An Introduction to Physics-informed Neural Networks

Ezequiel S. D. Santos

13 de dezembro de 2022

A b s t r a c t

The interest in the use of Machine Learning and how it works for having a high performance in daily problems is growing. In this article we show a little how PINNs work, which is one of the Machine Learning techniques. And we show some of its applications. The analysis was performed using the PyTorch framework.

1 Introdução

O desenvolvimento do Deep Learning se originou na década de 40, com as contribuições de McCulloch e Pitts Publish que produziram o primeiro modelo matemático de uma Neural Network e Alan Turing com o desenvolvimento de uma bomba eletromecânica capaz de decifrar a máquina Enigma [6, 1]. O Deep Learning é uma subárea do Machine Learning (ML) que, por sua vez, é uma subárea da Artificial Intelligence (AI). A Artificial Intelligence é uma tecnologia capaz de criar máquinas que simulem o aprendizado humano através de padrões [12] e o Machine Learning permite que sistemas aprendam e melhorem a partir de experiências [11]. Com isso o Deep Learning se baseia no uso das Neural Networks onde os algoritmos possuem estruturas motivadas por redes neurais do cérebro humano, possuindo assim a capacidade de imitar o processo de aprendizado humano quando o cérebro recebe informações do ambiente ao seu redor [7]. Neste trabalho faremos um estudo de uma das ferramentas de Deep Learning, as Physics-informed Neural Networks (PINNs).

As PINNs utilizam técnicas para resolver equações diferenciais, encontrando uma Neural Network capaz de aproximar as soluções de uma EDP. No qual a Neural Network é obtida minimizando uma função de custo/perda baseando-se nos dados de entrada.

Na Seção 2 foram introduzidos os principais conceitos de ML e Matemática para conceber este trabalho. Na Seção 3 foram aplicados métodos de PINNs para encontrar uma solução aproximada de uma EDP. Por fim, a Seção 4 traz detalhes importantes que foram aplicados na metodologia.

2 Fundamentação Teórica

2.1 Deep Learning

O Deep Learning se baseia em Neural Networks. Basicamente sua aplicação se tem através de uma entrada de um conjunto de dados para se obter uma saída (como um conjunto de regras) [4]. Veremos na Subseção 2.9 uma aplicação de um modelo baseado nas Neural Networks.

Para melhor entendimento, uma Neural Network é composta por diversas camadas constituídas de neurônios, cujas camadas são divididas em: camada de entrada sendo o input (que recebe os dados de entrada), camada de saída sendo o output (que retorna o resultado) e as camadas ocultas ao qual são conhecidas como camada hidden (que fazem cálculos para facilitar a Neural Network a encontrar os outputs) [5].

Como exemplo, pegando um conjunto de dados com $K \in \mathbb{N}$ registros. Sejam $i = 1, \ldots, K$ e $n \in \mathbb{N}$. Tomaremos $x_i \in \mathbb{R}^n$ sendo o vetor input de comprimento n, isto é, para n parâmetros temos o vetor input $x_i \in \mathbb{R}^n$. Para o output, tomaremos a função $y : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ tal que $y(x_i)$ é o vetor output de comprimento $m \in \mathbb{N}$. Logo, o par $(x_i, y(x_i))$ é o $i-$ésimo exemplo de treinamento. A figura 1 ilustra uma Neural Network.

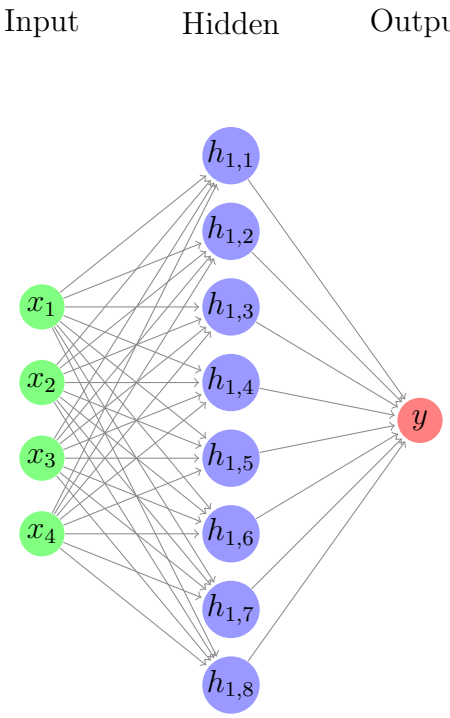


Figura 1: Ilustração de uma Neural Network.

