O PROBLEMA XOR USANDO REDES NEURAIS ARTIFICIAIS EM PYTHON E COM FUNÇÃO DE ATIVAÇÃO TANGENTE HIPERBÓLICA

THE XOR PROBLEM USING ARTIFICIAL NEURAL NETWORK IN PYTHON AND WITH HYPERBOLIC TANGENT AS ACTIVATION FUNCTION

Lino Timóteo Conceição de Brito
Discente das Faculdades Integradas de Bauru, Bauru, SP, Brasil; linotim@hotmail.com

Ronaldo César Dametto
Docente e Coordenador do Curso Ciência da Computação das Faculdades Integradas de Bauru, Bauru, SP, Brasil; rdametto@gmail.com

RESUMO
O presente trabalho mostra o problema XOR sendo resolvido por uma rede neural em Python. Para isso usa-se uma função tangente hiperbólica como função de ativação. Há uma série de passos para se analisar a rede neural, desde o cálculo das entradas até o cálculo da saída. Existem muitas funções que podem ser usadas em redes neurais, aqui é usada a função tangente hiperbólica e essa mesma função é derivada para se calcular o erro, afim de que este seja mínimo e o resultado final possa estar muito próximo do desejado. O método da descida do gradiente faz uma direção de busca para encontrar o erro mínimo. O gradiente é uma derivada parcial e como toda derivada, esta pode calcular o valor mínimo ou máximo de uma função. Redes neurais podem ser usadas para a aprendizagem de máquinas com um processo iterativo ou repetições (epochs). É feita também simulações com números diferentes de repetições e assim o erro vai diminuindo e o valor da saída XOR vai se aproximando do desejado, isto é, a rede neural vai aprendendo.
Palavras-chave: Linearmente separável; Linearmente não separável; Descida do gradiente, Derivada.

ABSTRACT
The present article shows the XOR problem being solved by a neural network using Python. For that it is used a hyperbolic tangent function as activation function. There are some steps to analyse the neural network, since the input calculations until the output calculations. There are many functions witch can be used in neural networks, herein is used the hyperbolic tangent function and this same function is derived to calculate the error, for this error be minimal and the output can be very near of the desired result. The gradient descendent searchs for the minimal error. The gradiente is a partial derivative and like all derivatives, can be used to find the minimal or maximal of a function. Neural networks can be used for machine learning with iterative process or repetitions (epochs). It is also made simulations with different numbers of iterations and thus the error decreases and the output XOR gets the desired result and the neural network learns.
Keywords: Linearly separable; Linearly non separable; Gradient descendente; Derivative.

1 INTRODUÇÃO

O emprego de redes neurais tem sido uma tentativa eficiente de imitar o cérebro humano. Um estudo mais detalhado sobre redes neurais pode ser feito em [1] e [2]. O uso de redes neurais resolve problemas matemáticos e de engenharia com precisão muitas vezes satisfatórias e podem ser utilizadas para aprendizagem de máquinas (Chapmann, 2017). "Remarkably, the resultant nonlinearities often produce soft XOR functions, consistent with recent experimental observations about interactions between inputs in human cortical neurons." (Kim, 2022).

O objetivo deste trabalho é treinar uma rede neural artificial para resolver o problema XOR. Para isso são usadas no código (algoritmo) várias repetições (epochs). À medida que se aumenta o número de iterações método do gradiente vai fazendo uma busca para encontrar o erro mínimo, assim a saída vai se aproximando àquela esperada com a diminuição gradativa do erro. Esse erro é uma derivada da própria função de ativação, que no caso deste trabalho, é a função tangente hiperbólica.

O problema XOR difere do problema OR e AND. O problema XOR, como será visto mais adiante, não pode ter seus zeros e uns separados por uma linha reta, o que não acontece com o problema OR e AND que é possível separar os zeros dos uns com uma linha reta. Assim o perceptron de uma camada se adapta bem para o problema AND e OR. Porém para o problema XOR, o perceptron de uma camada começa a apresentar problemas, como apresentado em [5]. Então é necessário acrescentar uma camada intermediária, chamada de camada escondida (Hidden layer).

Baseado em [5] há uma sequência de passos para se calcular o processo iterativo, atualizando-se os pesos e minimizando o erro. A rede neural pode ser comparada com o neurônio biológico, onde os pesos são as sinapses.

Foi utilizada a biblioteca numpy do python para fazer o cálculo de matrizes. Foi usada como função de ativação a tangente hiperbólica, embora exista uma grande variedade de funções na literatura (Granatyr, 2022). A derivada da função hiperbólica é apresentada no código neste trabalho. Um estudo sobre funções hiperbólicas e suas derivadas pode ser encontrado em [4]. Aqui a derivada da função tangente hiperbólica pode ser obtida pela fórmula da derivada da divisão de duas funções (Leithold, 1994). A sequência de passos do algoritmo pode ser encontrada em [5]. O valor inicial dos pesos foi calculado como em [5].

