UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”
CÂMPUS EXPERIMENTAL DE SÃO JOÃO DA BOA VISTA
BACHARELADO EM ENGENHARIA AERONÁUTICA

MATHEUS FELIPE PACHECO

ESTUDO DA UTILIZAÇÃO DE ALGORITMOS DE DEEP-LEARNING JUNTO AO MÉTODO DOS ELEMENTOS FINITOS PARA OTIMIZAÇÃO DE ESTRUTURAS AERONÁUTICAS RETICULADAS

SÃO JOÃO DA BOA VISTA
2022

MATHEUS FELIPE PACHECO

ESTUDO DA UTILIZAÇÃO DE ALGORITMOS DE DEEP-LEARNING JUNTO AO MÉTODO DOS ELEMENTOS FINITOS PARA OTIMIZAÇÃO DE ESTRUTURAS AERONÁUTICAS RETICULADAS

Trabalho de Conclusão de Curso apresentado à Universidade Estadual Paulista "Júlio de Mesquita Filho" como requisito para obtenção de título de Bacharel em Engenharia Aeronáutica.

Orientador: Prof. Dr. Murilo Sartorato

SÃO JOÃO DA BOA VISTA

2022

P116e

Pacheco, Matheus Felipe

Estudo da utilização de algoritmos de deep-learning junto ao método dos elementos finitos para otimização de estruturas aeronáuticas reticuladas / Matheus Felipe Pacheco. -- São João da Boa Vista, 2022

61 p. : tabs., fotos

Trabalho de conclusão de curso (Bacharelado - Engenharia Aeronáutica) - Universidade Estadual Paulista (Unesp), Faculdade de Engenharia, São João da Boa Vista

Orientador: Murilo Sartorato

1. Aprendizado do computador. 2. Análise estrutural (Engenharia). 3. Otimização matemática. 4. Método dos elementos finitos. 5. Python (Linguagem de programação de computador). I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Engenharia, São João da Boa Vista. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"

FACULDADE DE ENGENHARIA - CÂMPUS DE SÃO JOÃO DA BOA VISTA

GRADUAÇÃO EM ENGENHARIA AERONÁUTICA

TRABALHO DE CONCLUSÃO DE CURSO

ESTUDO DA UTILIZAÇÃO DE ALGORITMOS DE DEEP-LEARNING JUNTO AO MÉTODO DOS ELEMENTOS FINITOS PARA OTIMIZAÇÃO DE ESTRUTURAS BIDIMENSIONAIS RETICULADAS

Aluno: Matheus Felipe Pacheco

Orientador: Prof. Dr. Murilo Sartorato

Banca Examinadora:

- Murilo Sartorato (Orientador)
- Emerson de Oliveira Batista (Examinador)
- Rita de Cássia Domingos (Examinadora)

A ata da defesa com as respectivas assinaturas dos membros encontra-se no prontuário do aluno (Expediente nº 057/2022)

São João da Boa Vista, 29 de junho de 2022

RESUMO

O presente trabalho se propõe a estudar o uso de técnicas de *Deep Learning* em conjunto com o método dos elementos finitos para resolver e otimizar estruturas simples, bidimensionais, treliçadas. Primeiramente, uma determinada geometria de treliça bidimensional com seções transversais variáveis é escolhida como caso de estudo; em seguida, o método dos elementos finitos é implementado usando a linguagem Python e usado para resolver o problema para uma população gerada aleatoriamente de conjuntos de áreas de seções transversais e um para um par de limites de deslocamento e tensão, que são gerados aleatoriamente dentro de um intervalo de valores. São utilizadas diferentes estratégias para gerar as populações, algoritmos de soma zero e diferentes distribuições estatísticas. Em seguida, uma rede é treinada para encontrar uma saída binária que indica se algum limite de tensão e deslocamento previamente definido foi excedido. Finalmente, essa rede treinada é usada como restrição para um algoritmo genético gerar famílias de áreas otimizadas, com objetivo de utilizar uma segunda rede que terá como entrada um valor para deslocamento máximo e tensão máxima e retornará as áreas otimizadas. Ao contrário da primeira rede neural, esta segunda rede neural possui uma saída não binária, sua resposta são os valores para as áreas que minimizam o peso da estrutura. A metodologia difere das usualmente utilizadas na literatura ao aplicar duas redes treinadas independentementes ao invés de uma. Para criar a arquitetura das redes, foram utilizadas as bibliotecas pré-implementadas em Python Keras, Tensor-Flow e Scikit-Learn. A metodologia é então utilizada para testar a influência de diferentes hiper parâmetros e diferentes estratégias de geração de população nas razões de convergência e significância dos resultados.

Palavras-Chave: Análise estrutural (Engenharia); Aprendizado do computadorg; Otimização matemática; Método dos elementos finitos; Python (Linguagem de programação de computador)

ABSTRACT / RESUMEN

The present work proposes to study the use of deep learning techniques in conjunction with the finite element method to solve and optimize simple, two-dimensional, lattice structures. First, a given two-dimensional truss geometry with variable cross-sections is chosen as a case study; then the finite element method is implemented using the Python language and used to solve the problem for a randomly generated population of sets of cross-sectional areas and one for a pair of displacement and stress limits, which are randomly generated within a range of values. Different strategies are used to generate the populations, zero-sum algorithms and different statistical distributions. Next, a network is trained to find a binary output that indicates whether any previously defined voltage and displacement thresholds have been exceeded. Finally, this trained network is used as a constraint for a genetic algorithm to generate families of optimized areas, in order to use a second network that will have as input a value for maximum displacement and maximum stress and will return the optimized areas. Unlike the first neural network, this second neural network has a non-binary output, its response is the values for the areas that minimize the weight of the structure. The methodology differs from those usually used in the literature by applying two independently trained networks instead of one. To create the architecture of the networks, pre-implemented libraries in Python Keras, Tensor-Flow and Scikit-Learn were used. The methodology is then used to test the influence of different hyperparameters and different population generation strategies on the convergence and significance ratios of the results.

Keywords: Structural analysis; Deep-Learning; Mathematical optimization; Finite Element Method; Python.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Redes neurais ou redes neurais artificiais (do inglês *neural network*, em tradução livre - NN) são uma ferramenta utilizadas do aprendizado de máquina (do inglês *machine learning* - *ML*) que por sua vez é um subconjunto dentro da grande área da inteligência artificial (Géron, 2018).

Com o avanço da tecnologia na área computacional, juntamente com o interesse das empresas de tratarem, analisarem e predizerem dados com mais eficiência, a grande área da inteligência artificial juntamente com suas ramificações como aprendizado profundo apresentaram um crescimento exponencial nos últimos quinze anos (Chollet, 2018).

A proposta do presente projeto é o estudo da utilização de técnicas de aprendizado profundo (do inglês deep learning, em tradução livre - DL) para resolução de estruturas reticuladas aeronáuticas. Para isso, uma revisão bibliográfica e aprendizado sobre Estruturas Aeronáuticas, Aprendizado de Máquina, Algoritmos Genéticos e o Método dos Elementos Finitos foi realizada

1.1. INTELIGÊNCIA ARTIFICIAL, MACHINE LEARNING E DEEP LEARNING

Inteligência artificial é uma subárea da ciência da computação que trata da capacidade de uma máquina de atuar como um ser humano em tarefas que são geralmente realizadas por ele.

Já o Machine learning é um subconjunto da Inteligência Artificial e é um algoritmo que consegue não só aprender por conta própria, sem precisar da inferência humana, como aprimorar seu desempenho através da experiencia adquirida em um treinamento.

Assim como o *ML*, *DL* também é um subconjunto do aprendizado de máquina Figura 1. A diferença do *ML* para o *DL* se deve ao fato de que algoritmos que utilizam *DL* transformam suas entradas usando mais camadas e neurônios por camadas. Em cada camada, a entrada é transformada por um neurônio artificial, cujos parâmetros são "aprendidos" por meio do treinamento, portanto a inteligência artificial pode ser definida com a capacidade das máquinas de pensarem e de replicarem o comportamento humano. Já o *machine learning* é a forma, ou seja, o algoritmo que permite que a inteligência artificial seja aplicada.

A rede consegue aprender a identificar padrões conforme atualiza seus pesos e vieses que funcionam como sinapses cerebrais. Através de uma função de custo, os pesos são

atualizados a cada iteração da rede até que se atinja o mínimo global, que resultará na solução do problema ou no treinamento de uma função com a maior precisão possível e menor taxa de perda.

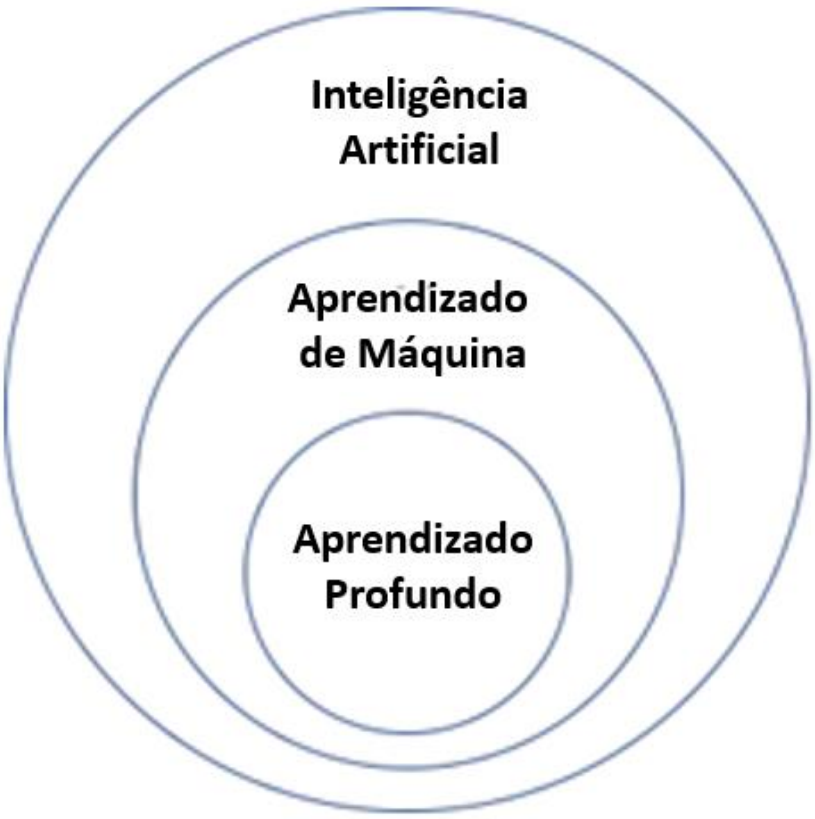


Figura 1. Posição do aprendizado profundo dentro da inteligência artificial (adaptado de IBM, 2020)

A utilização de *DL* como uma ferramenta para processamento de dados vem crescendo cada vez mais nos últimos anos por dois principais fatores que tiveram uma evolução exponencial na última década (Chollet, 2018). São eles:

- Poder computacional: como o aprendizado da rede ocorre através de várias camadas de neurônios, dependendo da complexibilidade do problema isso pode requerer um grande poder computacional, porém como os avanços em tecnologia na área da computação tanto em software como em hardwares tem um crescimento exponencial, isso facilitou a utilização dessa ferramenta, pois dessa forma a taxa de convergência para o mínimo global é alcançada mais rapidamente.

- Quantidade de dados: Para que uma rede com aprendizado profundo possa aprender e identificar os padrões de um problema de forma ótima, isso demanda uma quantidade de dados suficientemente grande para que a rede consiga aprender. Isso pode variar de caso para caso, mas assim como no tópico anterior, a capacidade de armazenamento, bem como a quantidade de dados que temos disponível também aumentou de forma a facilitar a utilização de *DL*.

1.2. REDES NEURAIS

Redes neurais artificiais como são conhecidas e implementadas são relativamente recentes, o aprendizado de máquina (ML) é uma ferramenta importante para o objetivo de alavancar tecnologias em torno da inteligência artificial. Por causa de suas habilidades de aprendizado e tomada de decisão, o aprendizado de máquina é frequentemente chamado de IA, embora, seja uma subdivisão da IA. Até o final da década de 1970, fazia parte da evolução da IA porém ele se ramificou para evoluir por conta própria. (Foote, 2021).

Atualmente, grande parte treinamento de reconhecimento de fala está sendo feito por uma técnica de Machine Learning chamada memória de longo prazo (do inglês ong short-term memory, na tradução livre), um modelo de rede neural descrito por Jürgen Schmidhuber e Sepp Hochreiter em 1997 (Hochreiter, 1997).

Em 2006, o um programa do Instituto Nacional de Padrões e Tecnologia – avaliou os algoritmos de reconhecimento facial populares da época. Varreduras de rosto em 3D, imagens de íris de olhos humanos e imagens de rosto de alta resolução foram testadas. Suas descobertas sugeriram que os novos algoritmos eram dez vezes mais precisos do que os algoritmos de reconhecimento facial de 2002 e 100 vezes mais precisos do que os de 1995 (NIST, 2010).

O aprendizado de máquina agora é responsável por alguns dos avanços mais significativos da tecnologia. Está sendo usado para a nova indústria de veículos autônomos, e para explorar a galáxia, pois ajuda na identificação de exoplanetas (Foote, 2021).

A busca pelo entendimento do cérebro humano e como os neurônios interagem entre si é uma ideia muito mais antiga do que imagina. O primeiro artigo onde houve uma busca do entendimento das interações entre neurônios em reconhecimento de padrões (Mcculloch, 1943). A principal ideia desse artigo foi uma comparação estudada entre o funcionamento de neurônios e funções binarias.

Após décadas de estudo e pesquisas, Yann LeCun (LeCun, 1989) conseguiu pela primeira vez criar um algoritmo de retro propagação para identificação de números em uma imagem (Figura 2) .

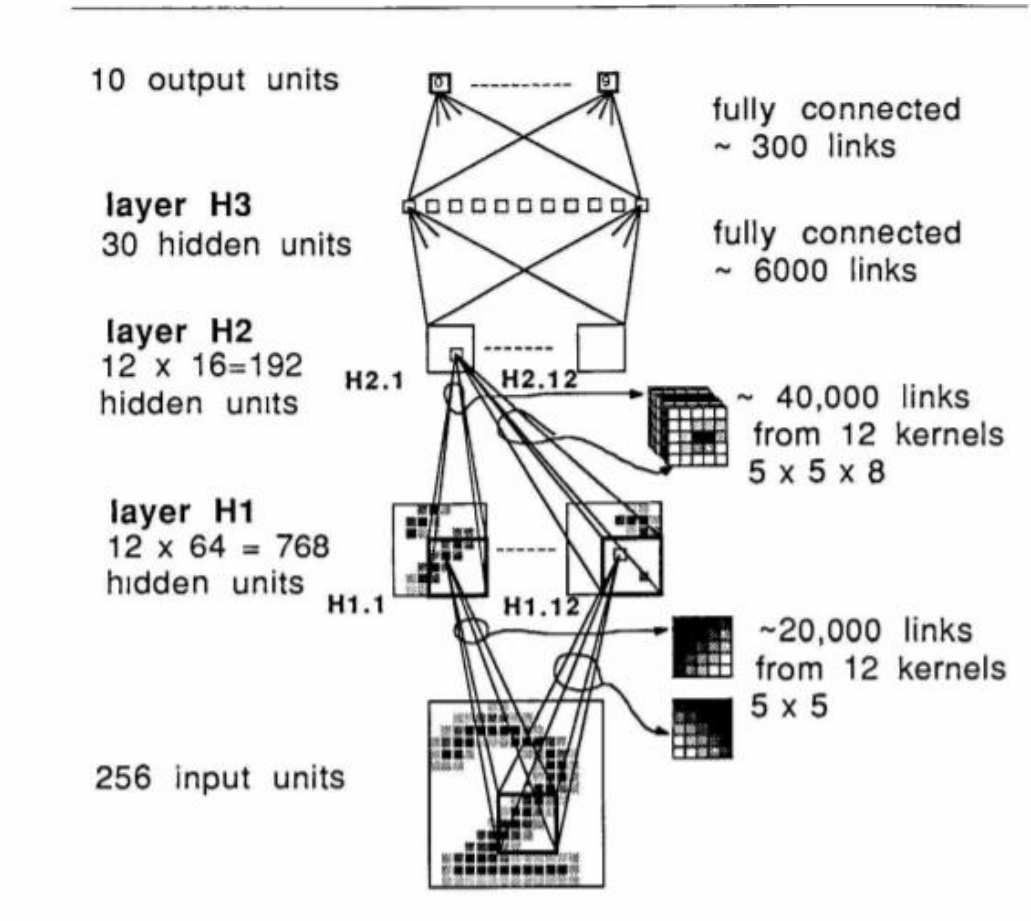


Figura 2. Primeira rede neural a usar retro propagação *(LeCun, 1989)*.