2.2 Funções de Ativação

Cada rede neural do cérebro humano, além das conexões entre si, possuem neurônios importantes para tomar decisões. Por exemplo, se um humano percebe algum sinal de perigo, seu cérebro ativa imediatamente os neurônios responsáveis por se defender ou fugir do perigo. Isso ocorre de forma análoga nas Neural Networks. A ativação de um neurônio no Deep Learning é feito através de uma função de ativação.

2.2.1 A função sigmoid

A função sigmoid $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$ definida por

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

desempenha o papel de ativação para uma Neural Network.

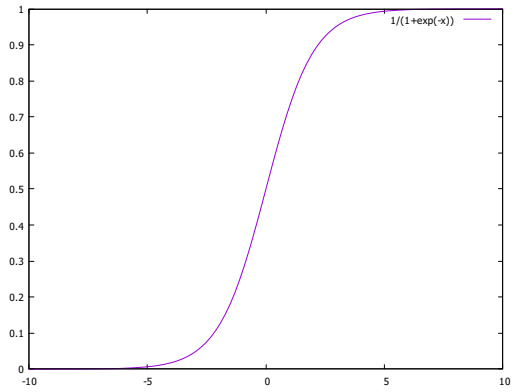


Figura 2: Função sigmoid.

2.3 A função Tanh

A função tanh $tanh : \mathbb{R} \longrightarrow \mathbb{R}$ definido por

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.2}$$

desempenha o papel de ativação para uma Neural Network. Como esta função se assemelha a função sigmóide, ela pode ser escrita como

$$tanh(x) = 2 \cdot \sigma(2x) - 1 \tag{2.3}$$

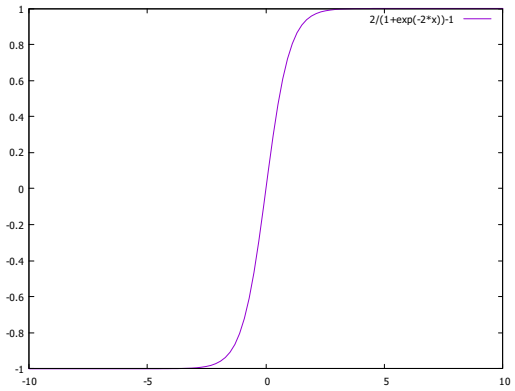


Figura 3: Função Tanh.

2.4 Neural Networks

Na Figura 4, segue a arquitetura de uma Neural Network. Cada camada de neurônios conectados entre si possui seu próprio weight, bias e ativação. Neste caso, pegamos como exemplo a função sigmoid para desempenhar o papel de ativação. Em geral, os wights, denotados por $w_{ij}^{[l]}$, e bias, denotados por $b_i^{[l]}$, são considerados os parâmetros de uma Neural Network. Em que i representa o neurônio da camada $l = L_n$ (onde n é quantidade de camadas e $l = 2, \ldots, n$) e j é o neurônio da camada $l - 1 = L_{n-1}$.

Cada neurônio da primeira camada (L_1) recebe uma informação. Consequentemente, cada neurônio da próxima camada ganha um weight e, após isso, combina essas informações da camada L_1, emite seu próprio bias e aplica a função de ativação (ou função sigmoide). O

output de um neurônio i na camada l é expresso como

$$y_i^{[l]} = \sigma\left(\sum_{j=1}^{N_{l-1}} w_{ij}^{[l]} y_j^{[l-1]} + b_i^{[l]}\right) \tag{2.4}$$

com $N_l \in \mathbb{N}^+$ sendo a quantidade de neurônios da camada l e $N_{l-1} \in \mathbb{N}^+$ sendo a quantidade de neurônios da camada $l-1$.