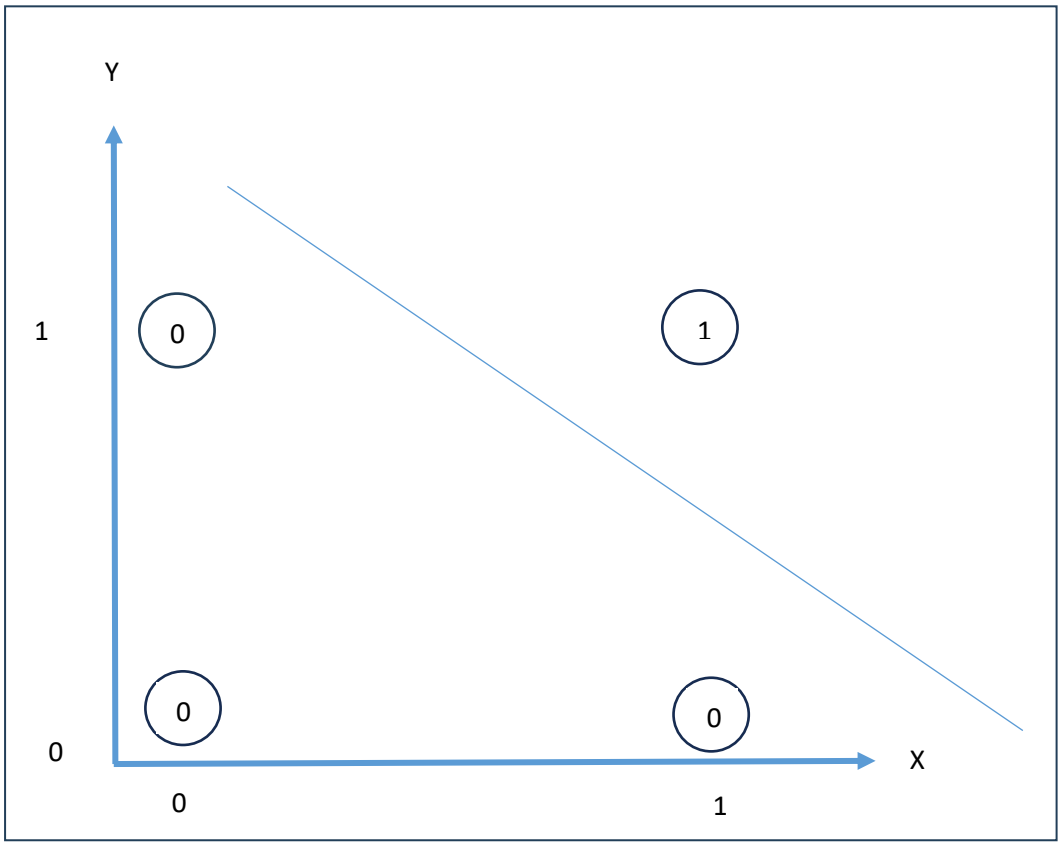
Neste presente trabalho são feitas simulações com 100, 1000 e 10000 repetições, onde o gradiente vai minimizando o erro e a saída vai se tornando mais próxima da desejada. A

medida que aumenta o número de repetições a rede neural vai aprendendo e o erro pode ser minimizado com um número adequado de iterações (epochs). Para a derivada e posteriormente encontrar o gradiente é usada a derivada da relação entre duas funções e também a regra da cadeia.

2 FUNDAMENTAÇÃO TEÓRICA

O problema AND e seu gráfico lógico é mostrado na figura 1. Repare que é possível separar os zeros dos uns com uma única reta. Desse modo, o perceptron de uma camada, que também é demonstrado em [5], funciona bem para esse tipo de problema.

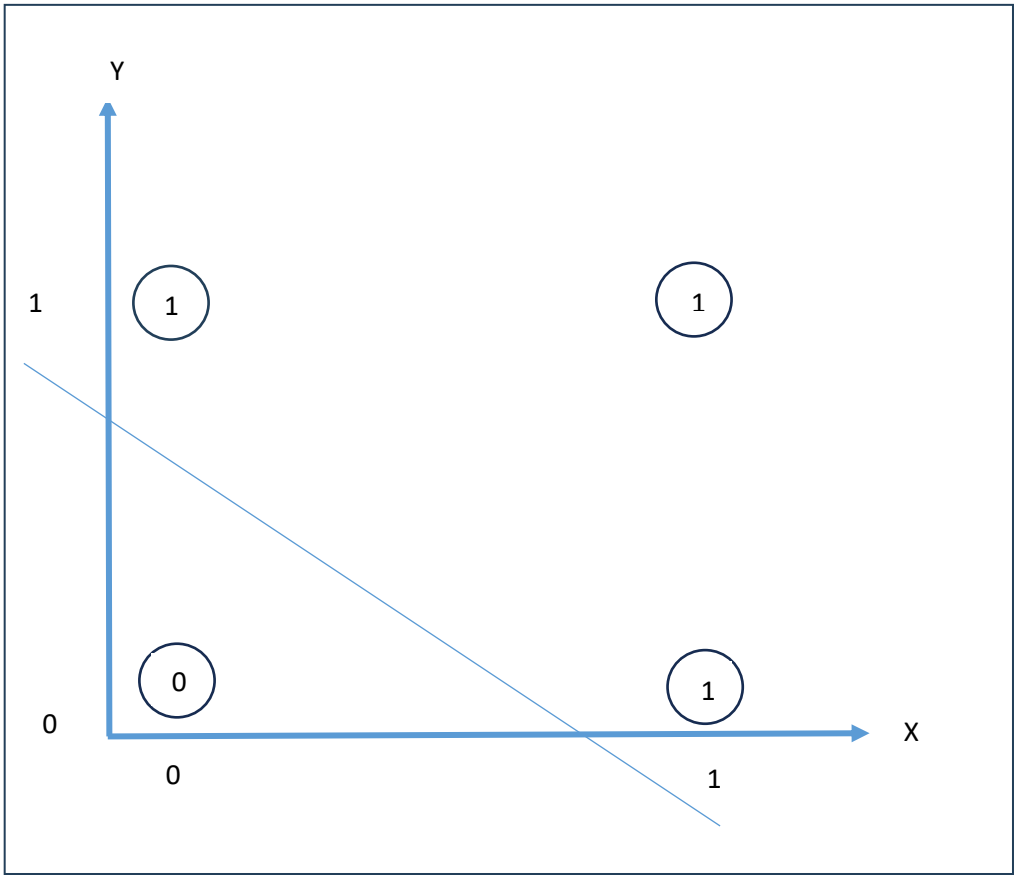
Figura 1 - O problema lógico AND. É possível separar os zeros dos uns com uma única reta



Fonte: Dados da Pesquisa

Algo semelhante ocorre com o problema lógico OR. Esse problema também pode ter seus zeros e uns separados por uma única reta e de modo semelhante ao problema AND. É o que ocorre na figura 2 mostrada a seguir.