A *Figura 2* representa um esquema de como a rede funcionava. Primeiramente a imagem era inserida na camada de entrada e seus pixels eram separados em um vetor com as intensidades de cada pixel. Após isso, os dados passaram por três camadas ocultas e para enfim chegar a camada de saída, que possui 10 neurônios, um para cada algarismo numérico. A rede neural encontrou uma maneira de identificar os padrões dos números através da ativação de neurônios específicos, fazendo com que cada neurônio tivesse um peso que seria ativado de maneira muito mais intensa de acordo com o número mostrado.

Existem diversas formas onde a inteligência artificial se encaixa na grande área da engenharia, desde sua utilização para substituir tarefas rotineiras, até a análise de grandes conjuntos de dados (Ribeiro, 2020).

Uma grande área de pesquisa tanto da engenharia aeronáutica quanto de outras engenharias é a otimização de estruturas. Falando mais especificamente da otimização

dimensional de estruturas, a mesma já possui métodos conhecidos como o método dos elementos finitos (MEF) que será também abordado nesse projeto, juntamente com técnicas de inteligência artificial e DL para conectar o problema de otimização dimensional de uma estrutura utilizando redes neurais (Lopez, 2013).

Falando mais especificamente de *DL*, como tratamos de grandes quantidades de dados para treinar redes que identificam padrões, desde que tenhamos um conjunto de dados suficientemente grande podemos implementar uma rede em *DL* para qualquer problema da engenharia. Exemplos de aplicações são a análise de fluxos de voos e tráfego aéreo para criação de rotas otimizadas de forma automática, utilização de redes que realizam reconhecimento em estruturas civis e identificam possíveis falhas, existem muitos exemplos que podem ser citados quando se entende que DL é uma ferramenta que pode ser aplicada em qualquer área (IBEC, 2018).

Sabendo disso, o presente projeto configurou uma rede neural utilizando aprendizado profundo para encontrar a solução do problema de uma estrutura reticulada aeronáutica.

1.3. MÉTODO DOS ELEMENTOS FINITOS

O Método dos Elementos Finitos (MEF) é um procedimento numérico utilizado para obter soluções aproximadas de problemas de valores de equações diferenciais. Ele subdivide o domínio de um problema em partes menores, denominadas elementos finitos, por isso seu nome.

Para problemas com geometria, carregamento e condições de apoio simples, comumente se utilizam métodos clássicos que retornam soluções exatas, porém quando se tem estruturas complexas, é necessária uma solução aproximada e assim, a utilização do MEF (Nicoletti, 2018).

1.4. OBJETIVO GERAL

O objetivo desse trabalho foi a pesquisa na área de estruturas aeronáuticas reticuladas, através da análise da influência de cargas pontualmente aplicadas utilizando algoritmos em *DP* bem como o método dos elementos finitos para construir uma rede neural que encontre a solução do problema.

1.5. OBJETIVOS ESPECÍFICOS

Para obtenção desse objetivo geral, os seguintes objetivos específicos foram traçados:

- Implementação do método dos elementos finitos para estruturas reticuladas para criação de base de dados
- Estudar a implementação de redes de DL junto a bibliotecas Python: Tensor Flow, *Keras* e *scikit-learn*
- Estudo da Utilização de Algoritmos de DL junto ao método dos Elementos Finitos para análise de estruturas aeronáuticas reticuladas

2. REVISÃO BIBLIOGRÁFICA E FUNDAMENTAÇÃO TEÓRICA

A fundamentação teórica será feita através das subseções abaixo, focando em redes neurais do tipo *DL* e o método dos elementos finitos para estruturas reticuladas.

2.1. FUNCIONAMENTO DE UMA REDE NEURAL VIA *DL*

Uma rede neural do tipo DL é baseada em princípios heurísticos e/ou biológicos e vem do funcionamento de neurônios, suas conexões e seu agrupamento. A definição de neurônio, camadas, e suas conexões e relações é explicada nas seções abaixo, bem como os conceitos de treinamento, ativação e otimização.

2.1.1. Neurônio

O Neurônio biológico: a unidade mais básica do cérebro humano é o neurônio, ele é responsável pela transmissão de informações. Existem bilhões de neurônios e centenas de bilhões de conexões entre eles cérebro humano; quando a informação é passada através dos nervos terminais para neurônios vizinhos, através de um pulso elétrico, temos então a chamada sinapse, como mostra a Figura 3 (Data Science Academy, 2022).

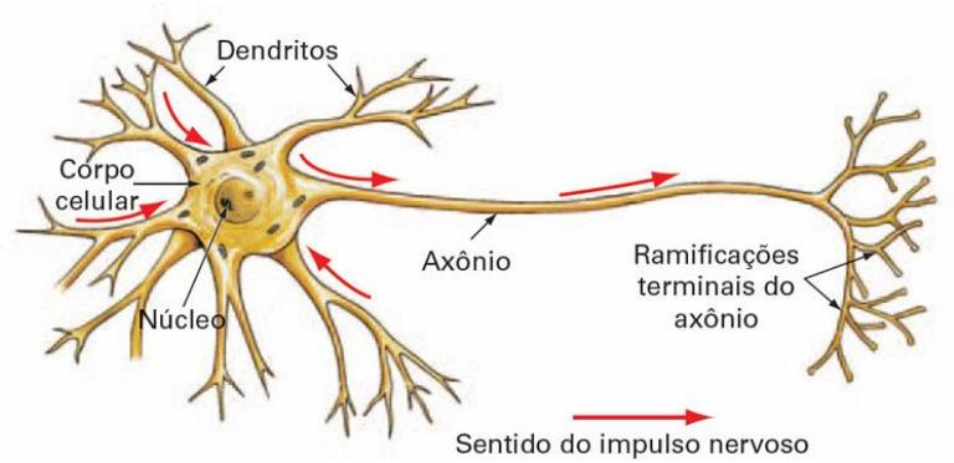


Figura 3. Representação Simplificada do Neurônio Biológico *(Data Science Academy, 2020)*

O neurônio computacional: buscando entender o funcionamento do neurônio biológico e realizar sua implementação computacionalmente, pesquisas começaram a surgir sendo o primeiro modelo mais bem aceito proposto por Warren McCulloch e Walter Pitts em 1943. A Sinapse dos neurônios matemáticos ocorre quando um neurônio da rede passa a informação para os neurônios da próxima camada através de um processo de multiplicação e soma ponderada das entradas multiplicadas por pesos e vieses, que são os valores que serão atualizados para a convergência da rede (Mcculloch, 1943).

Explicando de forma mais detalhada, os impulsos elétricos seriam os sinais de entrada, que no caso da rede neural artificial, são representadas pelas entradas (*inputs*). Uma rede pode variar tanto o número de camadas, quanto a quantidade de neurônios que cada camada possui. Cada neurônio é responsável por armazenar um valor numérico, pode ser inteiro ou real (geralmente armazenado via ponto flutuante), negativo ou positivo dependendo de cada caso.

O neurônio computacional recebe um ou mais sinais de entrada e devolve um único sinal de saída, que pode ser distribuído como sinal de saída da rede, ou como sinal de entrada para um ou vários outros neurônios da camada posterior (Figura 4). Depois que a rede completa sua primeira iteração, que será explicado nas seções a seguir, os pesos de cada camada são atualizados e os neurônios passam a representar novos valores. Esse processo é repetido várias e várias vezes até que os neurônios da camada de saída estejam dentro de uma faixa de valores com um erro pré-determinado aceitável. Dessa forma, pode-se entender o neurônio computacional como uma entidade que transforma diversas entradas através de alguma função dada e as combina esses resultados através de uma combinação linear. (Data Science Academy, 2020)

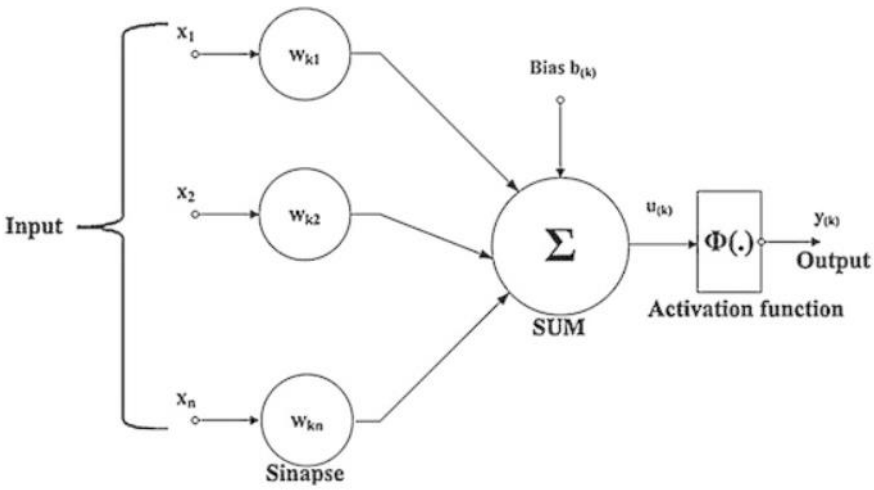


Figura 4. Representação Simplificada do Neurônio Matemático *(Data Science Academy, 2020)*

2.1.2. Camadas de entrada, ocultas e de saída

A camada de entrada (*input layer*) é a primeira camada de uma rede neural, ela representa o domínio dos valores de variáveis independentes do nosso problema, ou seja, para essa camada, o número de neurônios será igual ao número de variáveis do problema. Os valores que serão inseridos na primeira camada não são multiplicados anteriormente por qualquer peso ou adicionado a qualquer viés. Além do papel de alimentação de toda a rede, o número de neurônios da camada de entrada é um parâmetro extremamente importante. Quanto mais complexo um problema é, mais variáveis ele possuirá, dessa forma, quanto mais neurônios na camada de entrada uma rede em *DL* tiver, maior será o custo computacional associado. Por conta disso, existe uma busca de minimizar sempre que possível o número de neurônios da camada de entrada, isso é, tentar representar um problema com o mínimo de variáveis possíveis, sem sacrificar qualquer parâmetro que seja fundamental para o aprendizado da rede (Chollet, 2018).

Após a camada de entrada, podemos ter n camadas ocultas (*hidden layers*), porém isso não significa que quanto mais complexo o problema for, maior o número de camadas ocultas ou de neuronio por camadas a arquitetura da rede deva possuir. Aumentar o número de camadas ocultas pode refinar o resultado obtido, porém deve-se fazer um balanceamento adequado, pois, caso seja utilizado um número exagerado de camadas ocultas, o treinamento da rede pode exigir um custo computacional muito alto ou divergir, devido ao aumento exponencial da complexidade da função final (Chollet, 2018).

Por fim, temos a camada de saída (*output layer*) que representa o domínio dos valores de variáveis dependentes do nosso problema. Para se treinar uma rede em *DL* precisamos de ambos os dados de entrada e de saída do caso a ser analisado, pois é apenas comparando os resultados de cada iteração com os resultados reais de uma dada realização da população na camada de saída que a rede convergirá para os pesos e vieses que solucionam o problema (Chollet, 2018).

2.1.3. Pesos e vieses

Como dito anteriormente, uma rede neural é composta por elementos que formam sua arquitetura e por parâmetros que influenciam sua eficiência e funcionabilidade. Aqui definir a arquitetura como sendo o número de camadas, o número de neurônios, as funções de transformação de cada neurônio, os tipos de dados etc.

Assim que a arquitetura da rede é escolhida, ela pode ser modificada e alterada ao longo de um projeto ou estudo, porém para um dado treinamento, não ocorre uma alteração nos valores de entrada ou saída, no número de neurônios ou camadas ou até mesmo no tipo de função utilizada. Porém como nenhum desses parâmetros se altera ao longo das iterações, o que faz com que a rede aprenda ou convirja para o resultado desejado são os pesos e vieses (Calôba, 2002).

Dentro de uma rede neural há uma camada de entrada (do inglês *input layer,* em tradução livre), que obtém os sinais de entrada, os transforma, e os passa para a próxima camada. Em seguida, a rede neural contém uma série de camadas ocultas (*hidden layers*) que aplicam transformações nos dados de entrada. É dentro dos neurônios das camadas ocultas que os pesos (*weights*) são aplicados. Desta forma, a rede começa multiplicando os valores de entrada inseridos por um peso pré-determinado ou aleatório e em seguida soma a um viés (*bias*) (Figura 5). O viés é utilizado para aumentar os graus de liberdade dos ajustes dos pesos. Dessa maneira, cada neurônio é conectado a todos os neurônios da próxima camada, até chegar na camada de saída, como mostra o esquema a seguir.

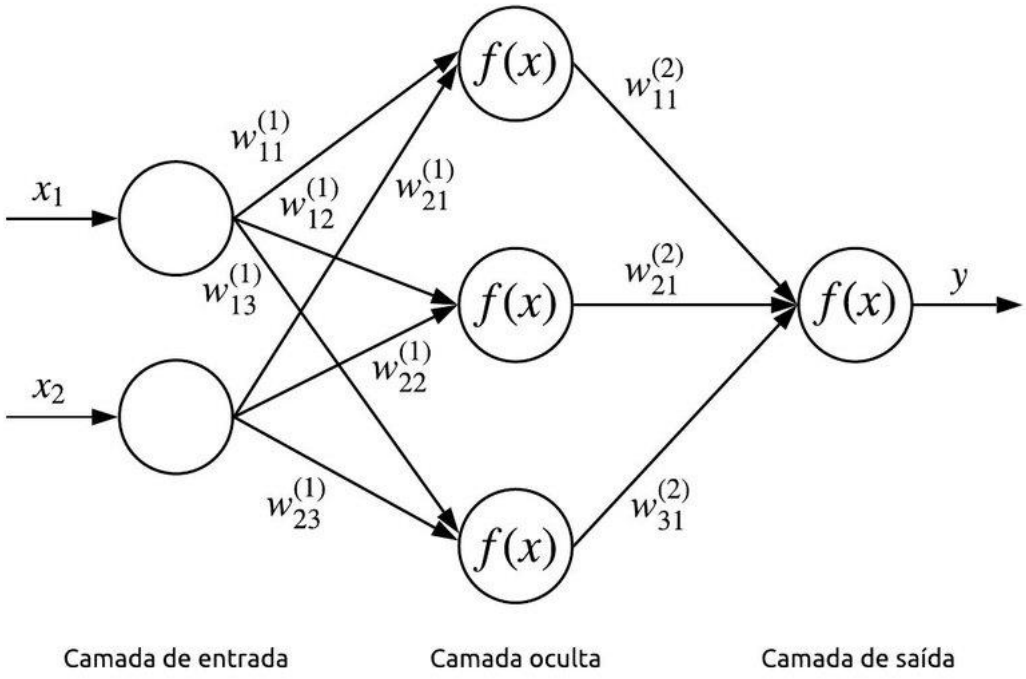


Figura 5. Esquema de uma rede com arquitetura 2-3-1 *(Calôba, 2002)*

Por exemplo, um único nó pode captar os dados de entrada e multiplicá-los por um valor de peso atribuído, em seguida, adicionar um viés antes de passar os dados para a próxima camada. A camada final da rede neural também é conhecida como a camada de saída (do inglês *output layer*, em tradução livre).

2.1.4. Camada de ativação

As camadas de ativação são um conjunto de uma ou mais camadas ocultas pertencentes a rede cujos neurônios possuem uma função de ativação. A finalidade da inserção de uma função de ativação na rede tem dois principais motivos:

- De-linearização: Para entender esse fenômeno, devemos entender a matemática por trás do funcionamento de uma rede neural (Sharma, 2017). Temos que a saída y de uma rede com duas camadas ocultas é dada por:

$$y = \phi(\phi(XW_1)W_2)\omega \quad (1)$$

onde

$y = saída\ da\ rede$
$\phi = função\ de\ ativação$
$X = matriz\ de\ dados\ de\ entrada$
$W_n = pesos\ das\ camadas\ ocultas$
$\omega = pesos\ da\ camada\ de\ saída$

A equação acima possui os valores dos vieses omitidos sem perda de generalidade. Sabendo disso, podemos retirar ϕ da equação acima e reescrever o produto dos pesos como u:

$$y = (XW_1)W_2\omega = Xu \quad (2)$$

Adotando uma arquitetura de rede onde nenhuma camada possui neurônios com funções de ativação, chegamos em um modelo de regressão linear. Isso não significa que a rede não funciona ou esteja errada, porém esse modelo é adotado para problemas mais simples, onde não exista a necessidade de operações de de-linearização. Dito isso, a inserção de funções de ativação é de extrema importância pois faz com que a rede se comporte de maneira não-linear, podendo capturar efeitos não-lineares (Hutter, 2014).