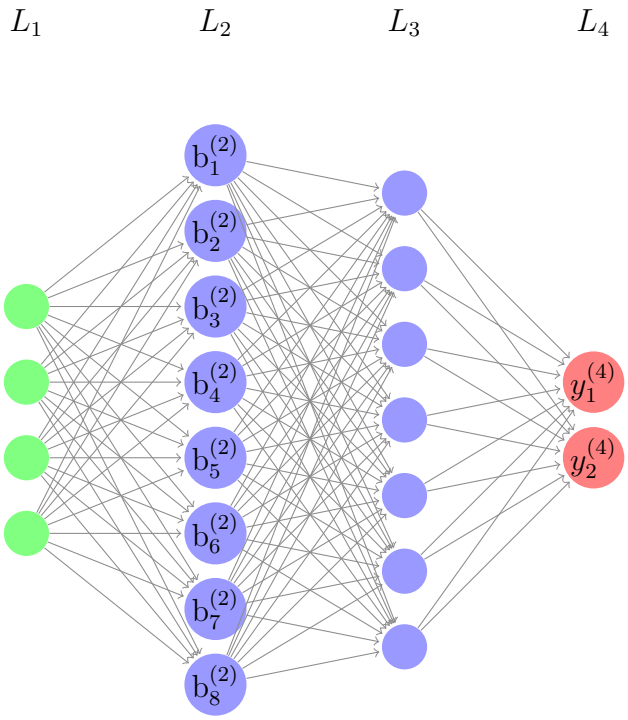


Figura 4: Neural Network.

Agora, sem perda de generalidade, podemos tomar a matriz $W^{[l]}$, contendo todos os weights, como

$$W = \begin{pmatrix} w_{11}^{[l]} & \dots & w_{1N_{l-1}}^{[l]} \\ \vdots & \ddots & \vdots \\ w_{N_l 1}^{[l]} & \dots & w_{N_l N_{l-1}}^{[l]} \end{pmatrix} \tag{2.5}$$

Além disso, na camada l tomamos o vetor de bias como $b^{[l]} = \left[b_1^{[l]}, \dots, b_k^{[l]}\right]$ e o vetor de ativação como $y^{[l]} = \left[y_1^{[l]}, \dots, y_k^{[l]}\right]$. Logo, a Equação 2.4 pode ser reescrita fazendo-se

$$y^{[l]} = \sigma\left(W^{[l]} y^{[l-1]} + b^{[l]}\right). \tag{2.6}$$

Ou seja, isso mostra que uma Neural Network é uma composição de funções.

2.5 A função de Custo

A função de custo é usada para medir o erro entre o output que foi previsto e o que se sabia de todos os exemplos de treinamento. Em outras palavras, a função de custo verifica se o output da rede é parecido com o que era esperado. Com isso, melhoramos os weights e bias das camadas em uma Neural Network. Por simplicidade, suponha que, dada uma classificação, nossa rede obtém um output de 0.5, contudo esperava-se 2.0. Logo o erro da rede foi de $2.0 - 0.5 = 1.5$.

Mas, como o nosso conjunto de dados de treinamento é geralmente grande e precisamos fazer a média de todos os dados para que tenhamos o erro, adotar-se-á uma função que retorne somente valores positivos. Para resolver isso, usamos como base o erro quadrático médio (assumindo $||\cdot||$ como a norma euclidiana) para chegar na seguinte expressão da função de custo

$$\frac{1}{2N} \cdot \sum_{i=1}^{N} ||y_i(x_i) - \hat{y}_i(x_i)||^2 \tag{2.7}$$

onde N é a cardinalidade do conjunto de dados, $x_i \in \mathbb{R}^n$ é o input, $y_i(x_i) \in \mathbb{R}^m$ é o output da Neural Network e $\hat{y}_i(x_i)$ é o output esperado.

2.6 Equações Diferenciais Parciais

Se uma equação possui uma relação entre a variável independente x, a função incógnita $y = f(x)$ e suas derivadas então esta equação é vista como uma equação parcial da forma

$$F(x, y, y', \ldots, y^{(n)}) = 0.$$

Agora se a função incógnita da equação diferencial for uma função de mais de uma variável então essa equação diferencial é dita como uma equação diferencial parcial (EDP).

2.6.1 Equação de Burgers

A equação de Burgers é da forma

$$u_t + auu_x = \mu u_{xx},\ x \in \mathbb{R},\ t > 0 \tag{2.8}$$

onde $a \neq 0$ e $\mu > 0$ são constantes. Ela foi inserida pelo matemático e físico Johannes Martinus Burgers no seu trabalho sobre turbulência em fluidos [3]. Suas aplicações possuem foco, principalmente, na física.