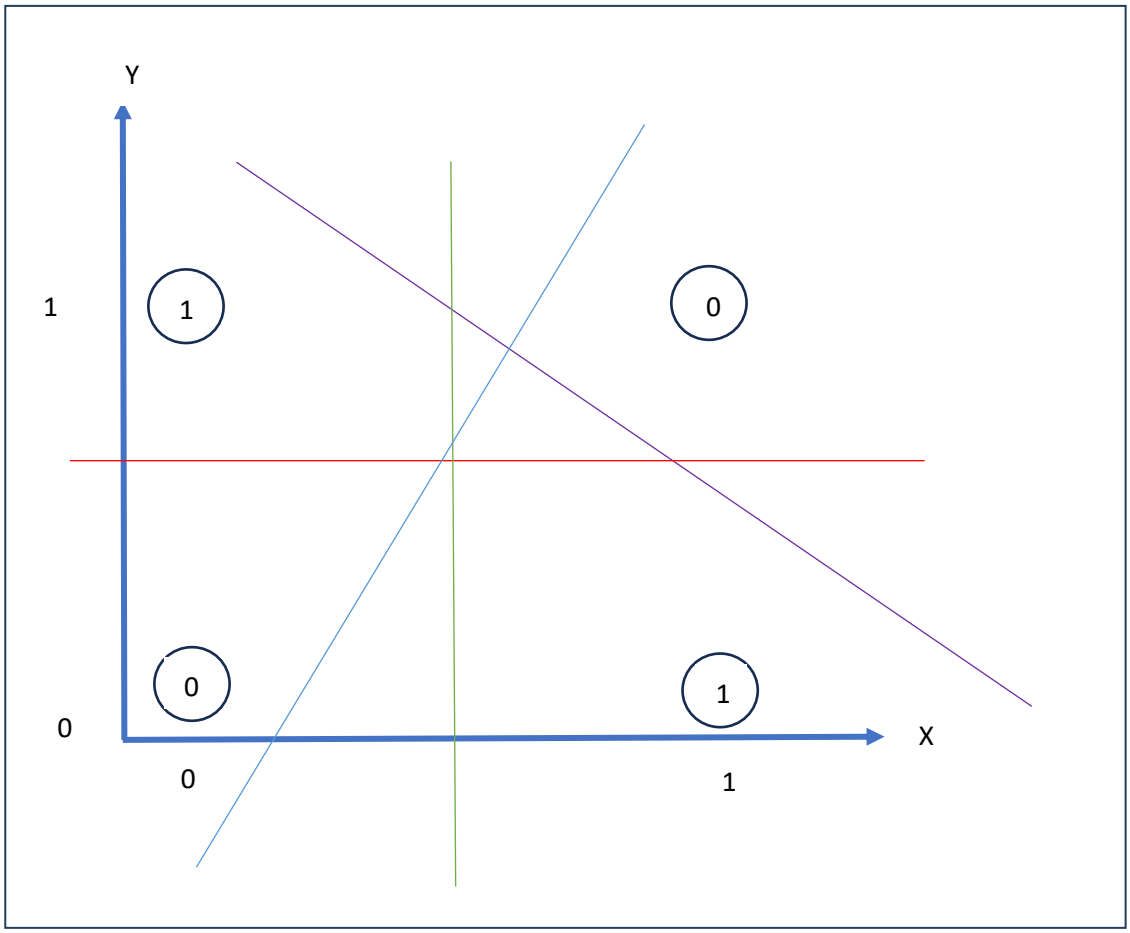
Figura 2 - O problema lógico OR. Repare que é possível ainda separar os zeros dos uns com apenas uma reta



Fonte: Dados da Pesquisa

No problema XOR ocorre algo diferente do que foi exposto antes. Agora não é mais possível separar os zeros dos uns com uma única reta. Assim, como exposto em [5], o perceptron de uma camada já não satisfaz a solução do problema e é necessário adicionar uma camada intermediária (camada oculta). Isso é caracterizado como um problema linearmente não separável, como ilustrado na figura 3. Repare que nessa figura várias retas são traçadas, mas não resolve o problema.

Figura 3 - O problema lógico XOR. Caracteriza um problema linearmente não separável