- Suavização dos efeitos da atualização dos pesos: Com a atualização dos pesos e vieses após cada iteração da rede, cada ligação entre os neurônios tem seu valor atualizado para fazer a rede convergir para o resultado. Além do problema de linearização ao não implementar uma função de ativação, outro problema que pode ocorrer é a complexibilidade computacional exigida para encontrar o "peso ideal" para a ligação entre os neurônios. Isso ocorre pois conforme o treinamento da rede progride, a alteração de um dado peso pode fazer com que a saída desse neurônio mude completamente, sendo necessário um ajusto extremamente preciso para não modificar o resto da rede, pois a saída de uma camada oculta será a entrada da próxima camada. Com a implementação das funções de ativação, ocorre uma suavização nesse efeito, ou seja, pequenas mudanças nos pesos e vieses causam uma pequena alteração na saída, fazendo com que um neurônio seja ativado ou não em certa conexão (Hutter, 2014).

Na matemática, as funções e suas derivadas podem ser contínuas ou descontínuas. São ditas contínuas as funções cujo $\lim_{x \to a} f(x) = f(a)$ em todos os pontos de seu domínio, ou seja, para cada ponto de seu domínio existe um ponto correspondente em sua imagem. Caso a função tenha um ou mais pontos de seu domínio que não possua uma imagem, é dito que a função possui um ponto de descontinuidade, ou seja, essa função é descontínua.

As funções polinomiais, racionais, trigonométricas, exponenciais e logarítmicas, estão dentro da família de funções contínuas, e assim como elas, as funções estudadas e utilizadas nesse projeto também são contínuas.

Existem diversos tipos de funções de ativação e, com os avanços nas pesquisas, cada vez mais novas funções mais eficientes e com propostas e características diferentes surgem. A seguir serão apresentadas algumas funções que são comumente usadas e foram estudadas para a implementação da rede desse projeto.

A linguagem utilizada para programação da rede será Python, e a biblioteca que permitirá a criação de diferentes arquiteturas será a Keras, incorporando funções das bibliotecas TensorFlow e SKLearnkit, portanto foram escolhidas três funções de ativação dentro do repertório disponível pela biblioteca utilizada.

I.Sigmoide

A função sigmoide é dada por:

$$\sigma(x) = \frac{1}{1 + e^{x}} \tag{3}$$

Já sua derivada é

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{4}$$

Ela tem aplicação não só no ramo da computação, mas também de outras áreas como economia. Essa função tem uma maior aplicabilidade em casos onde se espera uma saída binária, ou seja, quando temos um problema de classificação binária e não de regressão. Seu uso nesse tipo de problema ocorre pois ela assume apenas valores entre 0 e 1 (Figura 6), trazendo um resultado analogo ao funcionamento de um neuronio biológico com a não ativação (zero) ou ativação (um) (Sharma, 2017).

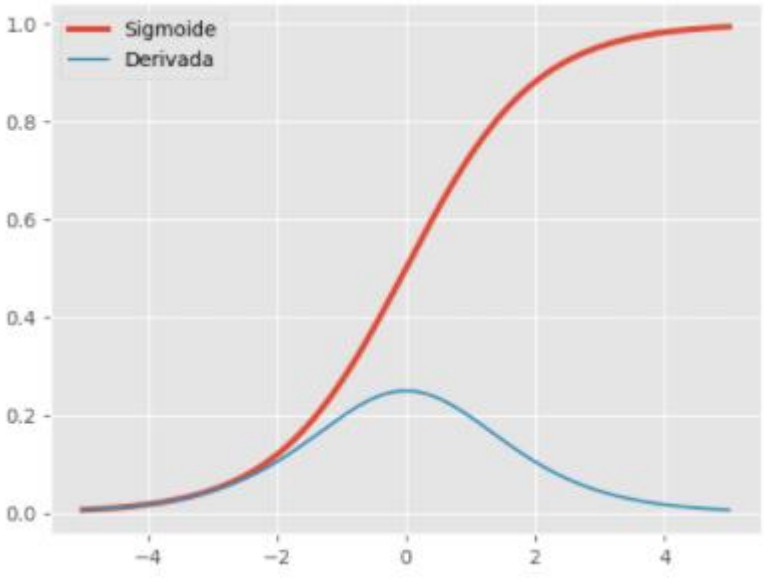


Figura 6. Gráfico da função Sigmoid e sua derivada *(Facure, 2017)*

Observando o comportamento da função sigmoid, podemos ver que sua curva forma um S, dando assim origem ao nome Sigmoid. Já se tratando de sua derivada, podemos observar que ocorre uma saturação para valores abaixo de -5 e acima de 5, fazendo com que exista uma dificuldade de treinamento para problemas de regressão .

II.Tangente hiperbólica

A função tangente hiperbólica é dada por:

$$\tanh(x) = 2\sigma(2x) - 1 \tag{5}$$

Já sua derivada é:

$$tanh'(x) = 1 - \tanh^2(x) \tag{6}$$

A função tangente hiperbólica possui características semelhantes a função sigmoide, pois também possui um formato em S, porém seus valores variam de -1 a 1 (Figura 7). Sua aplicação é útil na área das engenharias também, como por exemplo cálculo de comprimento e tensão de cabos para transportes elétricos ou no setor aeroespacial para o cálculo de revestimentos de superfícies ideais para aeronaves (Sharma, 2017).

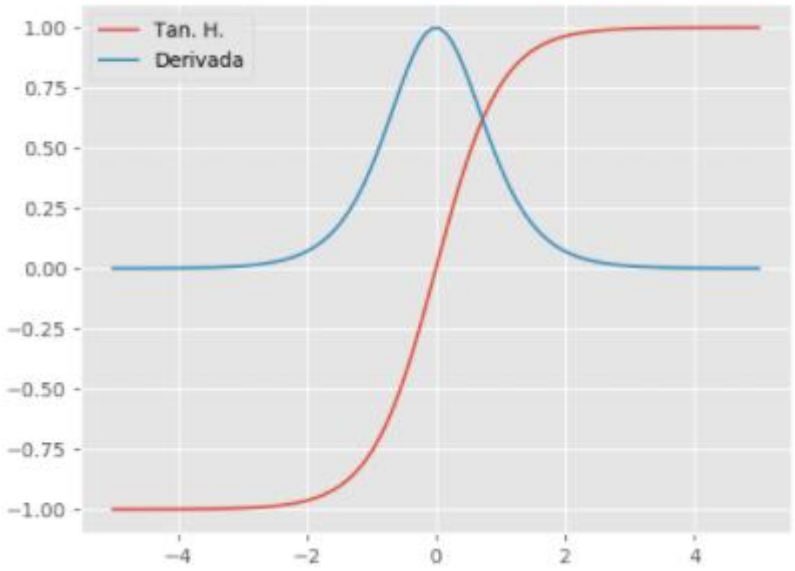


Figura 7. Gráfico da função tangente hiperbólica e sua derivada (Facure, 2017)

Como pode-se observar, a derivada da função hiperbólica também possui saturação para valores abaixo de -2 e acima de 2.

III.ReLU

Finalmente temos a função ReLU (Rectified Linear Unit), que é dada por:

$$\mathrm{ReLU(x)} = \max\{0, \mathrm{x}\} \tag{7}$$

Já sua derivada é:

$$ReLU'(x) = \begin{Bmatrix} 1, se\ x \geq 0 \\ 0, se\ x < 0 \end{Bmatrix} \tag{8}$$

A ReLU é muito utilizada em redes neurais atualmente. Umas das principais vantagens dessa função se deve ao fato dela não ativar todos os neurônios simultaneamente, ou seja, como ela resulta em 0 para valores negativos, apenas os neurônios com valores positivos serão ativados, fazendo com que cada neurônio fique ”treinado” para identificar algum padrão específico e também aumentar a eficiência computacional (Figura 8) (Brownlee, 2019a).

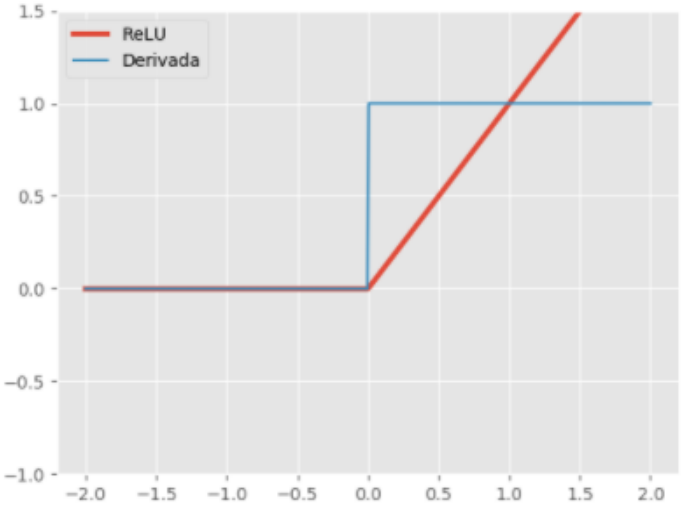


Figura 8. Gráfico da função ReLU e sua derivada *(Facure, 2017)*

Uma desvantagem que essa função possui é o deslocamento dos gradientes em direção ao zero durante o treinamento fazendo com que os pesos deixem de ser atualizados dependendo dos processos de otimização que são utilizados, porém existem algoritmos que corrigem esse problema.

2.1.5. Função de custo

A função de perda ou função de custo (*loss function*) é utilizada em uma rede neural para poder avaliar uma possível solução, ou seja, a eficiência de um conjunto de pesos e vieses. Portanto, busca-se sempre a minimização da função custo negativada quando se trata de problemas de maximização (Nishimoto, 2018).

Para toda rede que será treinada, é necessária uma função de custo, porém não existe apenas uma opção de função disponível para escolha, mas sim várias onde cada uma possui vantagens e desvantagens dependendo do tipo de problema que está sendo analisado (Data Science Academy, 2021).

Outro fator que deve ser levado em conta é a relação entre a função de ativação da camada de saída e a função de custo que será utilizada, pois ambos os elementos da arquitetura estão conectados (Brownlee, 2019b).

Para o presente projeto, Foram estudados três tipos de problemas que podem ser tratados utilizando ML e DL, são eles:

a) **Problema de regressão:** Um problema em que você deseja prever uma quantidade de valor real.

Configuração da camada de saída: Um ou mais nós com uma unidade de ativação linear.

Função de Custo: Erro Quadrado Médio (MSE). Seu resultado é sempre positivo independentemente do sinal dos valores previstos e reais e um valor perfeito é 0,0. O valor da perda é sempre minimizado e ela é utilizada em problemas de regressão pois uma vez que essa métrica eleva o erro ao quadrado, predições muito distantes do real aumentam o valor da medida muito facilmente, fazendo com que os pesos e vieses sejam atualizados mais bruscamente quando o erro é muito alto (Gomes, 2019).

b) **Problema de classificação binária:** Um problema em que você deseja classificar um exemplo como pertencente a uma entre duas classes.

Configuração da camada de saída: Um nó com uma unidade de ativação sigmoide.

Função de perda: Entropia cruzada, também conhecida como perda logarítmica. Num problema de classificação, busca-se dividir as saídas em classes (ou grupos), onde cada grupo tem sua distribuição probabilística. No caso de um problema de classificação binária, é comum exprimirmos as classes com os valores 0 ou 1, sendo assim, a função de entropia cruzada nos retorna a probabilidade da saída pertencer ao grupo 0 ou ao grupo 1.

Função de perda: Entropia cruzada, também conhecida como perda logarítmica.

Como o modelo aprende a retornar um valor para cada classe, nesse caso é comum aplicarmos a função Softmax antes de calcular o custo. Essa função transforma todos os valores para o intervalo [0, 1] ao mesmo tempo em que garante que a soma dos resultados seja igual a 1, ou seja, a soma das probabilidades de uma instância pertencer a qualquer classe deve ser igual a 100%, o que indica que ela certamente pertence a alguma das classes (Browniee, 2020).

c) **Problema de classificação multi-classe:** Um problema em que você classifica um exemplo como pertencente a uma de mais de duas classes.

Configuração da camada de saída: Um nó para cada classe usando a função de ativação Softmax.

Função de perda: Entropia cruzada.

2.1.6. Pontuação R2

Além da função de custo ser utilizada para a otimização da rede, como será explicado na próxima seção, ela também tem um importante papel de servir como métrica para calcular o

desempenho da rede. Porém a função de custo não é a única métrica possível que pode ser utilizada (Wikipédia, 2021).

Outra pontuação muito utilizada para medir a performance de uma rede é a pontuação R2, ela funciona medindo a quantidade de variação nas previsões pelo conjunto de dados, ou seja, a diferença entre as amostras de dados fornecidas e as previsões do modelo e pode ser diretamente adaptada à época atual de treinamento de uma rede neural (Wikipédia, 2021).

A pontuação R2 foi utilizada como forma de medida de performance nesse modelo, portanto é importante a explicação da interpretação de seu resultado. Ela é calculada através da fórmula a seguir.

$$R^2 = 1 - \frac{SQ_{res}}{SQ_{tot}} \tag{9}$$

Onde

$$SQ_{res} = \sum_{i=1}^{n} (y_i - \widehat{y_i})^2 \ , \qquad \widehat{y_i} = valor\ estimado \tag{10}$$

$$SQ_{tot} = \sum_{i=1}^{n} (y_i - \overline{y_i})^2 \ , \qquad \overline{y_i} = média\ das\ observações \tag{11}$$

O valor obtido sempre estará entre 0 e 1, quanto mais próximo de 0 for a pontuação, pior será o desempenho para um conjunto de dados ainda não visto, já quanto mais próximo de 1, mais perfeitamente o modelo será treinado (Wikipédia, 2021).

2.1.7. Camada de otimização

Assim que a função de custo for definida para seu modelo, é necessário escolher um algoritmo que fará a otimização para minimizar a função de custo. A função de custo que será otimizada é chamada comumente chamada de função objetivo. O tipo mais comum de problemas de otimização encontrados no aprendizado de máquina são a otimização de uma função contínua .

Para o problema que será analisado, foi feito o estudo e análise da implementação de dois algoritmos de otimização muito utilizados em *DL*, o Gradiente Descendente Estocástico (*Stochastic Gradient Descent*) e o ADAM (*Adaptive Moment Estimation*).

I. Gradiente Descendente Estocástico:

O Gradiente Descendente Estocástico é baseado nos algoritmos descendentes clássicos de otimização não linear. Os algoritmos descendentes clássicos possuem características X, Y e Z que fazem com que sua aplicabilidade a redes neurais seja baixa, sendo pouco utilizado. O Gradiente Descendente Estocástico melhora essas características de modo a ser mais aplicável.

Dado um conjunto de dados de treinamento de n entradas, onde $f_i(x)$ é a função de custo para o conjunto de treinamento i e x é o vetor dos dados de entrada temos

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{12}$$

A equação acima basicamente significa que a função objetivo é uma média das funções de custo para cada amostra do conjunto.

O gradiente da função objetivo é dado por

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x) \tag{13}$$

O gradiente descendente estocástico (SGD) reduz o custo computacional em cada iteração comparado ao seu antecessor, pois em cada iteração da descida do gradiente estocástico, é calculado utilizando um subconjunto da população e esse é estatisticamente próximo do gradiente real para a aplicação na camada de otimização (Bottou, 2018).

Quando há mais exemplos no conjunto de dados de treinamento, custa mais computacionalmente do que o gradiente descendente, portanto, o Gradiente Descendente Estocástico é o preferido nesses casos. Porém, problemas podem ocorrer quando utilizamos SGD como algoritmo de otimização (Figura 10), pois ele é muito sensível ás mudanças na taxa de aprendizado (que será explicada na próxima seção), fazendo com que a otimização acabe ficando com maiores ruídos e dificuldades ao longo das iterações, comparada com outros algoritmos como até mesmo o algoritmo descendente clássico Figura 9 (Zhang, 2020).

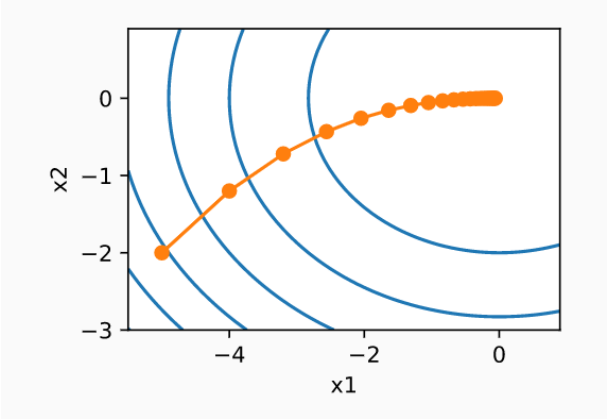


Figura 9. Convergência do Gradiente Descendente *(Dive into Deep Learning, 2020)*.