2.7 Método Steepest Descent

O método steepest descent é usado para minimizar o erro entre o output calculado (y) e o output esperado ($\hat{y}$) [2, 10, 9]. Para isso, redefinimos iterativamente os parâmetros em uma Neural Network. Com efeito, seja $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ diferenciável em x_0. Tome $\varphi(t) = f(x_0 + tu)$, onde $||u|| = 1$. Derivando $\varphi(t)$ pela regra da cadeia

$$\begin{aligned} \varphi'(t) &= \frac{\partial f}{\partial x_1} \cdot \frac{\partial x_1}{\partial t} + \cdots + \frac{\partial f}{\partial x_n} \cdot \frac{\partial x_n}{\partial t} \\ &= \frac{\partial f}{\partial x_1} \cdot u_1 + \cdots + \frac{\partial f}{\partial x_n} \cdot u_n \\ &= \nabla f(x_0 + tu) \cdot u. \end{aligned}$$

Como $\varphi'(0) = \nabla f(x_0) \cdot u = ||\nabla f(x_0)|| \cos(\theta)$, com θ sendo o ângulo entre $\nabla f(x_0)$ e u, segue-se que $\varphi'(0)$ minimiza quando $\theta = \pi + 2\pi n$ para todo $n \in \mathbb{N}$. Isso significa que o método steepest descent mostra que dado um valor inicial x_0, e encontrando algum $t = t_n > 0$ que minimiza a função

$$\varphi_n(t) = f(x_n - t\nabla f(x_n)), \tag{2.9}$$

obtém-se

$$x_{n+1} = x_n - t_n \cdot \nabla f(x_n). \tag{2.10}$$

Podemos ver na Subseção 2.9.1 que o passo t influencia diretamente na minimização para uma Neural Network. Por isso, deve-se atentar a escolha do melhor passo.

2.8 Algoritmo Backpropagation

O algoritmo Backpropagation é usado para aprendizado supervisionado de Neural Networks usando gradient/steepest descent [8]. Para tal, usaremos a forma simples para a função de custo no qual é vista como

$$C = \frac{1}{2}||y^{[l]} - \hat{y}||^2.$$

Assim, assumimos que

$$z^{[l]} = W^{[l]}y^{[l-1]} + b^{[l]}.$$

Temos que

$$y^{[l]} = \sigma(z^{[l]}).$$

Por conseguinte, aplicando a regra da cadeia para calcular a sensibilidade da função de custo em relação ao weight entre as camadas hidden e output, chegamos em

$$\frac{\partial C}{\partial w_{ij}^{[l]}} = \frac{\partial C}{\partial z_i^{[l]}} y_j^{[l-1]} = \frac{\partial y_i^{[l]}}{\partial z_i^{[l]}} \frac{\partial C}{\partial y_i^{[l]}} y_j^{[l-1]} = \sigma'(z_i^{[l]}) \circ (y_i^{[l]} - \hat{y}_i) \circ y_j^{[l-1]}, \tag{2.11}$$

em que $y_j^{[l-1]} = \frac{\partial z_i^{[l]}}{\partial w_{ij}^{[l]}}$. Vale ressaltar que as equações anteriores são válidas para todos os casos em que $2 \leq l \leq n$, onde n é a quantidade de camadas.
Da mesma forma, calculamos a sensibilidade da função de custo em relação ao bias entre as camadas hidden e output:

$$\frac{\partial C}{\partial b_i^{[l]}} = \frac{\partial C}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial b_i^{[l]}} = \frac{\partial C}{\partial z_i^{[l]}}. \tag{2.12}$$

Agora para atualizar os weights usamos a Equação 2.11 e para os bias usamos a Equação 2.12. Isso resulta em

$$W^{[l]} = W^{[l]} - \alpha \nabla C = W^{[l]} - \alpha \frac{\partial C}{\partial W^{[l]}}; \tag{2.13}$$

$$b^{[l]} = b^{[l]} - \alpha \nabla C = b^{[l]} - \alpha \frac{\partial C}{\partial b^{[l]}}, \tag{2.14}$$

onde $\alpha > 0$. Lembrando que $W^{[l]}$ é a matriz de weights e $b^{[l]}$ é o vetor de bias.

2.9 Aplicação

Nesta subseção, veremos a aplicação do Backpropagation.

2.9.1 Backpropagation

Foi gerado um conjunto de dados definidos por pontos em $\mathbb{R}^2$. Os pontos estão divididos em preto e vermelho, como é visto na Figura 5.