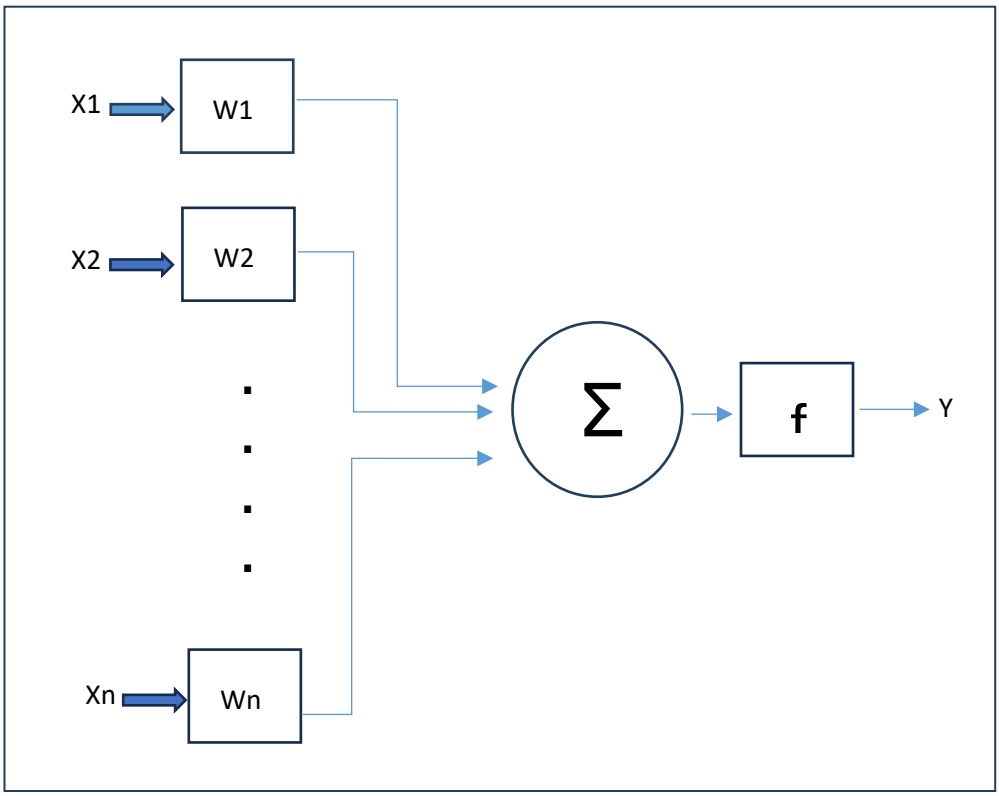


Fonte: Dados da Pesquisa

A figura 4 representa uma modelagem matemática do perceptron de uma camada. *X* são as entradas e *W* são os pesos. Cada entrada é multiplicada pelo seu respectivo peso, cada produto passa por um somador e essa soma é o *x* da função de ativação e assim o valor do neurônio é caliculado. *Y* representa a saída que é calculada através da função de ativação que no caso deste trabalho é a função tangente hiperbólica.

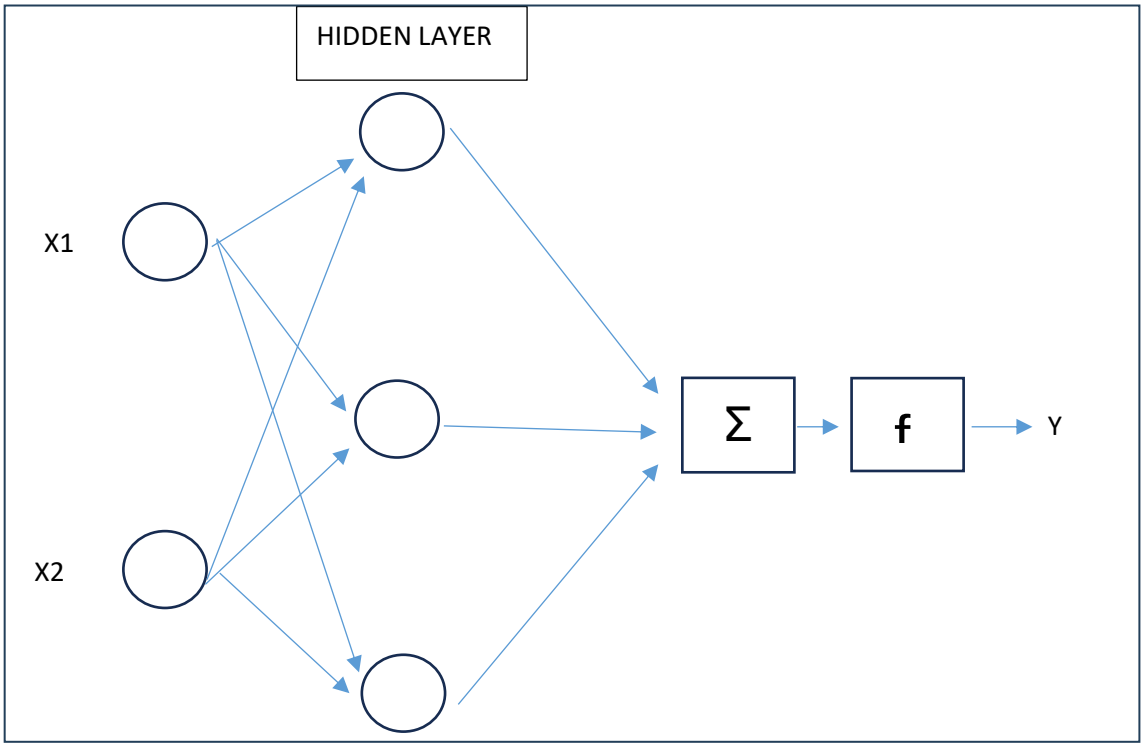
Para o problema XOR é necessário adicionar uma camada oculta como mostrado na figura 5. A adição da camada oculta torna os cálculos mais complexos. Problemas mais complicados envolvem um número maior de camadas. Aqui é apresentado um problema simples com camada oculta.

