067.

MAO, Z., JAGTAP, A. D., KARNIADAKIS, G. E., 2020, "Physics-informed neural networks for high-speed flows", *Computer Methods in Applied Mechanics and Engineering*, v. 360 (mar.), pp. 112789. ISSN: 00457825. doi: 10.1016/j.cma.2019.112789. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0045782519306814>.

PASZKE, A., GROSS, S., MASSA, F., et al., 2019, "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: Wallach, H., Laro-

chelle, H., Beygelzimer, A., et al. (Eds.), *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., pp. 8024–8035.

PINA, D., KUNSTMANN, L., DE OLIVEIRA, D., et al., 2021a, "Provenance Supporting Hyperparameter Analysis in Deep Neural Networks", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 12839 LNCS, pp. 20–38. ISSN: 0302-9743. doi: 10.1007/978-3-030-80960-7_2. ISBN: 9783030809591.

PINA, D., NEVES, L., OLIVEIRA, D. D., et al., 2021b, "Captura Automática de Dados de Proveniência de Experimentos de Aprendizado de Máquina com Keras-Prov". In: *Anais Estendidos do XXXVI Simpósio Brasileiro de Bancos de Dados*, pp. 69–74, Porto Alegre, RS, Brasil, b. SBC. doi: 10.5753/sbbd_estendido.2021.18165. Disponível em: <https://sol.sbc.org.br/index.php/sbbd_estendido/article/view/18165>. ISSN: 0000-0000 event-place: Rio de Janeiro.

PINA, D. B., 2020, *Captura de dados de proveniência para apoiar a análise de hiperparâmetros em redes de aprendizado profundo*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, abr.

RAISSI, M., PERDIKARIS, P., KARNIADAKIS, G., 2019, "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 (fev.), pp. 686–707. ISSN: 00219991. doi: 10.1016/j.jcp.2018.10.045. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0021999118307125>.

RAISSI, M., YAZDANI, A., KARNIADAKIS, G. E., 2020, "Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations", *Science*, v. 367, n. 6481 (fev.), pp. 1026–1030. ISSN: 0036-8075, 1095-9203. doi: 10.1126/science.aaw4741. Disponível em: <https://www.science.org/doi/10.1126/science.aaw4741>.

SILVA, R. M., PINA, D., KUNSTMANN, L., et al., 2021, "Capturing Provenance to Improve the Model Training of PINNs: first hand- on experiences with Grid5000", p. 7.

SILVA, V., DE OLIVEIRA, D., VALDURIEZ, P., et al., 2018, "Dfanalyzer: runtime dataflow analysis of scientific applications using provenance", *Proceedings*

of the VLDB Endowment, v. 11, n. 12 (ago.), pp. 2082–2085. ISSN: 2150-8097. doi: 10.14778/3229863.3236265. Disponível em: <https://dl.acm.org/doi/10.14778/3229863.3236265>.

SPINNER, T., SCHLEGEL, U., SCHÄFER, H., et al., 2020, “explAIner: A Visual Analytics Framework for Interactive and Explainable Machine Learning”, *IEEE Transactions on Visualization and Computer Graphics*, v. 26, n. 1 (jan.), pp. 1064–1074. ISSN: 1941-0506. doi: 10.1109/TVCG.2019.2934629. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

STIASNY, J., MISYRIS, G. S., CHATZIVASILEIADIS, S., 2021, “Physics-Informed Neural Networks for Non-linear System Identification for Power System Dynamics”. In: *2021 IEEE Madrid PowerTech*, pp. 1–6, Madrid, Spain, jun. IEEE. ISBN: 978-1-66543-597-0. doi: 10.1109/PowerTech46648.2021.9495063. Disponível em: <https://ieeexplore.ieee.org/document/9495063/>.

SUN, L., GAO, H., PAN, S., et al., 2020, “Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data”, *Computer Methods in Applied Mechanics and Engineering*, v. 361 (abr.), pp. 112732. ISSN: 00457825. doi: 10.1016/j.cma.2019.112732. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S004578251930622X>.

THUEREY, N., HOLL, P., MUELLER, M., et al., 2021, *Physics-based Deep Learning*. WWW. Disponível em: <https://physicsbaseddeeplearning.org>.

ZHANG, S., LIU, R., TAO, X., et al., 2021, “Deep Cross-Corpus Speech Emotion Recognition: Recent Advances and Perspectives”, *Frontiers in Neurorobotics*, v. 15. ISSN: 1662-5218. doi: 10.3389/fnbot.2021.784514.