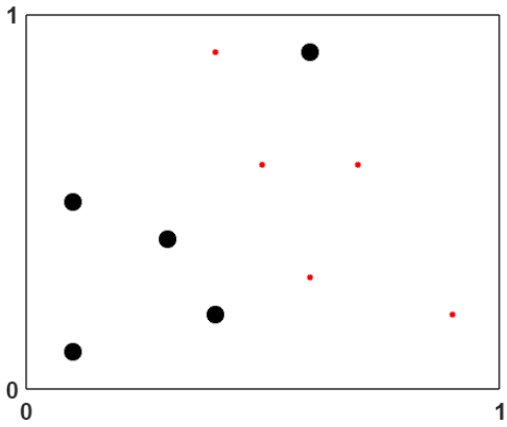
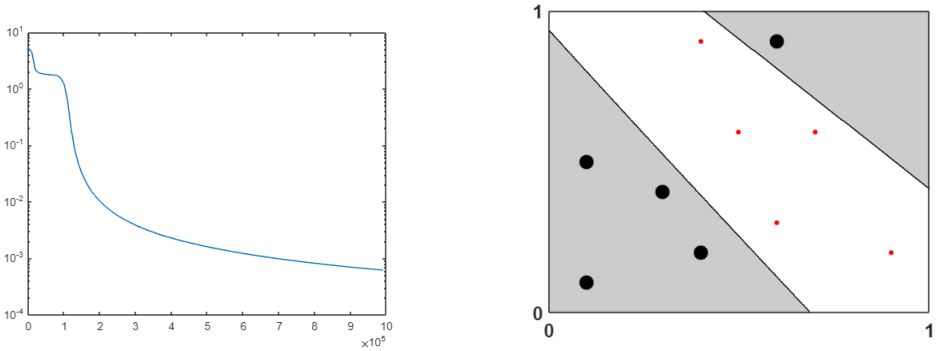


Figura 5: Conjunto de dados em $\mathbb{R}^2$.

Na próxima etapa aplicamos o Backpropagation para treinar nossa Neural Network (com $\alpha = 0.05$ e $\alpha = 0.005$) para o processo de classificação que tem como objetivo separar os pontos pretos dos vermelhos no conjunto de dados (Ver Figura 5). Abaixo seguem os resultados para diferentes passos α.

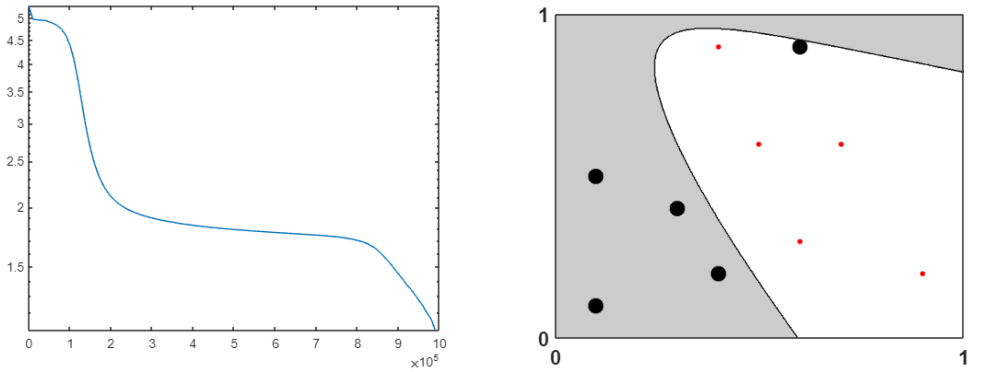
- Para $\alpha = 0.05$:



(a) Função de custo para $\alpha = 0.05$ (b) Curva de separação para $\alpha = 0.05$

Figura 6: Aplicação do Backpropagation.

- Para $\alpha = 0.005$:



(a) Função de custo para $\alpha = 0.005$ (b) Curva de separação para $\alpha = 0.005$

Figura 7: Aplicação do Backpropagation.