Figura 4 - Modelagem matemática para o perceptron de uma camada



Fonte: Dados da Pesquisa

Figura 5 - O perceptron multicamada



Fonte: Dados da Pesquisa

3 MATERIAIS E MÉTODOS

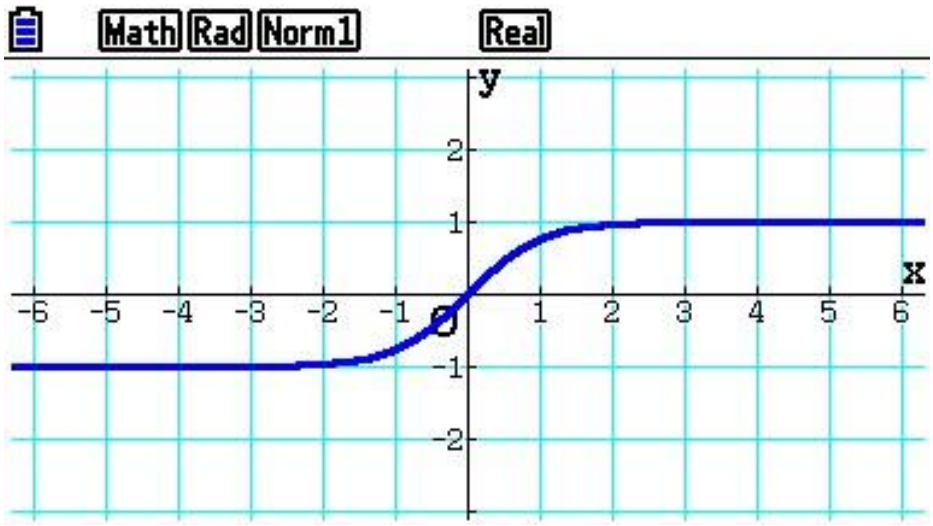
Em redes neurais existem muitas funções matemáticas para resolver problemas, como apresentado em [5]. O primeiro passo para treinar a rede neural com problema XOR é a escolha da função de ativação. Em [5] é usada a função sigmoide e aqui é usada a função tangente hiperbólica e o conjunto de passos também pode ser encontrado na mesma referência. O passo seguinte é calcular o processo chamado de FeedFoward. Esse processo faz os cálculos desde a entrada até a saída, por isso o nome.

É necessário em seguida calcular o erro, para que este seja mínimo e a saída seja como esperada, ou pelo menos o mais próximo disso. Esse erro é uma derivada, assim é útil a derivada da função de ativação (tangente hiperbólica). Para isso usa-se o método da descida do gradiente, que vai buscando o valor mínimo do erro a cada repetição (processo iterativo). Em seguida tem-se o processo BackPropagation que faz um cálculo de trás para frente, atualizando os pesos a cada iteração.

3.1 – A função tangente hiperbólica

A figura 6 mostra o gráfico da função tangente hiperbólica, plotada em uma calculadora gráfica. A sua derivada segue um processo semelhante à derivada da relação de duas funções, como apresentado adiante.

Figura 6 - Gráfico da função tangente hiperbólica



Fonte: Dados da Pesquisa

3.2 – Passos para a derivação da função tangente hiperbólica

A fincão tangente hiperbólica á dada por:

$$tgh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

A derivada do quociente de duas funções ($h(x) = \frac{f(x)}{g(x)}$), segundo [4], é dada por:

$$h'(x) = \frac{g(x)f'(x) - f(x)g'(x)}{g(x)^2}$$

Fazendo-se *f(x)* como numerador $(e^x - e^{-x})$ e *g(x)* como denominador $(e^x + e^{-x})$, e usando a regra da cadeia (a derivada de e^x é a própria função e a derivada de e^{-x} vale - e^{-x}), tem-se que:

$$f'(x) = (e^x + e^{-x})$$

De modo similar:

$$g'(x) = (e^x - e^{-x})$$

Agora substituindo em $h'(x)$:

$$h'(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

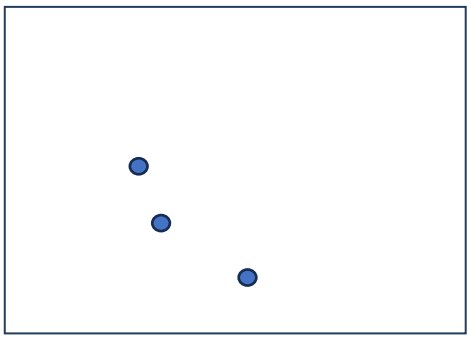
Após um certo algebrismo, a derivada da função tangente hiperbólica pode ser dada por:

$$tgh'(x) = \frac{4}{(e^x + e^{-x})^2}$$

3.3 – O método do gradiente

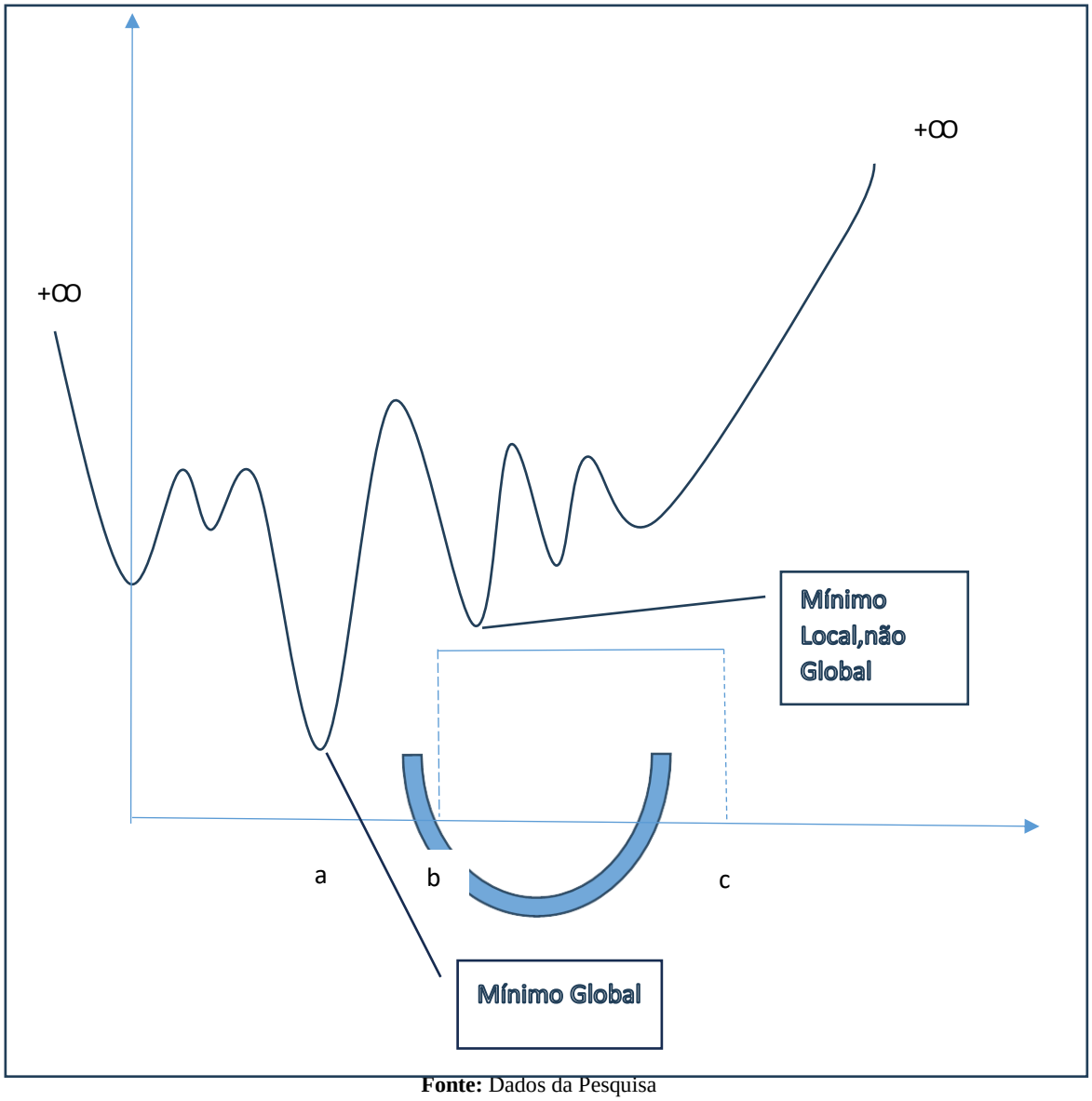
O gradiente faz uma direção de busca pelo ponto mínimo de uma função. Um exemplo bom para visualizar isso é imaginar uma bolinha sendo solta em uma tigela, como mostrado na figura 7. A bolinha fica oscilando para cima e para baixo até que repousa no fundo da tigela, isto é, quando alcança o ponto mínimo. É como se a cada iteração o valor do gradiente estivesse mais próximo do mínimo. Um problema que pode ocorrer durante a busca é que o gradiente pode ficar retido em um mínimo local e poder parecer um mínimo global. A figura 8 representa uma função hipotética para visualizar a diferença entre um mínimo local e u mínimo global.

Figura 7 - Exemplo para visualizar a busca do ponto mínimo



Fonte: Dados da Pesquisa

Figura 8 - Gráfico de uma função hipotética para visualizar um mínimo local e o mínimo global



Fonte: Dados da Pesquisa

Na figura 8, o intervalo entre *b* e *c* contém um mínimo relativo (mínimo local) que não representa o valor mínimo global da função. Esse ponto é mínimo somente neste intervalo. Repare que o ponto *a* representa o mínimo global, isto é, mínimo absoluto para todo domínio, desde menos infinito até o mais infinito.

3.4 – Passos e algoritmo

A sequência de passos para calcular a rede neural referida neste trabalho é apresentada a seguir, de acordo com [5].

1 – Escolha da função de ativação

2 – Processo Feed Foward

3 – Função erro (resposta correta – resposta calculada)

4 – Descida do Gradiente

5 – Back Propagation

Taxa de aprendizagem

Cálculo do delta

Momento

O algoritmo implementado é exposto a seguir. Duas funções foram implementadas, a função tangente hiperbólica para calcular o valor do neurônio, assim como a função derivada para calcular o erro. Os pesos iniciais são aleatórios e dependendo da precisão do "chute inicial" o algoritmo pode convergir mais rapidamente ou não. Os pesos são atualizados a cada ietração, tanto os pesos da camada de entrada como da camada de saída. As entradas multiplicadas pelo respectivo peso são somadas e essa soma será o *x* da função de ativação.

```
# Funcão Tangente Hiperbólica
import numpy as num
def tgh(sum):
   return (num.exp(sum) - num.exp(-sum))/(num.exp(sum) + num.exp(-sum))
# Derivada da função tangente Hiperbólica
def derivative(x):
   return 4/((num.exp(x)+num.exp(-x))*(num.exp(x)+num.exp(-x)))
input = num.array([[0,0],
                   [0,1],
```

```
                [1,0],
                [1,1]])
output = num.array([[0],[1],[1],[0]])
w0 = num.array([[-0.424, -0.74, -0.961],
            [0.358, -0.577,-0.469]])
w1 = num.array([[-0.017],[-0.893],[0.148]])
   # Numero de rodadas
epochs = 1000
   # Taxa de aprendizagem e momento
rateLearning = 0.3
momento = 1

for i in range(epochs):
   layerIn = input
   sumSinap0 = num.dot(layerIn,w0)
   hiddenLayer = tgh(sumSinap0)
   sumSinap1 = num.dot(hiddenLayer,w1)
   layerOut = tgh(sumSinap1)
   errorHiddenLayer = output - layerOut
   absAverage = num.mean(num.abs(errorHiddenLayer))
   print("O erro vale: " + str(absAverage))
   outputDerivative = derivative(layerOut)
   deltaOut = errorHiddenLayer*outputDerivative
   w1Trans = w1.T
   deltaOutXw = deltaOut.dot(w1Trans)
   deltaHiddenLayer = deltaOutXw*derivative(hiddenLayer)
   hiddenLayerTrans = hiddenLayer.T
   newWeight1 = hiddenLayerTrans.dot(deltaOut)
   w1 = (w1*momento) + (newWeight1*rateLearning)
   inputLayerTrans = layerIn.T
   newWeight0 = inputLayerTrans.dot(deltaHiddenLayer)
   w0 = (w0*momento) + (newWeight0*rateLearning)
```