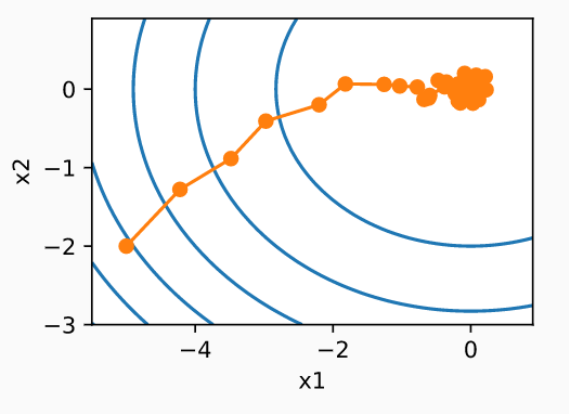


Figura 10. Convergência do Gradiente Descendente Estocástico *(Dive into Deep Learning, 2020)*.

II. ADAM

O algoritmo Adam combina várias técnicas em um algoritmo de aprendizado eficiente, por conta disso é o ou um dos algoritmos mais utilizados atualmente para aprendizado profundo.

Uma das principais características desse algoritmo está no fato dele utilizar médias móveis ponderadas exponenciais para determinar uma estimativa do primeiro e segundo momento do gradiente (Bottou, 2018).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (14)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (15)$$

onde $m_t \; e \; v_t$ são estimativas do primeiro momento e segundo momento dos gradientes, respectivamente.

Por padrão do *KERAS*, a biblioteca utilizada, as taxas de decaimento exponenciais do primeiro e segundo momento são, respectivamente, $\beta 1 = 0.9 \; e \; \beta 2 = 0.999$, e o algoritmo inicia com $s_0 = v_0 = 0$, fazendo com que a estimativa de variância se move muito mais lentamente do que o termo de momento. Um dos efeitos da maneira como o algoritmo ADAM

é implementado e também faz com que seja amplamente utilizado é a forma com que ele atualiza os pesos e viesses (Ruder, 2016).

Esse algoritmo faz com que dependendo da resposta de cada saída da rede, cada peso de sinapse seja alterado individualmente, ou seja, os valores de peso de uma determinada camada não são atualizados da mesma forma, e sim separadamente. Isso faz com que haja uma conversão mais rápida para a resposta desejada, porém, para gradientes com variância significativa, podemos encontrar problemas de convergência.

2.1.8. Hiper parâmetros

Os chamados hiper parâmetros são parâmetros de uma rede que são utilizados pra controlar o processo de aprendizado do modelo. O prefixo *"hiper"* indica que estão em *"um nível superior"* e controlam o processo e os outros parâmetros do modelo. Eles são definidos antes mesmo do início da aprendizagem, e não costumam ser alterados ao longo de todo processo, ao contrário do que acontece com os pesos e viesses, a não ser em casos específicos como a variação da taxa de aprendizado dinâmica que foi citada acima (Chollet, 2018).

Dito isso, podemos definir como hiper parâmetros de um modelo todos os parâmetros que são definidos antes do início do treinamento cujos valores ou configurações continuarão os mesmos quando o processo de aprendizado acabar. Inicialmente não existe uma fórmula ou método que dará os melhores hiper parâmetros, porém existem técnicas que podem ser utilizadas para obter uma estimativa inicial, como por exemplo realizar uma análise estatística dos dados da população inicial.

Uma rede de aprendizado profundo possui diversos hiper-parâmetros. Essa seção citará alguns deles, bem como suas contribuições para a rede (Hutter, 2014).

I. Taxa de aprendizado (do inglês *learning rate, em tradução livre*)

A taxa de aprendizado é um parâmetro de ajuste em um algoritmo de otimização que determina o tamanho do passo em cada iteração enquanto se move em direção a um mínimo de uma função de custo. Ela desempenha um importante papel dentro de uma rede, em suma, ela define a velocidade com que o seu modelo irá aprender. Porém escolher uma boa taxa de aprendizado não é tão simples, em caso da escolha de uma taxa muito alta, oscilações no treinamento podem impedir a convergência do processo de aprendizado, pois como o algoritmo

de otimização sempre buscam o mínimo global, um passo muito grande pode fazer com que o algoritmo simplesmente fique perdido durante o treinamento.

No caso da escolha de uma taxa muito pequena, começamos a ter o problema de *overfitting*, que ocorre quando um modelo estatístico se ajusta muito bem ao conjunto de dados anteriormente observado, mas se mostra ineficaz para prever novos resultados.

Dito isso, a seleção da taxa de aprendizado não é algo que é perfeitamente escolhido na primeira tentativa de treinamento do modelo, geralmente tendo de ser atualizada através de "tentativa e erro" ou de algoritmos automáticos do tipo previsor-corretor. Outro fator que influencia a eficiência da taxa de aprendizado e o tempo que o modelo levara para aprender ou se ele conseguirá aprender é a função de otimização utilizada. No caso da função escolhida para o modelo do projeto (Adam), as taxas de aprendizado que obtém uma boa performance variam entre 10^{-4} a 10^{-2} (Mack, 2018).

II. Número de camadas

Toda rede precisa ter ao menos duas camadas, a camada de saída e entrada, porém quando tratamos de problemas mais complexos, em especial modelos com comportamento não-linear, e principalmente quando falamos de DL, estamos falando da adição de outras camadas entre essas principais, as camadas ocultas.

Ainda não existe uma fórmula que define perfeitamente o número de camadas ocultas que seu modelo precisa para uma boa eficiência, sendo ainda um problema em aberto (Krishnan, 2021). Embora existam algoritmos de meta-aprendizagem de máquina, em geral eles resolvem o problema através de procedimentos *brute force*. Dessa forma, esse trabalho é feito através de algumas estimativas, juntamente com a repetição do treinamento do seu modelo fazendo testes para obter uma eficiência satisfatória.

É intuitivo que a adição de camadas ocultas exige mais tempo e poder computacional, porém adicionar camada atrás de camada em uma rede não é uma boa prática a ser adotada, pois isso fará o modelo se tornar extremamente complexo, fazendo com que o aprendizado se torne muito demorado ou até mesmo tornando a rede impossível de encontrar uma solução ou a rede ser necessariamente divergente. Para a maioria dos problemas, um número entre 1 e 3 camadas ocultas é o suficiente. A adição de mais camadas não costuma afetar a eficiência da rede na maior parte dos casos e só trará pontos negativos para o seu modelo (Krishnan, 2021).

III. Número de neurônios por camada.

Para toda rede, o número de neurônios da camada de saída e de entrada já é sabido, porém o número de neurônios das camadas ocultas é alvo de discussão atualmente. Como no exemplo anterior, o número de neurônios de uma camada é algo que deve ser estudada e testado para cada modelo, porém uma regra que comumente é utilizada e pode ser uma boa estimativa inicial é dada por (Bengio, 2012)

$$N_n = \frac{N_s}{\left(\alpha(N_i + N_o)\right)} \tag{16}$$

$N_n = numero\ de\ neurônios\ da\ camada\ oculta$
$N_o = número\ de\ neurônios\ da\ camada\ de\ saída$
$N_i = número\ de\ neuronios\ da\ camada\ de\ entrada$
$\alpha = Parãmetro\ arbitrário\ entre\ 5\ e\ 10$

IV. Número de épocas

Uma época é um termo utilizado para indicar o número de passagens de todo o conjunto de dados de treinamento que o algoritmo de aprendizado de máquina concluiu. Novamente, não existe um número perfeito pré-determinado para o número de épocas que seu modelo deva possuir. O número de épocas geralmente é ajustado após a visualização de alguns treinamentos, após uma análise do gráfico de aprendizado e/ou convergência, pode-se optar por diminuir o número de épocas caso seja aparente que a rede já havia obtido um bom resultado com um número menor de épocas, ou aumentar caso a curva de aprendizado não tenha estabilizado (Bengio, 2012).

Também existem outros hiper-parâmetros em uma rede neural que não serão aprofundados ou que já foram previamente discutidos, como por exemplo a função de otimização.

2.2. ALGORITMOS GENÉTICOS

A utilização de um algoritmo genético nesse trabalho teve um papel crucial para a otimização das áreas que foram utilizadas no treinamento da rede final. Para isso, a biblioteca *Scypy* foi utilizada.

A função *scipy.optimize.differential_evolution* foi escolhida para a otimização das áreas pois ela não usa os métodos de gradiente, ou seja, é de natureza estocástica, fazendo com que seja mais eficiente para encontrar mínimos globais, ao mesmo passo que requer um número maior de avaliações de funções do que as técnicas convencionais baseadas em gradiente (Scipy, 2022).

Essa função possui dois parâmetros obrigatórios que são a função que será otimizada, e o intervalo onde ela será otimizada, que em suma significa o intervalo no qual o algoritmo tentará minimizar a função.

Além disso, outros parâmetros opcionais podem ser adicionados de acordo com a aplicação desejada. No caso desse projeto, também foram utilizados parâmetros para um número máximo de iterações, uma tolerância, a população inicial e a taxa de mutação. Todos os parâmetros acima já possuem um valor padrão da função, porém a variação desses parâmetros ajudou a uma convergência mais rápida e precisa.

Depois de passados os parâmetros para a *scipy.optimize.differential_evolution*, como dito ela irá utilizar uma evolução diferencial para encontrar uma otimização global. Com cada passagem pela população, o algoritmo gera uma solução candidata que é misturada com outras para assim gerar uma solução candidata teste. Na estratégia utilizada, dois membros da população são escolhidos aleatoriamente e sua diferença é usada para alterar o melhor membro (Storn, et al., 1997).

Outra estratégia utilizada nesse projeto foi a penalização manual dos candidatos que não atendem os requisitos passados, com isso o algoritmo consegue uma convergência mais precisa para os valores de áreas desejados.

Com a a criação do candidato teste, um vetor de teste é gerado, e assim a escolha de um novo candidato ocorre caso o candidato teste se mostre pior do que o novo candidato, nesse caso o candidato teste é substituído completamente pelo novo candidato.

Esse processo é realizado até a tolerância estipulada ser atingida, e repetindo esse processo repetidas vezes, um conjunto de dados para o treinamento da rede final foi criado.

2.3. MÉTODO DOS ELEMENTOS FINITOS

Para aplicação do MEF, a estrutura é dividida em um número finito de partes ou elementos, fazendo com que a estrutura completa seja formada a partir da união de várias estruturas simples, essas conexões são denominadas Nós do Modelo, e são neles que as forças

são aplicadas. Já os denominados Elementos do Modelo são todos os seguimentos de reta que ligam os nós e representam as barras da treliça (Nicoletti, 2018).

A malha do Modelo é a união de todas as subdivisões dos elementos, e sua eficiência depende diretamente do refinamento feito. O refinamento da malha define como será a subdivisões dos elementos (GIL, 2015).

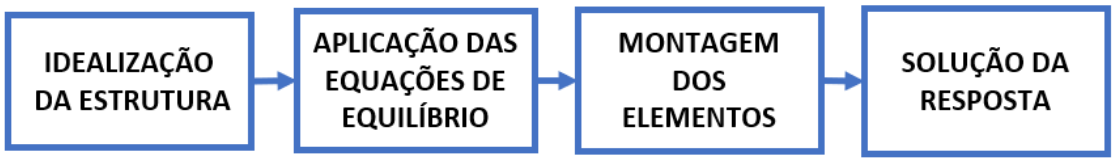


Figura 11. Método comumente empregado para análise de sistemas discretos. (elaborado pelo autor)

Após a idealização da estrutura, cada Elemento do Modelo está em equilíbrio.

A equação de equilíbrio geral que utiliza a abordagem da energia potencial é dada por (Kwon, 2000)

$$\Pi = U_e - W$$

$$\Pi = \frac{1}{2}\iiint_V \{\epsilon\}^T [D]\{\epsilon\} dV - \iiint_V \{u\}^T \{R\} dV - \iint_A \{u\}^T \{\emptyset\}\, dA - \{u\}^T \{P\} \qquad (17)$$

$Energia\ de\ deformação$ — $Forças\ de\ superfície$ — $Forças\ de\ corpo$ — $Forças\ pontuais$

Após a aplicação da equação de equilíbrio, devem ser atendidos todos os requisitos de interconexões de elementos e dessa forma, realizar a montagem dos elementos que será explicada na próxima seção. Por fim, a solução da resposta é obtida computacionalmente através da resolução das equações para calcular os deslocamentos e tensões de cada Elemento e Nó (Nicoletti, 2018).

2.3.1. Elementos de treliça

Uma treliça consiste em elementos retos, que só possuem deformação na direção axial, unidos por nós que formam unidades triangulares. As forças resultantes nos vários elementos das estruturas são sempre de tração ou compressão e necessariamente aplicadas nos nós para

garantir o equilíbrio (Beer, 2011). Aqui inserir exemplos de estruturas treliçadas, e algumas figuras, ponte, teto de estádio, cauda de helicóptero, captadores solares de satélites.

No caso desse projeto, as estruturas treliçadas estudados são planares, ou seja, todos os membros e nós se encontram no mesmo plano.

Como já dito, esse projeto terá como objetivo uma análise da deformação e tensão máximos na estrutura treliçada proposta. A deformação e a tensão máxima de um elemento treliça são respectivamente

$$\varepsilon(x) = \frac{\Delta L}{L} \quad e \quad \sigma(x) = E\varepsilon(x) \text{ (Beer, 2011)} \tag{18}$$

$onde$

$\varepsilon = deformação$

$\Delta L = variação\ do\ comprimento\ da\ barra$

$L = comprimento\ final\ da\ barra$

$\sigma = tensão$

$E = módulo\ de\ elasticidade$

2.3.2. O Método da Rigidez Direta

O método da rigidez direta é oriundo do campo da aeronáutica, onde durante o período entre a Primeira Guerra Mundial (1914-1918) e a Segunda Guerra Mundial (1939-1945), grandes potências econômicas, como Estados Unidos, Inglaterra e Alemanha, realizavam experimentos buscando desenvolver aviões mais velozes e com menores coeficientes de empuxo. (FELLIPA, 2000).

A formulação algébrica mais compacta desse método levam em consideração o equilíbrio de cada nó, de forma que os deslocamentos dos nós da estrutura são as suas incógnitas

$$Ku = f \text{ (Beer, 2011)} \tag{19}$$

onde:

$K = Matriz\ de\ rigidez\ global$

$u = Vetor\ de\ deslocamentos\ nodais$

$f = Vetor\ de\ forças\ nodais$

Para a matriz de rigidez local de uma treliça plana, temos que seus graus de liberdade nodais é expresso como

$$\{d^e\} = \{u_1 \ \ v_1 \ \ u_2 \ \ v_2\}^T \tag{20}$$

Onde o sobrescrito e corresponde ao nível do elemento. Sendo assim a matriz de rigidez correspondente tem dimensão 4×4

$$[K^e] = \begin{bmatrix} k & 0 & -k & 0 \\ 0 & 0 & 0 & 0 \\ -k & 0 & k & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{21}$$

Onde

$$k = \frac{AE}{l} \tag{22}$$

$E = módulo\ de\ elasticidade\ so\ material$

$A = Área\ da\ seção\ transversal\ da\ barra$

$l = comprimento\ da\ barra$

As linhas e colunas zeradas na matriz acima estão relacionadas com o deslocamento transversal, e é nula caso o elemento da treliça tenha apenas força axial, porém muitas estruturas planas consistem em elementos que possuem diferentes orientações, portanto para poder modelar a matriz de rigidez relacionada a essas estruturas é necessário ter os graus de liberdades dos elementos expressos em termos de um eixo de coordenadas global (Kwon, 2000).

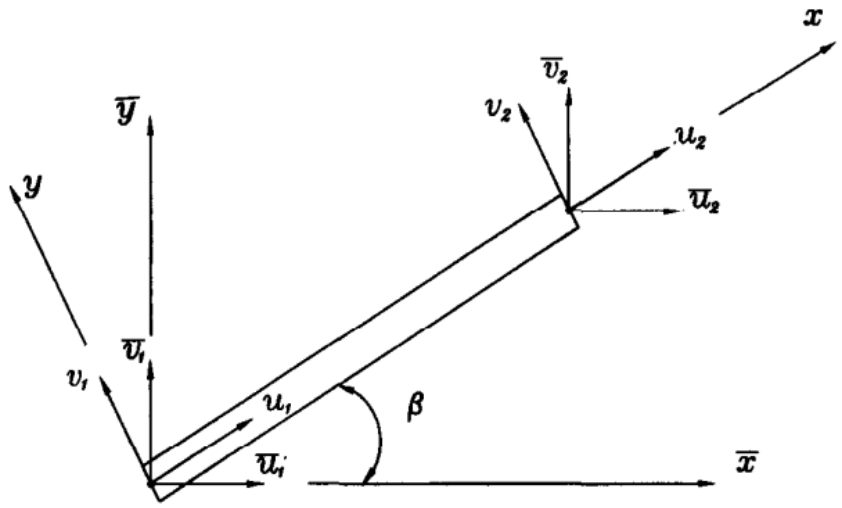


Figura 12. Elemento de treliça bidimensional generalizado *(Kwon, 2000)*.