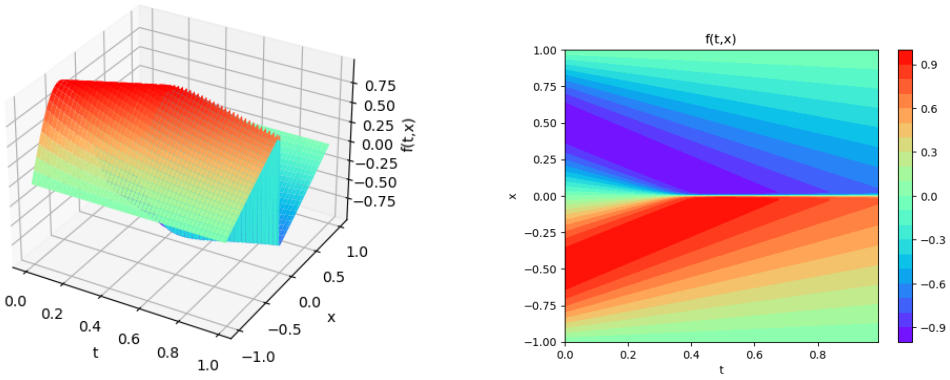
3 Metodologia

3.1 PINNs

Para este trabalho será considerada a equação de Burgers com as condições de contorno de Dirichlet

$$u_t + auu_x - \mu u_{xx} = 0,\ x \in [-1, 1],\ t \in [0, 1], \tag{3.15}$$
$$u(0, x) = -\sin(\pi x),$$
$$u(t, -1) = u(t, 1) = 0.$$

onde $\alpha = 1$ e o coeficiente de difusão é $\mu = \frac{0.01}{\pi}$.



(a) $\alpha = 1$ e $\mu = \frac{0.001}{\pi}$ (b) $\alpha = 1$ e $\mu = \frac{0.001}{\pi}$

Figura 8: Solução Numérica.

Como já visto, podemos expressar uma Neural Network como uma composição de funções da forma

$$NN(X) = W_n\sigma_{n-1}(W_{n-1}\sigma_{n-2}(\dots W_2(W_1X + b_1) + b_2) + \cdots) + b_{n-1}) + b_n.$$

Sendo assim, podemos usar uma Neural Network $NN(x, t)$ para se obter uma aproximação da função exata $u(x, t)$. Pela equação de Burgers, tem-se que

$$\frac{\partial NN}{\partial t} + N\left(\frac{\partial NN}{\partial x}\right) - \mu\frac{\partial^2 NN}{\partial x^2} \approx u_t + uu_x - \mu u_{xx} = 0 \tag{3.16}$$

Definimos da relação (3.16) a função

$$f(t, x) = \frac{\partial NN}{\partial t} + N\left(\frac{\partial NN}{\partial x}\right) - \mu\frac{\partial^2 NN}{\partial x^2} \tag{3.17}$$

Portanto:

1. A minimização de uma função perda relacionada a f é

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2 \tag{3.18}$$

onde N_f é o número de pontos no domínio (t, x).

2. Como já se sabe a resposta do conjunto de treinamento, pegamos N_u pontos da condição de contorno e condições iniciais para treinar a rede

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |y(t_u^i, x_u^i) - NN(t_u^i, x_u^i)|^2 \tag{3.19}$$

Logo, a função de custo final é

$$MSE = MSE_f + MSE_u \tag{3.20}$$

3.2 Parâmetros

Foi fixado o número total de pontos para o treinamento tomando $N_u = 200$ e o número de pontos no domínio (t, x) tomando $N_f = 10,000$. E para estes parâmetros utilizou-se 8 camadas profundas, onde cada camada contém 20 neurônios.