3.5 – Código detalhado

O código foi implementado pela linguagem de programação Python, pelo Spyder. Foi usada a biblioteca numpy para fazer cálculos entre matrizes. O trecho as seguir representa a implementação da função tangente hiperbólica, assim como a implementação de sua derivada, para o cálculo do erro.

```
# Funcão Tangente Hiperbólica
import numpy as num
def tgh(sum):
    return (num.exp(sum) - num.exp(-sum))/(num.exp(sum) + num.exp(-sum))
# Derivada da função tangente Hiperbólica
def derivative(x):
    return 4/((num.exp(x)+num.exp(-x))*(num.exp(x)+num.exp(-x)))
```

O trecho a seguir mostra os valores iniciais para a entrada e saída e os pesos para a camada de entrada e camada oculta para a saída.

```
input = num.array([[0,0],
                   [0,1],
                   [1,0],
                   [1,1]])
output = num.array([[0],[1],[1],[0]])
w0 = num.array([[-0.424, -0.74, -0.961],
                [0.358, -0.577,-0.469]])
                  w1 = num.array([[-0.017],[-0.893],[0.148]])
```

No próximo trecho é definida o momento, que como em [5] pode fazer como que o método do gradiente não fique retido em um mínimo local, às vezes funciona, ás vezes não. Também é definida a taxa de aprendizagem que dependendo do valor pode fazer o código convergir mais rápido ou não, mas pode perder o mínimo global com mais facilidade dependendo do valor. O comando “dot” é um comando especial do numpy que faz o produto entre matrizes, diretamente sem recorrer ao uso de vários “for”.

```
# Numero de rodadas
epochs = 1000
# Taxa de aprendizagem e momento
rateLearning = 0.3
momento = 1
for i in range(epochs):
layerIn = input
sumSinap0 = num.dot(layerIn,w0)
hiddenLayer = tgh(sumSinap0)
      sumSinap1 = num.dot(hiddenLayer,w1)
```

O trecho em seguida mostra o cálculo do erro que é a diferença entra a resposta correta e a resposta calculada.

```
layerOut = tgh(sumSinap1)
errorHiddenLayer = output - layerOut
absAverage = num.mean(num.abs(errorHiddenLayer))
print("O erro vale: " + str(absAverage))
```

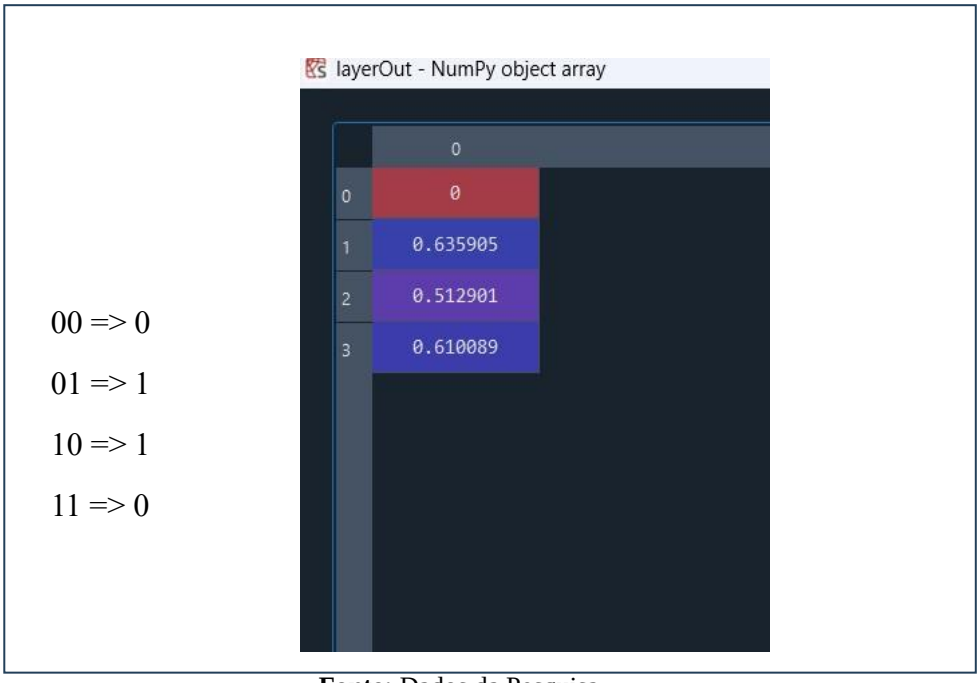
Finalmente, o próximo trecho representa o cálculo dos pesos. O comando “T” em destaque transforma a matriz original em sua transposta. A multiplicação entre matrizes pode apresentar erros de índice, assim é necessário calcular a transposta para que seja possível fazer a multiplicação.

```
outputDerivative = derivative(layerOut)
deltaOut = errorHiddenLayer*outputDerivative
w1Trans = w1.T
deltaOutXw = deltaOut.dot(w1Trans)
deltaHiddenLayer = deltaOutXw*derivative(hiddenLayer)
hiddenLayerTrans = hiddenLayer.T
newWeight1 = hiddenLayerTrans.dot(deltaOut)
w1 = (w1*momento) + (newWeight1*rateLearning)
inputLayerTrans = layerIn.T
newWeight0 = inputLayerTrans.dot(deltaHiddenLayer)
w0 = (w0*momento) + (newWeight0*rateLearning)
```