A Figura 12 mostra um elemento de treliça orientado em um ângulo arbitrário β em relação ao eixo horizontal $\bar{x}$. Pode-se observar que existem dois deslocamentos (u, v) associados sistema de coordenadas do elemento (x, y) e dois deslocamentos $(\bar{u}, \bar{v})$ associados ao sistema de coordenadas global $(\bar{x}, \bar{y})$ (Kwon, 2000). A relação entre esses dois sistemas de coordenadas é chamada de *transformação de coordenadas* e é dada por

$$\begin{Bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{Bmatrix} = \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & c & s \\ 0 & 0 & -s & c \end{bmatrix} \begin{Bmatrix} \bar{u}_1 \\ \bar{v}_1 \\ \bar{u}_2 \\ \bar{v}_2 \end{Bmatrix} \tag{23}$$

Onde

$c = \cos\beta$

$s = sen\ \beta$

Agora utilizando o conceito da energia potencial dada por

$$U = \frac{1}{2}\{d^e\}^T\{K^e\}\{d^e\} \tag{24}$$

Agora substituindo $\{d^e\} = [T]\{\bar{d}^e\}$

$$U = \frac{1}{2}\{\bar{d}^e\}^T[T]^T\{K^e\}[T]\{\bar{d}^e\} \tag{25}$$

No sistema de coordenadas $(\bar{x}, \bar{y})$

$$U = \frac{1}{2}\{\bar{d}^e\}^T[T]^T\{K^e\}[T]\{\bar{d}^e\} \tag{26}$$

Por fim, temos a matriz de rigidez transformada:

$$[\bar{K}^e] = \frac{AE}{l}\begin{bmatrix} c^2 & cs & -c^2 & -cs \\ -cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix} \tag{27}$$

Após o cálculo de todas as m matrizes locais onde $m = número\ de\ barras\ da\ treliça$, é necessário anexar cada matriz local em sua respectiva posição na matriz global, de forma que se obtenha uma matriz de dimensão $2n \times 2n$ onde $n = numero\ de\ nós$. A matriz global final será então, formada pela soma da sobreposição dos elementos com o mesmo índice, formando assim uma matriz simétrica. O script utilizado para realizar esse procedimento está descrito no Algoritmo 1 – Anexação das matrizes locais na matriz global e resolução do sistema.

Algoritmo 1 – Anexação das matrizes locais na matriz global e resolução do sistema

Def Resolução_Sistema().

[Matriz_Global]: matriz com dimensão (2*número de nós, 2*número de nós).

[Matriz_Global_Temporaria]: matriz com dimensão (2*número de nós, 2*número de nós).

[Matriz_Local]: matriz com dimensão (4,4).

[Matriz_Temporaria]: matriz de zeros com dimensão (2*número de nós, 4).

Laço for: irá se repetir n vezes onde n é o número de elementos da treliça.

[Matriz_Local] = $[\overline{K}^e]$ aplicado para cada elemento da treliça.

Para cada elemento, será inserido uma unidade na [**Matriz_Temporaria]** nas posições referentes à numeração dos nós associados aquele elemento, deste modo a [**Matriz_Temporaria]** terá 4 posições com o valor 1.

[Matriz_Global_Temporaria]=

[Matriz_Temporaria][Matriz_Local][Matriz_Temporaria]$^{\boldsymbol{T}}$.

[Matriz_Global] = soma de todas as [**Matriz_Global_Temporaria**].

Vetor_forcas = leitura do arquivo contendo as forças.

If a linhas ou coluna **in [Matriz_Global] ou Vetor_forcas** for referente ao nó com condição de contorno:

Zerar linha e coluna.

Else:

Continue.

Vetor_deslocamentos = resolução de K$u = f$.

Return Vetor_deslocamentos.

3. METODOLOGIA

A resolução de uma treliça planar 2D utilizando o método dos elementos finitos, seja para qualquer configuração, no caso desse trabalho uma treliça de 10 barras e uma de 52 barras, já é um problema conhecido e estudado na engenharia.

Para treliças mais simples, o método dos elementos finitos é uma ótima solução, porém o trabalho evolui da solução convencional em dois principais aspectos. Primeiramente criando uma rede neural que aprenda como substituir o método dos elementos finitos, mesmo que em um primeiro momento a rede demore mais para encontrar os pesos certo do que uma simples resolução via MEF, para treliça mais complexas a rede pode se tornar uma ferramenta mais poderosa.

Outro aspecto que este trabalho propôs foi o de estudar a implementação de uma segunda rede que consiga fazer o caminho inverso do MEF, ou seja, dado um par de deslocamento e tensão máxima, a rede retorna um conjunto de áreas que minimize o peso da treliça, tornando esse também um problema de otimização.

Através das duas redes neurais implementadas, um conjunto de dados foi criado com o objetivo de fornecer as áreas que minimizem o peso de qualquer estrutura fornecida através da minimização das áreas das seções transversais de cada barra da estrutura.

3.1. FLUXOGRAMA

Para este trabalho, foram utilizados o Método da Rigidez Direta, a aplicação de um algoritmo genético e métodos DL. Para uma melhor compreensão do que será abordado, a Figura 13 mostra um fluxograma que nos dá uma visão geral do trabalho.

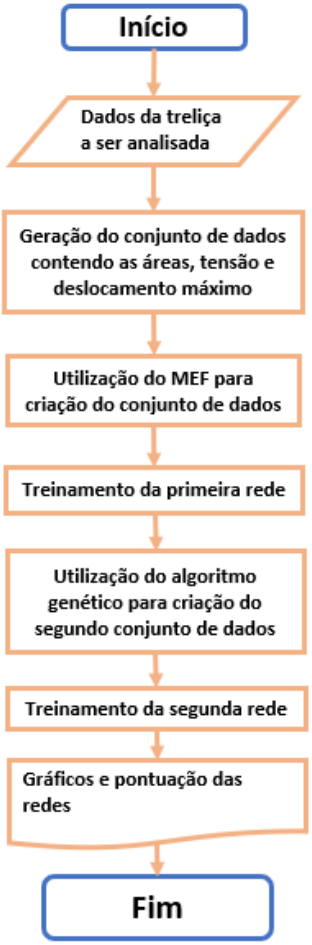


Figura 13. Fluxograma do projeto (elaborado pelo autor)

3.2. TRELIÇAS UTILIZADAS NO ESTUDO

3.2.1. Primeiro caso

Como dito anteriormente, o presente trabalho tem como objetivo a replicação dos resultados obtidos no artigo (Nguyen, 2020). A primeira treliça utilizada nesse artigo e que será reproduzida nesse projeto é uma treliça planar de 10 barras com um comprimento 9,144 m por barra com duas cargas pontuais aplicadas nos nós 2 e 4 (Figura 14).

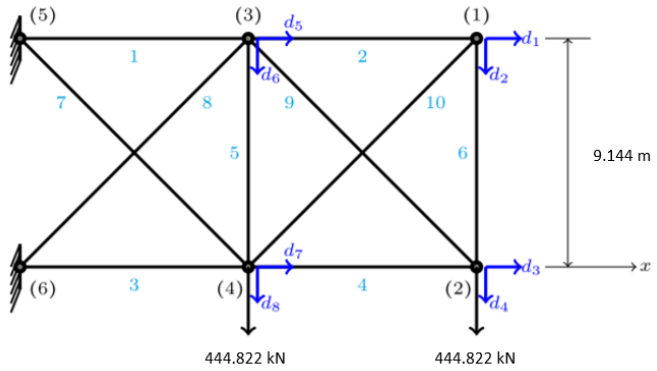


Figura 14. Configuração da treliça de 10 barras *(Nguyen, 2020)*.

3.2.2. Segundo caso

Assim como em (Nguyen, 2020), uma segunda treliça também será utilizada nesse estudo para efeito de validação. Como no primeiro caso, essa também é uma treliça planar, porém possui 52 barras de 2 ou 3m de comprimento (Figura 15).

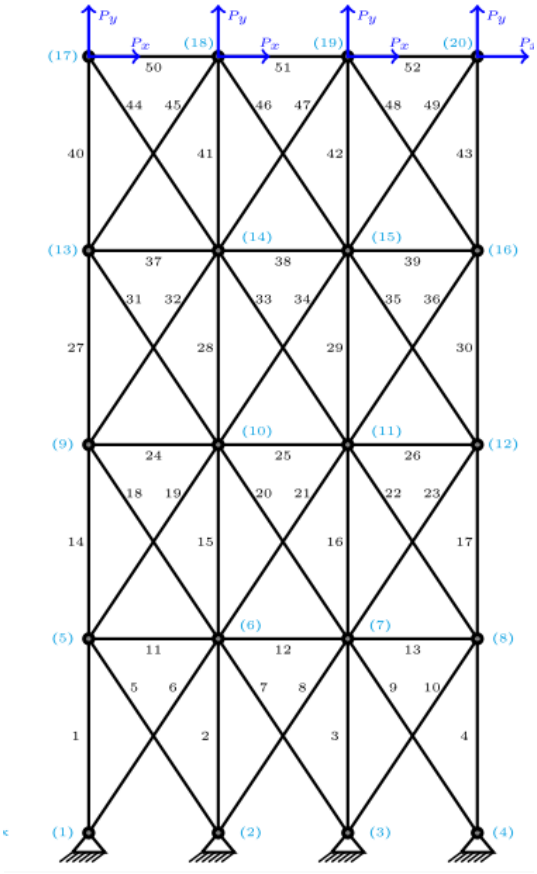


Figura 15. Configuração da treliça de 52 barras *(Nguyen, 2020)*.

3.3.GERAÇÃO DA POPULAÇÃO

Para a rede poder aprender, não podemos inserir uma população cujos parâmetros são iguais, portanto, deve existir uma variável que seja diferente ao longo de toda população. Alguns candidatos para esse papel seriam mudar a geometria da treliça para cada exemplo que será passado para rede, alterar o nó que a força será aplicada ou seu módulo ou até mesmo os nós que possuem condições de contorno. Porém para o presente trabalho, as áreas da seção transversal juntamente com o par de deslocamento e tensão máximas serão a variáveis escolhidas, esses últimos ariáveis de projeto úteis para análises.

Como apresentado na seção anterior, as treliças possuem o comprimento de suas barras fixos, porém a área da seção transversal será uma importante variável para o estudo desse projeto. Cada barra da treliça terá uma área de seção transversal que será o parâmetro que irá mudar ao longo de toda a população. Porém encontrar um banco de dados que possua um número suficientes de treliças com diferentes áreas de seção transversal para treinar a rede, com a geometria pré-determinada e os parâmetros escolhidos não é tão simples. Portanto para contornar esse problema fizemos uso do método dos elementos finitos para gerar a população de treinamento da rede.

Criar a própria população que será estudada tem uma série de vantagens, além de ser um ótimo exercício para o aprendizado do funcionamento e implementação do método dos elementos finitos em *Python*, tendo o controle de como seus dados serão gerados, podemos facilmente mudar qualquer parâmetro desejado e escolher os dados que mais nos são interessantes e criar um *dataset* com apenas variáveis que serão interessantes para o trabalho. O objetivo dessa seção é explicar como a população foi criada.

3.3.1. Estratégia para geração das áreas

Um dos argumentos que será passado para a função que resolve o método dos elementos finitos implementado em *Python* é a área da seção transversal das barras da treliça. Tendo isso em mente primeiramente foi gerada uma população de áreas resultando em um vetor de n_a linhas onde n_a é o número de linhas do *dataset* que será inserido na camada de entrada da rede. Inicialmente não era claro o valor exato de n_a que treinaria a rede suficientemente bem, então foi adotado um valor $n_a = 10000$.

A abordagem utilizada para a aleatoriedade das áreas utilizada nesse projeto difere do artigo, a explicação de cada uma bem como a diferença entre elas será apresentada nessa seção.

O método utilizado em (Nguyen, 2020) utiliza uma constante de controle A para a soma das áreas de cada conjunto. Explicando melhor, cada linha do vetor que será gerado possui 10 valores de área que são gerados aleatoriamente entre $0{,}001045\ e\ 0{,}02161\ m^2$, a constante A então é definida como um valor entre $(0{,}01045\ e\ 0{,}2161)$. Após a constante A ter seu valor definido, todos os conjuntos de área do conjunto de dados deverão ter sua soma igual a A, caso o valor extrapole ou não alcance o valor de A, um algoritmo já implementado recalcula as áreas de modo que cada iteração do cálculo deixe o valor da soma das áreas mais próximo do valor da constante. Utilizando esse método o artigo de referência cria 500 conjuntos de dados. Essa é uma estratégia válida que produziu os resultados esperados, porém a abordagem utilizada nesse trabalho é de certa forma mais simples e arbitrária, contribuindo para a convergência, alcançando resultados satisfatórios.

3.3.1.1. Primeiro caso

Ao invés de buscar minimizar o número de conjunto de áreas, os deslocamentos e tensões máximas também foram geradas de forma aleatória da mesma maneira que as áreas. Fazendo isso, podemos obter uma rede que será treinada para uma gama de tensões de escoamento e limite de deslocamentos máximos diferentes.

Algoritmo 2 – Função geradora da população de áreas

Def Geracao_areas:

Laço for: irá se repetir **n** vezes, onde **n** é o numero de conjuntos de áreas que se deseje gerar.

Laço for: irá se repetir **j** vezes, onde **j** é o número de elementos da treliça.

Areas = $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ para cada elemento da treliça.

Conjunto_areas = vetor com **j Areas**.

População_areas = vetor com **n Conjunto_areas**.

Return População_areas.

Onde

$\sigma = desvio\ padrão$

$\mu = média$

$\sigma^2 = variância$

Após a geração das áreas utilizando a função de densidade de probabilidade de distribuição normal contida na biblioteca *numpy*, um histograma das amostras foi gerado para uma melhor visualização dos valores dos conjuntos (Figura 16).

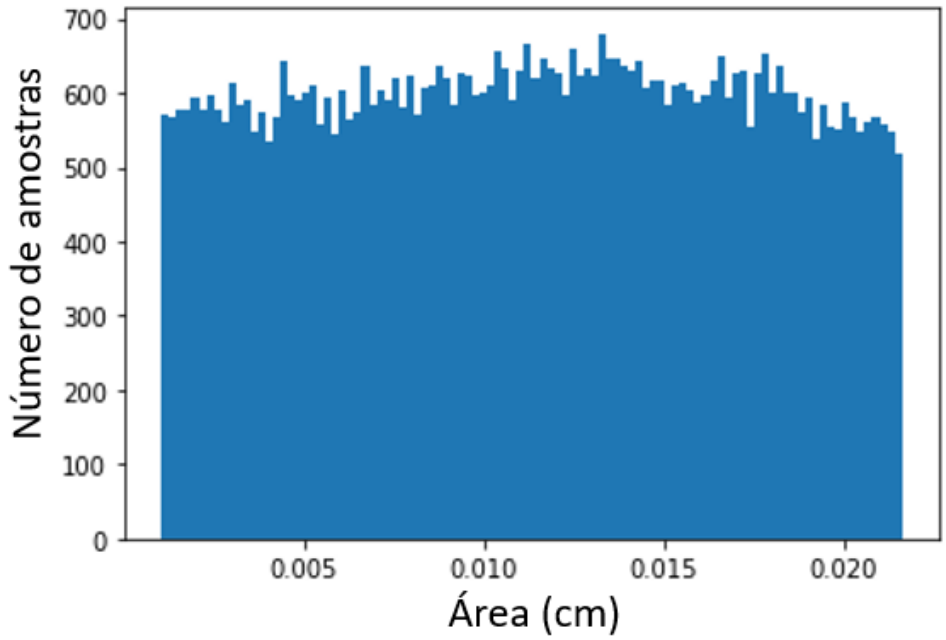


Figura 16 Histograma das amostras de áreas (elaborada pelo autor).

Com a função de distribuição normal, não temos um acúmulo onde a maior parte das amostras se encontra na média, e sim uma distribuição bem mais uniforme dos valores. Para obter o vetor de áreas que foi utilizado no treinamento, foram usadas como argumentos da função *numpy.random.normal()* um valor de $\sigma = 0{,}0128385$ *e* $\mu = 0{,}015329$. Esses valores forem obtidos via tentativa e erro até que a distribuição fosse satisfatória.