3.3 Neural Network

A criação da Neural Network foi feita pelo Torch, que é uma biblioteca de aprendizado de máquina de código aberto. A função de ativação usada na Neural Network foi a função Tanh (tangente hiperbólica).

```
    def __init__(self,cd):
        super().__init__()
        self.ls = nn.ModuleList([nn.Linear(cd[i], cd[i+1]) for i in
           range(len(cd)-1)])
        self.F_Perda = nn.MSELoss(reduction ='mean')
        self.atv = nn.Tanh()
        self.it = 0
        for j in range(len(cd)-1):
           nn.init.xavier_normal_(self.ls[j].weight.data, gain=1.0)
           nn.init.zeros_(self.ls[j].bias.data)
    def parte1(self,t):
        if torch.is_tensor(t)!= True:
            t = torch.from_numpy(t)
        U = torch.from_numpy(Ub).float().to(device)
        L = torch.from_numpy(Lb).float().to(device)
        t = (t - U)/(U - L)
        h = t.float()
        for i in range(len(cd)-2):
            k = self.ls[i](h)
            h = self.atv(k)
        h = self.ls[-1](h)
        return h
    def parte2(self,x,y):
        L_u = self.F_Perda(self.parte1(x), y)
        return L_u
    def parte3(self, X_Treino_NNf):
        p = X_Treino_NNf.clone()
        p.requires_grad = True
        U = self.parte1(p)
```

```
        Utx = autograd.grad(U,p,torch.ones([X_Treino_NNf.shape[0], 1])
           .to(device), retain_graph=True, create_graph=True)[0]
        Uttxx = autograd.grad(Utx,p,torch.ones(X_Treino_NNf.shape).to(
           device), create_graph=True)[0]
        Ux = Utx[:,[0]]
        Ut = Utx[:,[1]]
        Uxx = Uttxx[:,[0]]
        F = Ut + (self.parte1(p))*(Ux) - (mu)*Uxx
        L_f = self.F_Perda(F,Tx_f)
        return L_f
    def parte4(self,x,y,X_Treino_NNf):
        L_u = self.parte2(x,y)
        L_f = self.parte3(X_Treino_NNf)
        L_total = L_u + L_f
        return L_total
    def parte5(self):
        otim.zero_grad()
        L = self.parte4(X_Treino_NNu, U_Treino_NNu, X_Treino_NNf)
        self.it+=1
        L.backward()
        if self.it%100== 0:
            erro, _ = PINNs.parte6()
        return L
    def parte6(self):
        U_predicao = self.parte1(X_teste)
        L2_norm = torch.linalg.norm((U-U_predicao),2)/torch.linalg.
           norm(U,2)
        U_predicao = U_predicao.cpu().detach().numpy()
        U_predicao = np.reshape(U_predicao,(256,100),order='F')
        return L2_norm, U_predicao
```

- parte1: Passo da Neural Network (feedforward)
- parte2: Cálculo do MSE_u
- parte3: Cálculo do MSE_f
- parte4: Cálculo da função de custo total $MSE = MSE_u + MSE_f$
- parte5: Otimizador (LBFGS)
- parte6: Teste da Neural Network

Ademais, as predições foram mostradas em diferentes intervalos de tempo: $t = 0.25, 0.50, 0.75$.

4 Resultados

Com os métodos aplicados, foi feita uma avaliação do modelo. Com isso observou-se que a solução predita se aproxima muito bem da solução numérica $u(t, x)$, com um erro relativo de aproximadamente $0.5e^{-3}$. A Figura 9 mostra a solução aproximada de $u(t, x)$ realizada pela Neural Network.

Além disso, podemos observar na Figura 10 o desempenho da solução prevista, em que foi realizado uma comparação entre as soluções exata e prevista em intervalos de tempo $t = 0.25,\ 0.50,\ 0.75$.

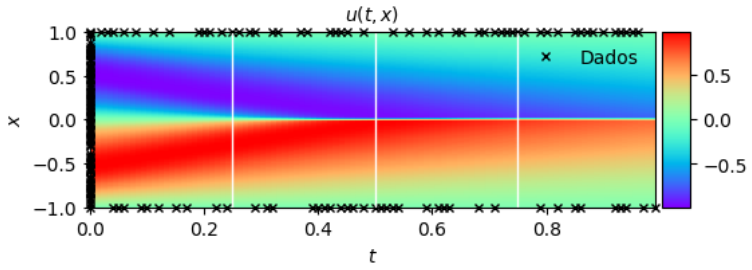


Figura 9: Solução análitica da Neural Network para $u(t, x)$.

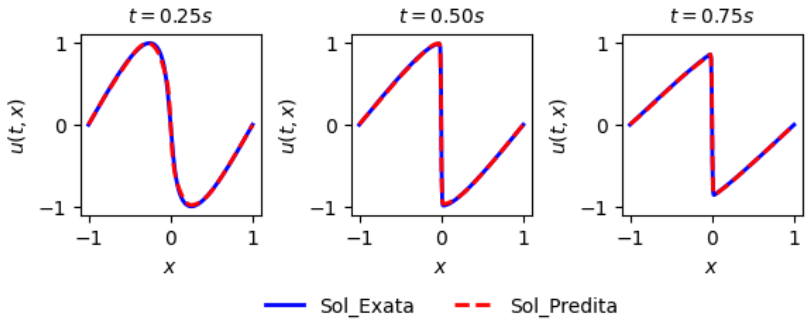


Figura 10: Comparação das soluções exata e predita em $t = 0.25,\ 0.50,\ 0.75$.

Veja que o erro relativo diminui conforme aumentamos o número de camadas ocultas e neurônios. Isto é, a precisão da rede aumenta na medida que as camadas ocultas e neurônios aumentam.