4 RESULTADOS E DISCUSSÃO

Foram feitas três simulações para a rede neural em estudo. Uma para 100 (cem) iterações (epochs), uma para 1000 (mil) iterações e uma para 10000 (dez mil) iterações, afim de validar a rede neural para o problema XOR. Na figura 9 é representado a saída correta e a saída simulada para o problema.

Figura 9 – Saída simulada para cem iterações



Fonte: Dados da Pesquisa

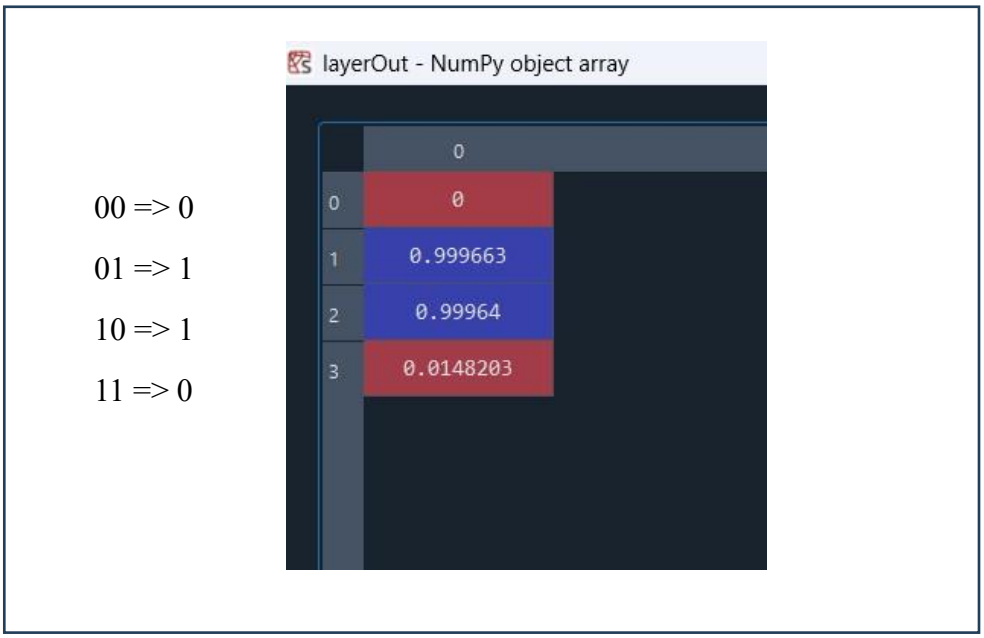
Na figura 9 a saída ainda ficou insatisfatória devido ao valor elevado do erro. Isso indica que são necessárias mais repetições e é feito na figura 10. Fazendo agora 1000 (mil) repetições, a rede neural ofereceu um resultado melhor com em erro bem menor.

Figura 10 – Saída simulada para mil iterações

Fonte: Dados da Pesquisa

Para melhorar ainda mais a saída simulada, foi feita dez mil iterações, como apresentado na figura 11. Repare que a medida que aumenta o número de iterações, a rede neural vais aprendendo e a saída desejada pode se aproximar cada vez mais da saída ideal.

Figura 11 – Simulação para dez mil iterações



Fonte: Dados da Pesquisa

5 CONCLUSÕES

1 – A rede neural aprende à medida que se aumenta o número de iterações. O erro vai diminuindo e a saída desejada se torna mais próxima da saída ideal.

2 – Esse problema resolvido pode servir de base para trabalhos futuros, como a inserção de mais de uma camada oculta para melhorar a performance da rede neural.

3 – Também serve de base para inserir um número maior de entradas, o que torna o problema mais complexo e com maior número de registros.

REFERÊNCIAS

[1] IOVINE, John. **Understanding Neural Networks**: The Experimenter‘s Guide. [*S. l.*], 2012. *E-book*.

[2] CHAPMANN, Joshua. **Building neural Networks**: Introduction to Artificial Neurons, Backpropagation Algorithms and Multilayer Feedforward Neural Networks. [*S. l.*], 2017. *E-book*.

[3] KIM, Juhyeon; ORHAN, Emin; YOON, Kijung; PITKOW, Xaq. Two-Argument Activation Functions Learn Soft XOR Operations Like Cortical Neurons. IEEE Accsess, [*S. l.*], p. 1-10, 30 maio 2022. Disponível em: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9785635. Acesso em: 20 jun. 2023.

[4] LEITHOLD, Louis. **O Cálculo com Geometria Analítica**. 3. ed. São Paulo: HARBRA ltda, 1994.

[5] **Redes Neurais Artificiais em Python. Direção**: Jones Granatyr. [*S. l.*: *s. n.*], 2022. Disponível em: https://www.udemy.com/course/redes-neurais-artificiais-em-python/learn/lecture/8048696#overview. Acesso em: 16 jun. 2023.