Para cada conjunto de áreas gerados anteriormente, um par de deslocamento e tensão máximas foram anexados, fazendo com que agora cada linha do conjunto de dados tenha 12 valores, sendo 10 áreas, um deslocamento máximo e uma tensão máxima, como mostra a Tabela 1.

O algoritmo 3 explica como essa configuração final do conjunto de dados é gerada, bem como os intervalos para os valores de deslocamento e tensão.

Algoritmo 3 – Geração final da entrada da primeira rede neural

For i in range(10000):

Populacao_areas = Geracão_areas(n=10000, j=10).

For i in Populacao_areas:

Anexar um valor para o deslocamento máximo entre $0.0381\ e\ 0.254\ m$ na última posição de cada linha de **i**.

Anexar um valor para a tensão máxima entre $68.95\ e\ 689.5MPa$ na última posição de cada linha de **i**.

Para tratar dos conjuntos de dados, a biblioteca *Pandas* foi utilizada, essa é uma biblioteca para análise de dados. Seu nome é derivado do termo "dados de painel" (*panel data*), um termo econométrico utilizado para se referir a conjuntos de dados estruturados multidimensionais. O Pandas introduziu dois novos tipos de objetos de armazenamento de dados: *Panda Series*, que tem uma estrutura em forma de lista, e *DataFrames*, que tem uma estrutura tabular. Os conjuntos de dados utilizados nesse projeto foram todos criados e manipulados utilizando os *Dataframes* de *Pandas.*

Para efeito de exemplificação, vamos escolher um candidato aleatório. Dentre o vetor com 100000 linhas, sendo 10000 conjuntos de áreas e 10 pares de deslocamento e tensão máximos por área, vamos tomar como exemplo o candidato número 12 e chama-lo de C_{12}.

0 C_{12} é uma lista contendo doze valores, 10 seções de áreas e um par de deslocamento e tensão máximos. Analisando melhor nosso candidato, vemos que seus 10 primeiros valores são:

Tabela 1. Valores das áreas, tensão e deslocamentos para um indivíduo do conjunto de dados

Posição do vetor	Valor
0	0.009547576
1	0.006029259
2	0.001399122
3	0.005749918
4	0.001367939
5	0.007794418
6	0.020270571
7	0.019539495
8	0.018979861
9	0.021266369
10	0.160795766
11	0.339005334

Como esperado, todos os valores de áreas estão entre $0{,}001045\ e\ 0{,}02161\ m^2$.

Seu valor de deslocamento máximo e tensão de escoamento são respectivamente $0{,}160795766\ e\ 0{,}339005336$. Note que o valor para a tensão foi na verdade normalizado fazendo uma simples divisão por 10^9, o motivo para isso será explicado na seção 4.2.

Com o vetor contendo o conjunto de áreas e o par de tensão e deslocamento máximos em mãos, o próximo passo é usá-lo como argumento no código no qual o método dos elementos finitos foi implementado.

3.3.1.2. Segundo caso

A estratégia utilizada para o segundo caso se assemelha muito ao caso anterior, apenas alterando dois procedimentos em toda a etapa. A primeira mudança é o valor para o deslocamento e tensão máximas, para o caso dessa treliça de 52 barras, os deslocamentos foram gerados aleatoriamente entre $0.004\ e\ 0.012m$ enquanto as tensões ficaram em um intervalo entre $15\ e\ 25GPa$.

Além disso, para não tornar a rede mais complexa do que o necessário, não foram gerados conjuntos com 52 seções de áreas transversais, uma área para cada barra. Ao invés disso, as barras foram divididas em 12 grupos (Tabela 2):

Tabela 2. Divisão dos conjuntos de áreas para a treliça de 52 barras.

Índice	Conjunto de Áreas
1	$A_1 - A_4$
2	$A_5 - A_{10}$
3	$A_{11} - A_{13}$
4	$A_{14} - A_{17}$
5	$A_{18} - A_{23}$
6	$A_{24} - A_{26}$
7	$A_{27} - A_{30}$
8	$A_{31} - A_{36}$
9	$A_{37} - A_{39}$
10	$A_{40} - A_{43}$
11	$A_{44} - A_{49}$
12	$A_{50} - A_{52}$

3.3.2. MEF implementado em *Python*

A função que contém o a solução do método dos elementos finitos possui três argumentos, o módulo de elasticidade, o vetor de áreas e os nós que possuem condição de contorno. Sua implementação foi feita em *Python* retratado no Algoritmo 4.

Algoritmo 4 – MEF implementado em *Python*

```
Def Truss_Solver (E, população_areas, cc)

    Matriz_Conectividade = read: matriz_conectividade.txt (matriz que relaciona o número de cada elemento com seus nós associados).

    Matriz_malha = read: matriz_malha.txt (matriz que relaciona cada nó com sua posição no plano de coordenadas).

    Forcas = read: forcas.txt. (matriz que relaciona a força aplicada com sua respectiva direção e nó associado).

    Resolucao_Sistema().

    Desloc_max = max([abs(u) for u in Vetor_deslocamentos]).

    Tensão_max = E*Desloc_max.

    Desloc_max_random = população_areas[10].
    Tensão_max_random = população_areas[11].
    If Tensão_max /1e9 <= Tensão_max_random:
        Tensao_max_binario = 1.
    Else:
        Tensao_max_binario = 0.
    If Desloc_max <= Desloc_max_random:
        Des_max_binario = 1.
    else:
        Des_max_binario = 0.


    return  (des_max_binario, tensao_max_binario)
```

O módulo de elasticidade escolhido para o primeiro caso foi o do alumínio 6262 $E = 69\ GPa$ e os nós que possuem condição de contorno podem ser observados na Figura 14 (nó 4 e 2)

Já para o segundo caso, foi escolhido E = 270 Gpa e os nós que possuem condição de contorno podem ser observados na Figura 15 (nó 1, 2, 3 e 4)

A geometria da treliça foi definida como sendo fixa, então o código faz o reconhecimento da estrutura treliçada através de um arquivo de texto que contém as coordenadas de cada nó, a relação entra as barras e os nós e os nós que possuem alguma força aplicada.

O script irá retornar 2 valores binários, caso o módulo da maior tensão sofrida por qualquer uma das barras seja maior que o valor de escoamento pré-definido, ele retorna 0, caso esteja dentro do limite retorna 1. Isso é executado da mesma forma para o deslocamento máximo, retornando o valor de 0 caso seja ultrapassado ou 1 caso contrário.

Essa foi a abordagem escolhida pois além da resposta binária representar muito bem uma margem positiva ou não para a estrutura, predições com os valores reais foram testadas, porém isso aumentou desnecessariamente a complexidade do problema.

Agora que todos os parâmetros foram definidos, a função é executada n_a vezes, faz com que o código passe todos os conjuntos de áreas e retorne n_a pares de valores binários.

3.4. IMPLEMENTAÇÃO DA REDE

Falando sobre a estratégia utilizada para a implementação da rede que fará o treinamento utilizando aprendizado profundo, foram escolhidas as bibliotecas *Keras, Tensorflow e Scikit-learn* da linguagem *Python*.

Primeiramente, é preciso definir as entradas e saídas da rede. Para o primeiro caso camada de entrada foi definida como tendo 12 neurônios, que recebem os valores das 10 seções de áreas de cada conjunto e também o deslocamento e tensão máximas do conjunto em questão.

Já no segundo caso, a camada de entrada possui 14 neurônios, pois possui dois valores de áreas a mais que no caso anterior. A camada de saída foi definida como tendo 2 neurônios, que são as respostas obtidas através do MEF, sendo 1 ou 0.

As definições acima foram um fator em comum entre todos os treinamentos, porém como dito anteriormente nesse trabalho, os hiper parâmetros finais da rede treinada que serão utilizados serão obtidos através da pontuação da rede para várias combinações diferentes.

3.5. IMPLEMENTAÇÃO DO ALGORITMO GENÉTICO

Depois que a rede é treinada até atingir uma acurácia satisfatória, um script utilizando algoritmo genético foi implementado para que se possa gerar um conjunto de áreas ótimas.

Para isso, foi utilizada a função *scipy.optimize.diferential_evolution*, uma função pertencente a biblioteca *scipy* que utiliza um método estocástico que é útil para problemas de otimização global.

A função que essa função minimizou foi a função peso da treliça, como o comprimento e a densidade são constantes, as áreas foram as variáveis que foram otimizadas.

A *scipy.optimize.diferential_evolution* recebeu como variáveis uma função a ser otimizada, no caso o peso da treliça, limites de valores para as variáveis a serem otimizadas, onde foram utilizados os mesmos limites usados para gerar as áreas anteriormente, um número máximo de iterações definido como 200, uma população inicial de 20 indivíduos e uma taxa de mutação de 5%.

Um detalhe crucial desse projeto é o método com o qual o peso da treliça foi passado para o algoritmo genético. Quando a função peso é chamada pelo algoritmo genético, além de retornar o peso total da treliça, uma verificação é feita, utilizando a primeira rede neural já treinada, caso o conjunto de áreas falhe por tensão ou deslocamento, o indivíduo é penalizado automaticamente, dessa forma, os indivíduos que não falharem de nenhum dos dois modos substituirão os indivíduos penalizados. A penalização ocorre no processo de verificação pela passagem pela primeira rede, no caso de falha, o indivíduo em questão tem seu valor da soma total do peso de cada barra da treliça multiplicada por dez vezes o limite máximo, fazendo com que ele seja posteriormente descartado pelo algoritmo, por ter resultado em uma pontuação muito alta.

Essa estratégia foi utilizada pois a função que resolve o método dos elementos finitos usada para gerar o primeiro conjunto de dados poderia ser utilizada, porém o tempo e custo computacional seria extremamente maior, visto que depois de treinada, uma rede neural apenas realiza vários cálculos de multiplicação entre a entrada e os pesos já treinados, oque a torna mais rápida e eficiente que uma função mais complexa como o método dos elementos finitos.

3.6. IMPLEMENTAÇÃO DA REDE NEURAL FINAL

A última etapa do estudo é criar uma rede neural, que irá receber como dados de entrada um par de deslocamento e tensão máximos e retornar um conjunto com as áreas ótimas.

O conjunto de dados que foi utilizado para treinar essa rede foi gerado pelo algoritmo genético citado no item anterior, devido ao tempo necessário para a geração de cada conjunto de áreas, diferente do primeiro conjunto de dados esse teve apenas 1200 exemplos.

Se tratando agora de um problema de regressão e não classificação, foi utilizada a função de ativação ReLU ao invés da Sigmóide e a função de custo escolhida foi o erro quadrático médio (*Mean Sqquared Error*)

Assim como no primeiro caso, os resultados obtidos bem como os hiper parâmetros com o melhor desempenho foram inseridos na seção de resultados.

4. RESULTADOS E DISCUSSÃO

Nesta seção serão apresentados e discutidos os resultados para ambos os casos citados acima, e também seus desempenhos em ambas as redes, no Algoritmo Genético e na resolução do Método dos Elementos Finitos.

4.1. RESULTADOS DO MEF IMPLEMENTADO EM PYTHON

A implementação do método dos elementos finitos em um script em *Python* foi um sucesso, o código retorna os valores esperados. Aplicando as duas treliças apresentadas nesse estudo, o código consegue encontrar a solução para essas e qualquer estrutura treliçada 2D, desde que se tenha as coordenadas de cada um de seus nós em um arquivo de texto, o mesmo retornou os valores exatos para deslocamentos, tensões e reações dessa forma validando a eficácia do código.

Com o código funcionando como deveria, a configurações de treliça para ambos os casos estudados foram implementados e então a função foi executada n_a vezes, formando assim os *dataset*s que serão utilizados na camada de saída da rede.

Como pode ser vistos nas figuras Figura 17 e Figura 18, De acordo com as forças que foram aplicadas nesse estudo, o algoritmo pode reproduzir os resultados esperados e gerar a forma deformada da treliça.

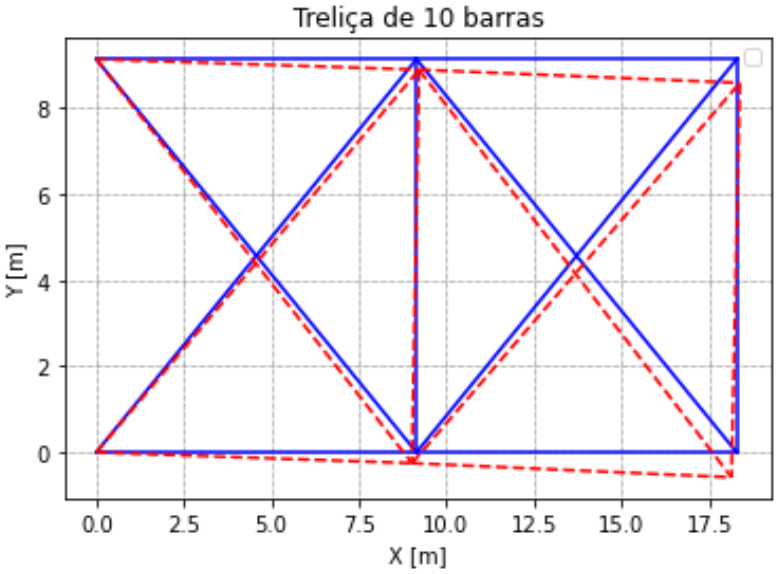


Figura 17 Deformação da treliça de 10 barras (elaborada pelo autor).

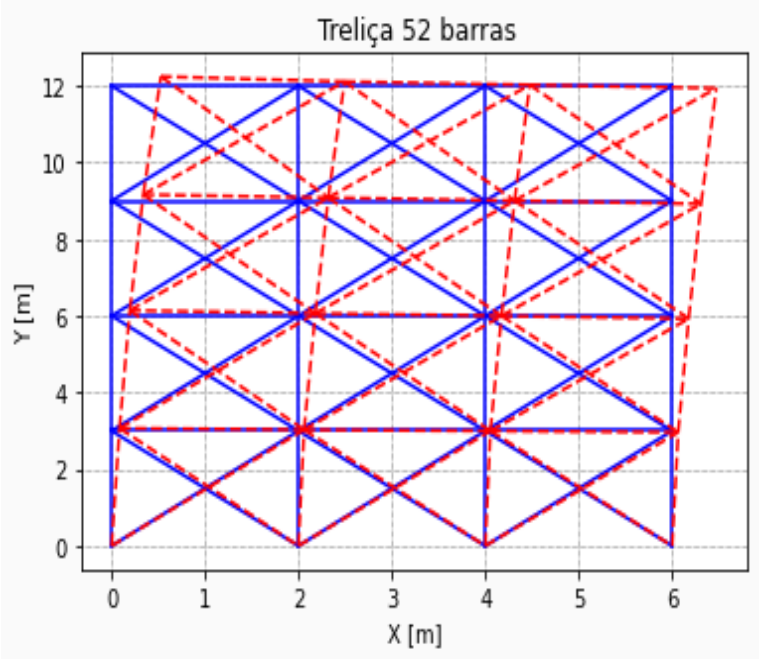


Figura 18. Deformação da treliça de 52 barras (elaborada pelo autor).

----- Treliça deformada

——— Treliça original

4.2. RESULTADOS DA PRIMEIRA REDE

Primeiramente foi feita uma divisão do *dataset*, 95% dele foi utilizado no treino e 5% foi utilizado para teste como forma de validação.

A rede teve uma série de resultados de acordo com a configuração dos hiper-parâmetros escolhidos. Nos primeiros modelos, a rede não obteve sequer a conversão para algum resultado, pois acabava se perdendo durante o processo de treinamento.

O primeiro grande problema logo no início dos primeiros modelos se deu, pois, a rede não conseguia convergir para nenhuma configuração, ou convergia com um resultado extremamente ruim. Esse primeiro erro ocorreu, pois nos primeiros modelos, a tensão máxima utilizada para o treinamento não estava normalizada, por conta disso em cada amostras os valores das áreas e deslocamento máximos tinham ordem entre 10^{-2} *e* 10^{-1} e um dos valores relacionado a tensão máxima tinha ordem entre 10^{8} *e* 10^{9}. Por conta dessa diferença entre os valores, a rede não conseguia encontrar um conjunto de pesos e vieses que resolvesse o problema. Após ser feita a normalização da tensão, a rede parou de se perder tão facilmente e começou a encontrar resultados melhores.

Outro fator que foi corrigido e estava causando a divergência da rede foi a função de ativação utilizada na última camada, como os valores de saída sempre são sempre 0 ou 1, utilizar a função *softmax* que pode assumir valores negativos ou a ReLU que pode assumir valores maiores que 1, foi fixada como padrão a função *sigmoid*, fazendo com que os resultados da última camada sempre fiquem entre 0 e 1.