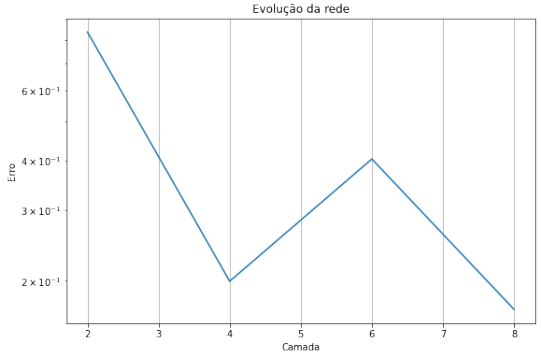


Figura 11: Evolução da rede com 10 neurônios.

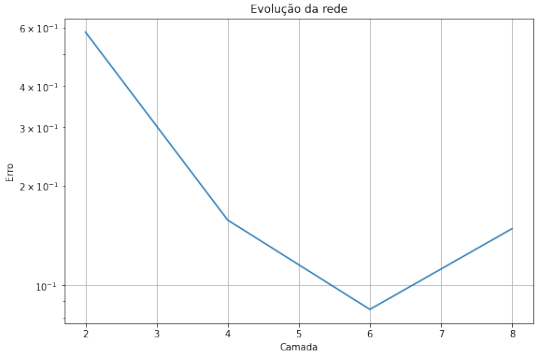


Figura 12: Evolução da rede com 20 neurônios.

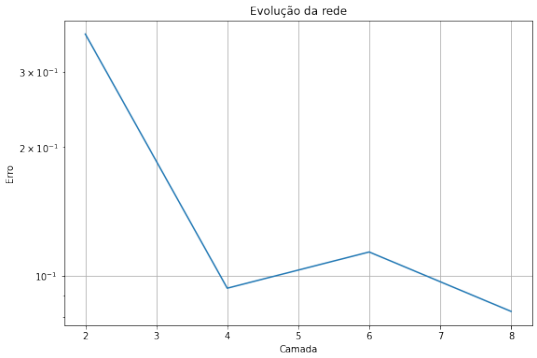


Figura 13: Evolução da rede com 40 neurônios.

No total o modelo levou cerca de 490 segundos para finalizar todos os processos numa placa Intel HD Graphics 5500 (GT2).

5 Considerações Finais

Foi visto neste trabalho como as physics-informed neural networks são eficientes para resolver problemas físicos como o fenômeno de turbulência. As PINNs, por sua vez, possuem uma alta capacidade de adequação às restrições impostas durante seu processo de aprendizagem e, além disso, um custo computacional baixo, o que torna suas aplicações mais promissoras. Desta forma, eles podem chegar em uma solução mais generalizada mesmo que a quantidade de dados disponíveis para o treinamento seja limitada/pouca. Logo as soluções de uma EDP podem ser encontradas sem mesmo saber as condições de contorno. Para qualquer solução encontrada nas PINNs, precisamos fornecer apenas algumas informações físicas do problema e os dados de treinamento (não tendo a necessidade de serem em grande quantidade).

Este trabalho pode ser aplicado em diversos outros problemas, não se limitando apenas às abordagens propostas aqui. Suas aplicações estão em áreas ligadas à Biologia, Física, Química, Economia, etc.

Referências

[1] Mcculloch pitts publish the first mathematical model of a neural network. `https://www.historyofinformation.com/detail.php?entryid=782`, 1943.

[2] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[3] Johannes Martinus Burgers. Hydrodynamics.—application of a model system to illustrate some points of the statistical theory of free turbulence. In *Selected papers of JM Burgers*, pages 390–400. Springer, 1995.

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

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[6] James Grime. Maths from the talk "alan turing and the enigma machine", 2013.

[7] Aaron Courville Ian Goodfellow, Yoshua Bengio. Deep learning. 2016.

[8] Toshinori Munakata. *Fundamentals of the new artificial intelligence*, volume 2. Springer, 1998.

[9] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2003.

[10] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[11] Alex Smola and SVN Vishwanathan. Introduction to machine learning. *Cambridge University, UK*, 2008.

[12] Satya Prakash Yadav, Dharmendra Prasad Mahato, and Nguyen Thi Dieu Linh. *Distributed artificial intelligence: A modern approach*. CRC Press, 2020.