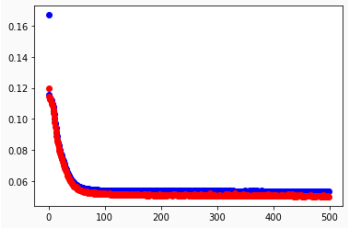
A utilização da função *tangente hiperbólica* não trouxe bons resultados em nenhum dos testes, então as configurações de arquitetura final utilizaram como função de ativação as funções *Sigmoid* ou ReLU.

Para a taxa de aprendizado, foram testados os resultados para 3 ordens de grandeza, alternando-a entre $0{,}001$, $0{,}01$ *e* $0{,}1$

A alternância entre essas configurações foi testada para quatro diferentes arquiteturas de rede, variando com número de camadas ocultas e número de neurônios por camada. Para comparar os resultados das configurações escolhidas, a pontuação R2 foi utilizada e as tabelas a seguir mostram a performance da rede para as configurações citadas acima, bem como os gráficos da evolução do aprendizado para a população de treino e teste, onde para a arquitetura da rede, o primeiro valor se refere ao número de neurônios na camada de entrada, o último valor o número de neurônios na camada de saída, e os valores entre esses o número de neurônios em cada camada oculta.

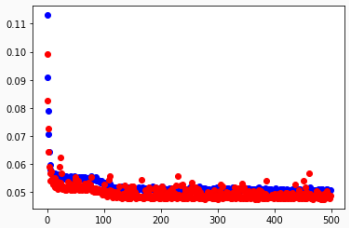
Arquitetura da rede: 12-12-2
Camadas ocultas: Sem funções de ativação
Camada de saída: Sigmoid

Learning rate = 0.001



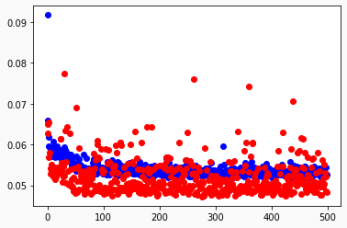
R2 (treino) = 0.450
R2 (teste) = 0.441

Learning rate = 0.01



R2 (treino) = 0.494
R2 (teste) = 0.503

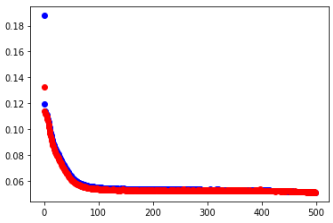
Learning rate = 0.1



R2 (treino) = 0.537
R2 (teste) = 0.520

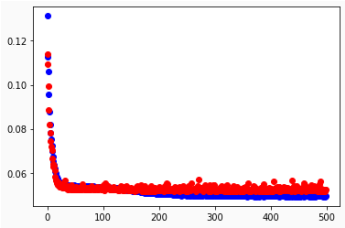
Arquitetura da rede: 12-12-2
Camadas ocultas: Relu
Camada de saída: Sigmoid

Learning rate = 0.001



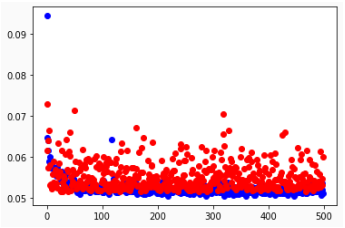
R2 (treino) = 0.505
R2 (teste) = 0.528

Learning rate = 0.01



R2 (treino) = 0.537
R2 (teste) = 0.579

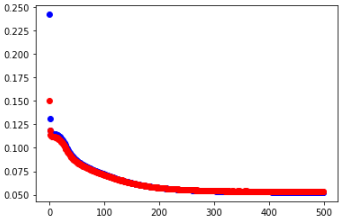
Learning rate = 0.1



R2 (treino) = 0.347
R2 (teste) = 0.345

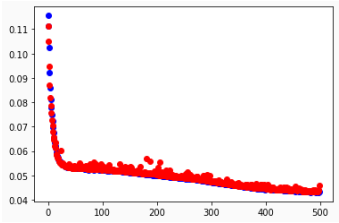
Arquitetura da rede: 12-12-2
Camadas ocultas: Sigmoid
Camada de saída: Sigmoid

Learning rate = 0.001



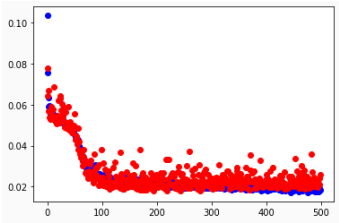
R2 (treino) =0.489
R2 (teste) = 0.489

Learning rate = 0.01



R2 (treino) = 0.551
R2 (teste) = 0.545

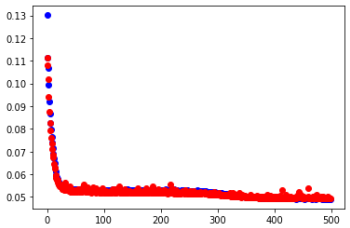
Learning rate = 0.1



R2 (treino) = 0.826
R2 (teste) = 0.845

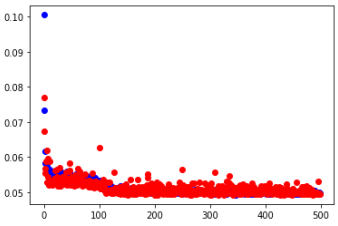
Arquitetura da rede: 12-100-2
Camadas ocultas: Sem funções de ativação
Camada de saída: Sigmoid

Learning rate = 0.001



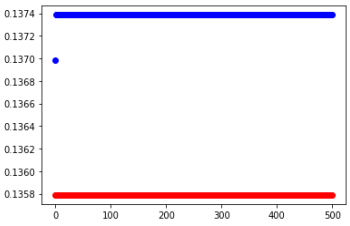
R2 (treino) = 0.491
R2 (teste) = 0.483

Learning rate = 0.01



R2 (treino) = 0.534
R2 (teste) = 0.584

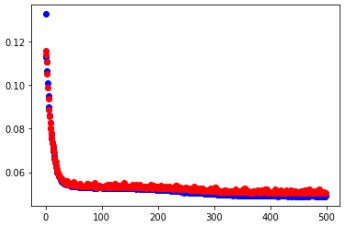
Learning rate = 0.1



R2 (treino) = -0.163
R2 (teste) = -0.178

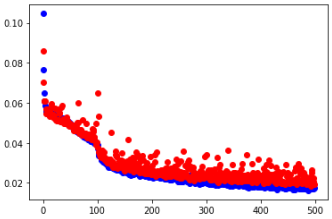
Arquitetura da rede: 12-100-2
Camadas ocultas: Relu
Camada de saída: Sigmoid

Learning rate = 0.001



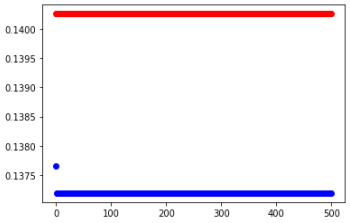
R2 (treino) =0.537
R2 (teste) = 0.523

Learning rate = 0.01



R2 (treino) = 0.758
R2 (teste) = 0.721

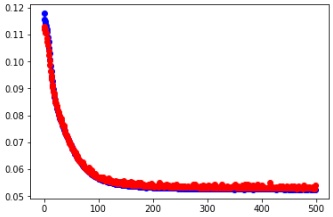
Learning rate = 0.1



R2 (treino) = -0.163
R2 (teste) = -0.177

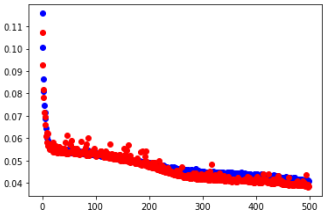
Arquitetura da rede: 12-100-2
Camadas ocultas: Sigmoid
Camada de saída: Sigmoid

Learning rate = 0.001



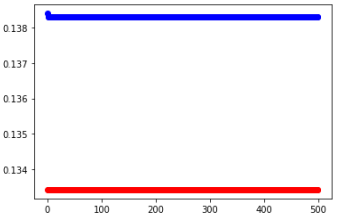
R2 (treino) = 0.486
R2 (teste) = 0.517

Learning rate = 0.01



R2 (treino) = 0.604
R2 (teste) = 0.593

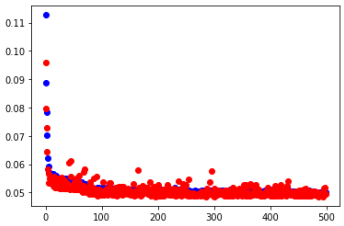
Learning rate = 0.1



R2 (treino) = -0.164
R2 (teste) = -0.162

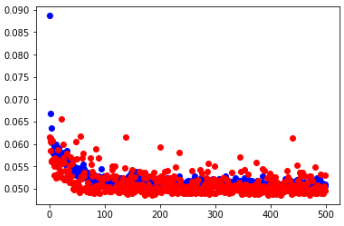
Arquitetura da rede: 12-100-100-2
Camadas ocultas: Sem funções de ativação
Camada de saída: Sigmoid

Learning rate = 0.001



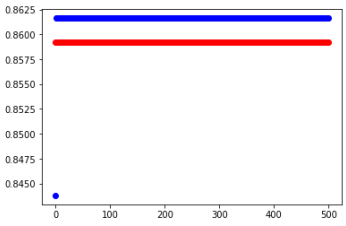
R2 (treino) = 0.540
R2 (teste) = 0.477

Learning rate = 0.01



R2 (treino) = 0.538
R2 (teste) = 0.475

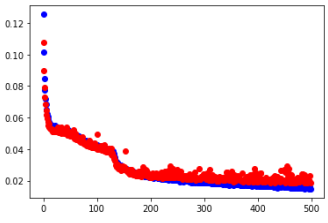
Learning rate = 0.1



R2 (treino) = -0.165
R2 (teste) = -0.143

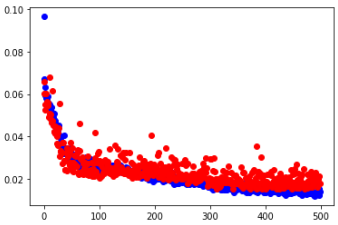
Arquitetura da rede: 12-100-100-2
Camadas ocultas: Relu
Camada de saída: Sigmoid

Learning rate = 0.001



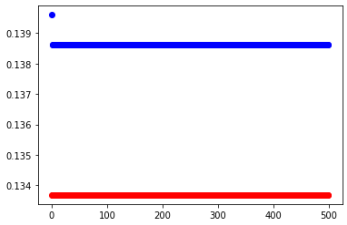
R2 (treino) = 0.864
R2 (teste) = 0.869

Learning rate = 0.01



R2 (treino) = 0.903
R2 (teste) = 0.910

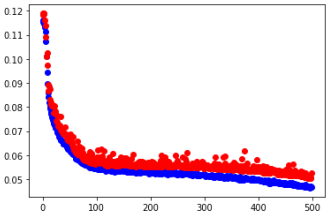
Learning rate = 0.1



R2 (treino) = -0.164
R2 (teste) = -0.162

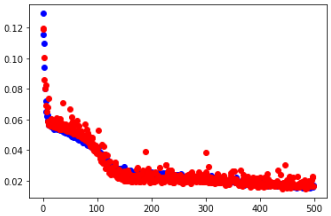
Arquitetura da rede: 12-100-100-2
Camadas ocultas: Sigmoid
Camada de saída: Sigmoid

Learning rate = 0.001



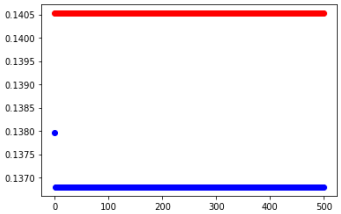
R2 (treino) = 0.529
R2 (teste) = 0.555

Learning rate = 0.01



R2 (treino) = 0.869
R2 (teste) = 0.860

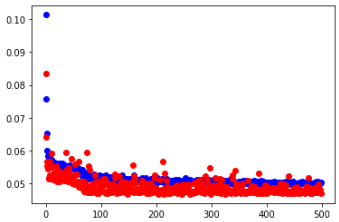
Learning rate = 0.1



R2 (treino) = -0.163
R2 (teste) = -0.177

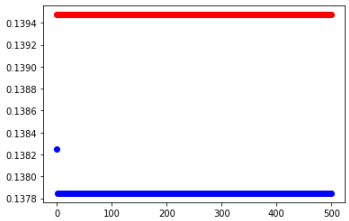
Arquitetura da rede: 12-100-100-100-2
Camadas ocultas: Sem funções de ativação
Camada de saída: Sigmoid

Learning rate = 0.001



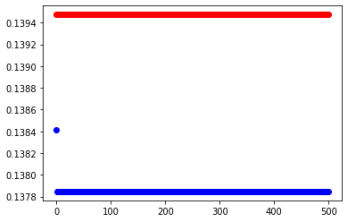
R2 (treino) = 0.544
R2 (teste) = 0.479

Learning rate = 0.01



R2 (treino) =-0.165
R2 (teste) = -0.149

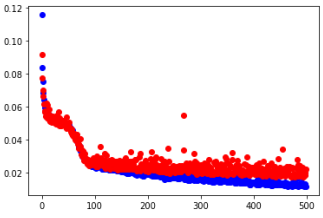
Learning rate = 0.1



R2 (treino) = -0.165
R2 (teste) = -0.149

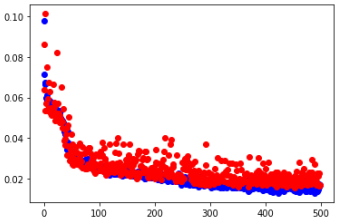
Arquitetura da rede: 12-100-100-100-2
Camadas ocultas: Relu
Camada de saída: Sigmoid

Learning rate = 0.001



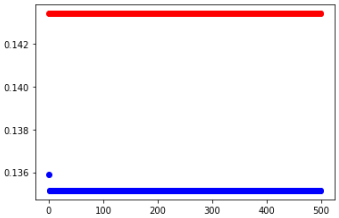
R2 (treino) = 0.840
R2 (teste) = 0.827

Learning rate = 0.01



R2 (treino) = 0.896
R2 (teste) = 0.866

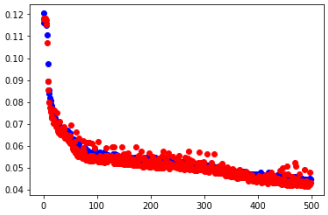
Learning rate = 0.1



R2 (treino) = -0.163
R2 (teste) = -0.188

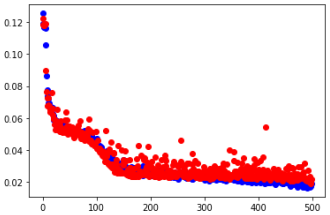
Arquitetura da rede: 12-100-100-100-2
Camadas ocultas: Sigmoid
Camada de saída: Sigmoid

Learning rate = 0.001



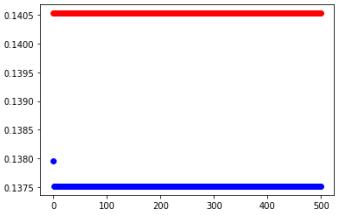
R2 (treino) = 0.565
R2 (teste) = 0.502

Learning rate = 0.01



R2 (treino) = 0.96
R2 (teste) = 0.95

Learning rate = 0.1



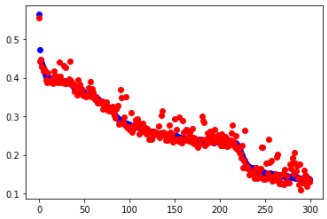
R2 (treino) = -0.164
R2 (teste) = -0.152

Analisando as pontuações acima, vemos que a melhor pontuação foi alcançada para uma arquitetura 12-100-100-100-2, um $learning\ rate = 0.01$ e camadas ocultas com função de ativação Sigmoid e camada de saída com função de ativação Sigmoid, com um $R2 = 0.96\ para\ treino\ e\ 0{,}95\ para\ teste$

Utilizando uma rede neural muito semelhante à da primeira rede, apenas foi alterado o número de neurônios na camada de entrada de 12 para 14. As configurações que obtiveram os melhores resultados na primeira rede foram utilizadas, não necessitando novamente a alteração de vários hiper parâmetros em busca da melhor performance, já que o problema era muito semelhante ao caso anterior.

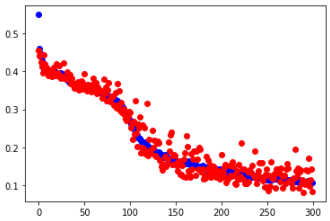
Arquitetura da rede: 14-100-100-100-2
Camadas ocultas: Relu
Camada de saída: Sigmoid

Learning rate = 0.001



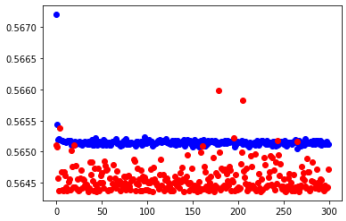
R2 (treino) = 0.797
R2 (teste) = 0.793

Learning rate = 0.01



R2 (treino) = 0.818
R2 (teste) = 0.798

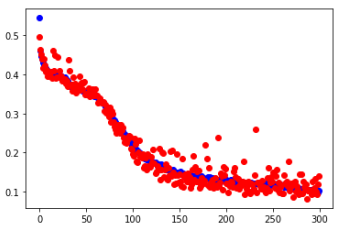
Learning rate = 0.1



R2 (treino) = -0.01
R2 (teste) = -0.01

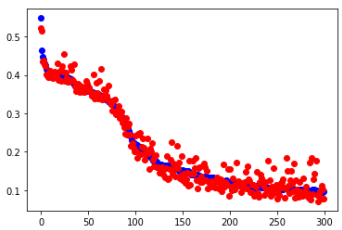
Arquitetura da rede: 14-100-100-100-2
Camadas ocultas: Sigmoid
Camada de saída: Sigmoid

Learning rate = 0.001



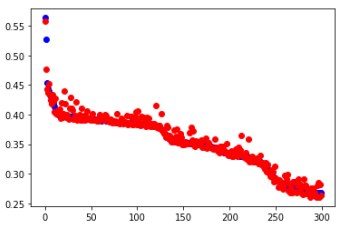
R2 (treino) = 0.824
R2 (teste) = 0.814

Learning rate = 0.01



R2 (treino) = 0.882
R2 (teste) = 0.880

Learning rate = 0.1



R2 (treino) = 0.578
R2 (teste) = 0.566

4.3. RESULTADOS ALGORITMO GENÉTICO

Após o algoritmo genético gerar 3000 exemplos para o conjunto de dados, foi feita uma validação dos resultados inserindo as áreas geradas pelo script juntamente com a tensão e deslocamento máximos relacionados a cada indivíduo na função que resolve o problema pelo método dos elementos finitos.

O resultado obtido foi muito satisfatório, pois em cerca de 90% dos casos as áreas não falharam por tensão ou deslocamento. Para o intervalo de valores escolhidos, o algoritmo convergiu para uma solução onde a tensão nunca chegasse a 70% da tensão limite, mas no caso do deslocamento, para todos os casos o valor obtido não ultrapassava o limite pré-estabelecido. Nos outros 10% dos casos, a tensão também não era ultrapassada, mas alguma barra da treliça falhou por deslocamento, porém por um valor muito pequeno, como mostra a Tabela 3, que contém os resultados para os 10 primeiros pares de tensão e deslocamentos otimizados pelo algoritmo genético após serem inseridos na primeira rede neural já treinada. Isso pode ser facilmente corrigido adicionando um fator de segurança ou adicionando uma margem de segurança aos valores gerados pelo algoritmo genético.

Tabela 3. Resultados para os 10 primeiros conjuntos otimizados pelo algoritmo genético

	Tensão	Deslocamento
1	0,999	0,678
2	0,999	0,875
3	0,999	0,489
4	0,999	0,687
5	0,999	0,863
6	0,999	0,456
7	0,999	0,563
8	0,999	0,478
9	0,999	0,799
10	0,999	0,895

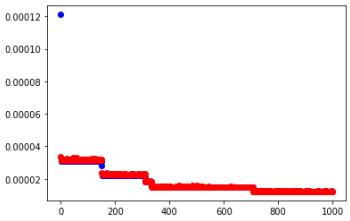
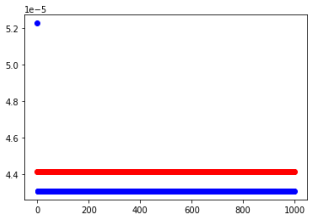
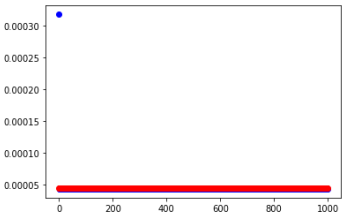
4.4. RESULTADOS SEGUNDA REDE

Para a segunda rede, foi necessária a alteração de alguns hiper parâmetros, pois diferentemente da primeira rede, esse foi um caso de regressão, e não de classificação.

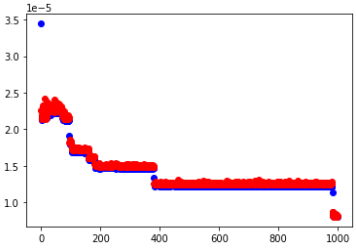
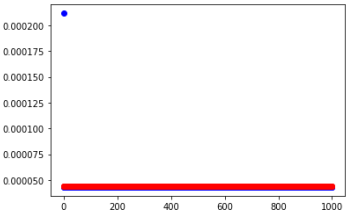
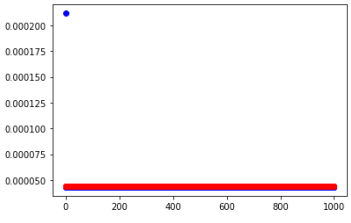
Para obter um melhor desempenho, a função de otimização foi alterada de Sigmóid para ReLU, pois agora os valores de áreas da camada de saída são números reais cima de 0, fazendo com que a ReLU seja a candidata perfeita. Além disso, a função de custo também foi alterada de entropia cruzada para a função de erro quadrático médio, pois também é mais indicada para casos de regressão.

Após a definição desses hiper parâmetros, semelhantemente à primeira rede a taxa de aprendizado e o número de camadas e neurônios por camadas foi alterado para um estudo de quais são os melhores hiper parâmetros. Os gráficos mostrando a curva de aprendizado e a pontuação dos casos estudados estão a seguir.

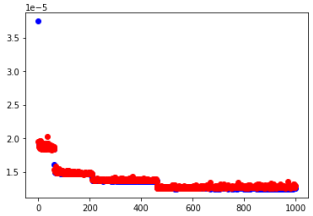
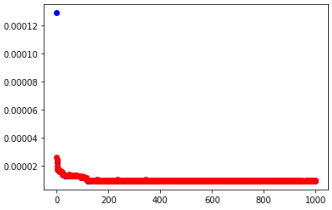
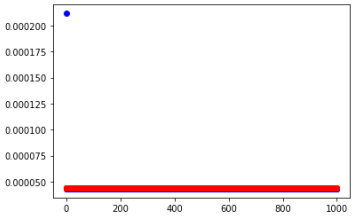
Arquitetura da rede: 2-100-10
Camadas ocultas: Relu
Camada de saída: Relu

Learning rate = 0.001	Learning rate = 0.01	Learning rate = 0.1
		
R2 (treino) = -0.001 R2 (teste) = -0.001	R2 (treino) = -2.100 R2 (teste) = -2.100	R2 (treino) = -2.100 R2 (teste) = -2.100

Arquitetura da rede: 2-100-100-10
Camadas ocultas: Relu
Camada de saída: Relu

Learning rate = 0.001	Learning rate = 0.01	Learning rate = 0.1
		
R2 (treino) = 0.255 R2 (teste) = 0.215	R2 (treino) = -2.100 R2 (teste) = -2.100	R2 (treino) = -2.100 R2 (teste) = -2.100

Arquitetura da rede: 2-100-100-100-10 Camadas ocultas: Relu Camada de saída: Relu

Learning rate = 0.001	Learning rate = 0.01	Learning rate = 0.1
		
R2 (treino) = 0.175 R2 (teste) = 0.171	R2 (treino) = 0.125 R2 (teste) = 0.120	R2 (treino) = -2.100 R2 (teste) = -2.100

5. CONCLUSÃO

O trabalho teve como objetivo a reprodução dos resultados obtidos em (Nguyen, 2020) utilizando uma metodologia diferente, com a população de áreas gerada de forma diferente e com a estratégia de transformar o problema de regressão em um problema de classificação. Segundo os resultados apresentados na seção anterior, foi possível concluir que a estratégia adotada cumpriu os resultados esperados, uma vez que uma pontuação R2 acima de 0.95 para o primeiro caso e um R2 acima de 0.88 para o segundo, já é satisfatória levanto em conta uma população com 100000 conjuntos e um treinamento com 500 épocas.

Um importante ponto a se observar foi que para uma arquitetura de rede com uma camada oculta com 100 neurônios ou para mais camadas ocultas, a taxa de aprendizado de 0.1 não obteve convergência. Com isso podemos concluir que quanto mais complexa a rede se torna, um aprendizado rápido demais faz com que a rede acabe se perdendo e não convergindo e obtendo uma pontuação de R2 negativa.

Vale também ressaltar que o presente trabalho não teve como objetivo principal a criação de uma rede extremamente refinada, mas sim o estudo das diferentes arquiteturas e hiper parâmetros e como eles afetam o aprendizado da rede.

Se tratando do algoritmo genético implementado, o mesmo teve uma performance muito satisfatória, já que cumpriu o objetivo de gerar uma população de áreas ótimas que puderam ser validadas através do método dos elementos finitos.

Por fim, a segunda rede não conseguiu atingir a performance alcançada pela primeira rede, apenas atingindo uma pontuação máxima de $R2 = 0{,}255$ para teste e $R2 = 0{,}215$ para treino. Isso provavelmente se deve ao fato de um problema mais complexo, que possui apenas 2 variáveis de entrada e 10 variáveis de saída, fazendo com que seja mais complexo a identificação de padrões pela rede, e também pois o tamanho do conjunto de dados era inferior ao do primeiro caso. Enquanto o script utilizando o método dos elementos finitos gerou a população para a primeira rede, para a segunda o gerador do conjunto de dados foi a otimização do algoritmo genético, fazendo com que a resolução e otimização das áreas seja mais lenta.

6. BIBLIOGRAFIA

Beer, Ferdinand P. 2011. ESTÁTICA. *MECÂNICA VETORIAL PARA ENGENHEIROS:.* 2011.

Bengio, Yoshua. 2012. Practical Recommendations for Gradient-Based Training of Deep Architectures. 2012.

Bottou, Léon. 2018. Online Learning and Stochastic Approximations. 2018.

Browniee, Jason. 2020. *Deep Learning With Python.* 2020.

Brownlee, Jason. 2019a. A Gentle Introduction to the Rectified Linear Unit (ReLU). *machinelearningmastery.* [Online] 09 de 01 de 2019a. [Citado em: 24 de 01 de 2022.] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

—. 2019b. Machine Learning Mastery. *Loss and Loss Functions for Training Deep Learning Neural Networks.* [Online] 28 de 01 de 2019b. [Citado em: 29 de 01 de 2022.] https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/.

Calôba, Guilherme Marques. 2002. Cooperação entre redes neurais artificiais e técnicas 'clássicas' para previsão de demanda de uma série de vendas de cerveja na Austrália. *scielo.* [Online] 17 de 2002. [Citado em: 27 de 01 de 2022.] https://www.scielo.br/j/pope/a/GvmZQVNPKXxtMBBpkGsSjyH/?lang=pt.

Chollet, François. 2018. *Deep Learning with Python.* 2018.

Data Science Academy. 2021. *Deep Learning Book.* [Online] 2021. [Citado em: 2022 de 01 de 28.] https://www.deeplearningbook.com.br/cross-entropy-cost-function..

—. 2020. Capítulo 4 – O Neurônio, Biológico e Matemático. *Deep Learning Book.* [Online] 2020. [Citado em: 25 de 01 de 2022.] https://www.deeplearningbook.com.br/o-neuronio-biologico-e-matematico/.

—. 2022. O Neurônio, Biológico e Matemático. *Deep Learning Book.* [Online] 2022. [Citado em: 25 de 03 de 2022.] https://www.deeplearningbook.com.br/o-neuronio-biologico-e-matematico/.

Dive into Deep Learning. 2020. Dive Into Deep Learning. *Stochastic Gradient Descent.* [Online] 2020. [Citado em: 01 de 29 de 2022.] https://d2l.ai/chapter_optimization/sgd.html.

Facure, Matheus. 2017. [Online] 12 de 07 de 2017. [Citado em: 28 de 01 de 2022.] https://matheusfacure.github.io/2017/07/12/activ-func/.

Foote, Keith D. 2021. A Brief History of Machine Learning. *dataversity.* [Online] 2021. [Citado em: 10 de 06 de 2022.] https://www.dataversity.net/a-brief-history-of-machine-learning/#.

Géron, Aurélien. 2018. *Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow.* 2018.

GIL, FELIPE DE SOUZA LOURENÇO. 2015. ANÁLISE DA QUALIDADE DE MALHA DE ELEMENTOS. 2015.

Gomes, Pedro César Tebaldi. 2019. Regressão Linear: entenda como utilizar. *DataGeeks.* [Online] 2019. [Citado em: 20 de 03 de 2022.] https://www.datageeks.com.br/regressao-

linear/#:~:text=Dentro%20dessa%20equa%C3%A7%C3%A3o%2C%20o%20algoritmo,esperado%20que%20a%20demanda%20diminua..

Hochreiter, Sepp. 1997. LONG SHORT-TERM MEMORY. 1997.

Hutter, Frank. 2014. An Efficient Approach for Assessing Hyperparameter Importance. *Proceedings of Machine Learning Research.* [Online] 2014. [Citado em: 29 de 01 de 2022.] http://proceedings.mlr.press/v32/hutter14.html.

IBEC. 2018. Como a inteligência artificial pode ser utilizada na engenharia? *ibecensino.* [Online] 2018. [Citado em: 25 de 01 de 2022.] https://ibecensino.org.br/blog/dicas/como-a-inteligencia-artificial-pode-ser-utilizada-na-engenharia/.

IBM. 2020. Artificial Intelligence (AI). *IBM.* [Online] 03 de 06 de 2020. [Citado em: 27 de 01 de 2022.] https://www.ibm.com/uk-en/cloud/learn/what-is-artificial-intelligence.

Krishnan, Sandhya. 2021. How to determine the number of layers and neurons in the hidden layer? *Medium.* [Online] 8 de 09 de 2021. [Citado em: 28 de 01 de 2022.] https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3.

Kwon, Yong W. 2000. *The Finite Element Method using MATLAB.* New York : CRC Press, 2000.

LeCun, Y. 1989. Backpropagation applied to handwritten zip code recognition. 1989.

Lopez, Rafael Holdorf. 2013. Introdução à área de otimização matemática especialmente a sua aplicação em problemas de engenharia. 2013.

Mack, David. 2018. [Online] 09 de 04 de 2018. [Citado em: 27 de 01 de 2022.] https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2.

Mcculloch, Warren S. 1943. A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY. 1943.

Nguyen, Long C. 2020. Deep learning for computational structural optimization. 2020.

Nicoletti, Renato Silva. 2018. EQUACIONAMENTO DO MÉTODO DOS ELEMENTOS FINITOS ATRAVÉS DA ÁLGEBRA LINEAR E SUAS APLICAÇÕES NA ANÁLISE ESTRUTURAL. 2018.

Nishimoto, Bruno Eidi. 2018. Principais conceitos por trás do Machine Learning. *Medium.* [Online] 2018. [Citado em: 18 de 03 de 2022.]

NIST. 2010. https://www.nist.gov/programs-projects/face-recognition-grand-challenge-frgc. *NIST.* [Online] 2010. [Citado em: 12 de 09 de 2022.] https://www.nist.gov/programs-projects/face-recognition-grand-challenge-frgc.

Ribeiro, Andreza. 2020. Inteligência Artificial: quais as suas aplicações na engenharia? *Engenharia360.* [Online] 2020. [Citado em: 26 de 01 de 2022.] https://engenharia360.com/inteligencia-artificial-na-engenharia/.

Ruder, Sebastian. 2016. An overview of gradient descent optimization algorithms. *ruder.* [Online] 19 de 01 de 2016. [Citado em: 28 de 01 de 2022.] https://ruder.io/optimizing-gradient-descent/.

Scipy. 2022. scipy.optimize.differential_evolution . *Documentação SciPy.* [Online] 2022. [Citado em: 18 de 05 de 2022.] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html.

Sharma, Avinash. 2017. Understanding Activation Functions in Neural Networks. *Medium.* [Online] 2017. [Citado em: 15 de 03 de 2022.] https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

Storn, R. e Price, K. 1997. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization.* 1997.

Wikipédia. 2021. Coeficiente de determinação. *Wikipédia.* [Online] 15 de 08 de 2021. [Citado em: 2022 de 01 de 29.] https://pt.wikipedia.org/wiki/Coeficiente_de_determina%C3%A7%C3%A3o.

Zhang, Aston. 2020. *Dive into Deep Learning.* 2020.