Universidade Federal de Santa Catarina

Centro Tecnológico

Departamento de Automação e Sistemas

Treinamento de redes neurais multi-camadas

Inteligência Artificial Aplicada a Automação

Guilherme Henrique Ludwig
Juliano Ricardo da Silva
Prof. Dr. Eric Antonelo

Florianópolis - SC

Março de 2022

Conteúdo

1 Estrutura da Rede Neural

Este trabalho tem como objetivo a implementação do método da retropropagação (*backpropagation*) para o cálculo do gradiente da função de custo com relação aos pesos de uma rede neural multi-camadas.

Antes de iniciar a implementação do algoritmo em *Python*, é necessário definir a estrutura da rede neural multi-camadas. Neste caso, tem-se uma rede com padrão dimensional de duas entradas, uma saída e uma camada oculta formada por dois neurônios. A figura 1 mostra graficamente a estrutura da rede:

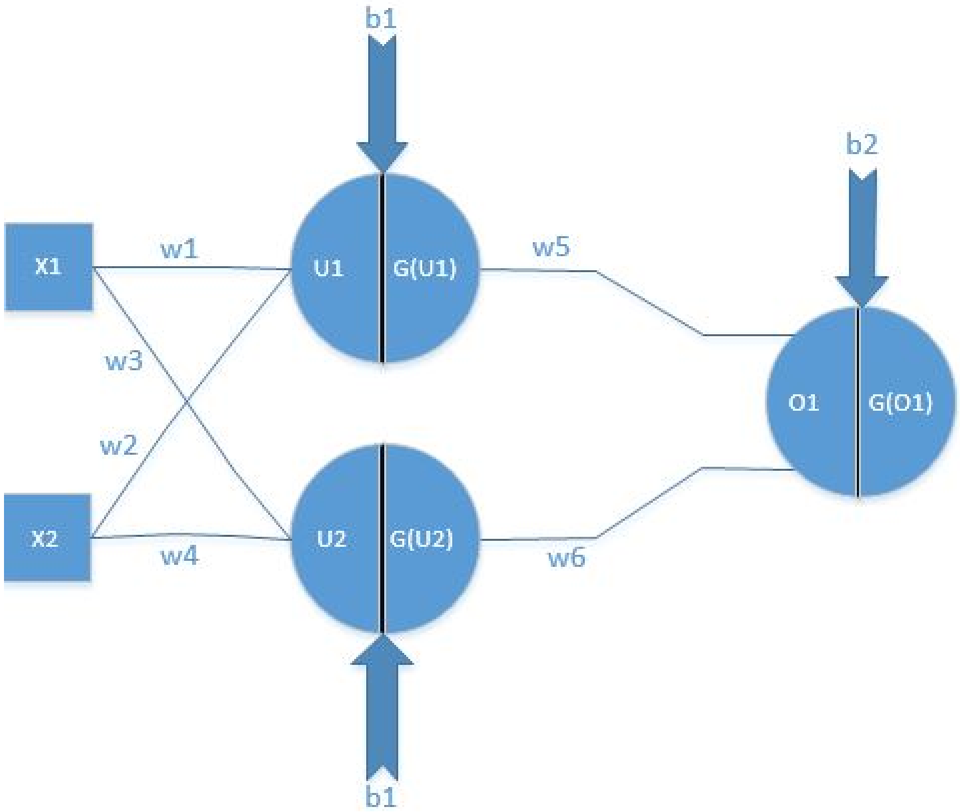


Figura 1: Estrutura da Rede Neural Multicamadas.

Sendo **x1,x2** as entradas, **o1** a saída, **b1,b2** os bias e **w1,w2,...,w6** os pesos sinápticos.

Após a implementação do algoritmo, a rede neural deverá ser utilizada para realizar a classificação binária de um conjunto de exemplos de treinamentos disponibilizados no arquivo *classification2.txt*.

2 Implementação do Algoritmo

A implementação do algoritmo foi realizada seguindo-se os passos indicados pelo enunciado do trabalho. A descrição do funcionamento do código estará disponível no vídeo <https://www.youtube.com/watch?v=fifLA5gFrds>. O código usado para o treinamento do conjunto de dados *classification2.txt* foi:

```python
#bibliotecas
import math
import random
import pandas as pd
import matplotlib.pyplot as plt
```

```python
import numpy as np

###########################################################################

#Extracao de todos os exemplos (entradas) do conjunto de treinamento
df = pd.read_csv(r"C:\Users\julia\Documents\classification2.txt")
print(df)

xs = df.iloc[:,:-1].values #2 primeiras colunas
ys = df.iloc[:,-1].values  #ultima coluna


#plote dos exemplos a serem separados
pos , neg = (ys==1).reshape(117,1) , (ys==0).reshape(117,1)
plt.scatter(xs[pos[:,0],0], xs[pos[:,0],1], c="r", marker="+")
plt.scatter(xs[neg[:,0],0], xs[neg[:,0],1], marker="o", s=10)
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend(["Accepted", "Rejected"], loc=0)
plt.show()

#valores das entradas x1, x2 e yout

x1=[0.051267,-0.092742,-0.21371,-0.375,-0.51325,-0.52477,-0.39804,-0.30588,
0.016705, 0.13191,0.38537, 0.52938,0.63882,0.73675,0.54666,0.322,0.16647,
-0.046659,-0.17339,-0.47869,-0.60541,-0.62846,-0.59389,-0.42108,-0.11578,
0.20104,0.46601,0.67339,-0.13882,-0.29435,-0.26555, -0.16187,-0.17339,
-0.28283,-0.36348,-0.30012,-0.23675,-0.06394,0.062788,0.22984,0.2932,
0.48329,0.64459,0.46025,0.6273,0.57546,0.72523,0.22408,0.44297,0.322,
0.13767,-0.0063364,-0.092742,-0.20795,-0.20795,-0.43836,-0.21947,-0.13882,
0.18376,0.22408,0.29896,0.50634,0.61578,0.60426,0.76555,0.92684,0.82316,
0.96141,0.93836,0.86348,0.89804,0.85196,0.82892,0.79435,0.59274,0.51786,
0.46601,0.35081,0.28744,0.085829,0.14919,-0.13306,-0.40956,-0.39228,
-0.74366,-0.69758,-0.75518,-0.69758,-0.4038,-0.38076,-0.50749,-0.54781,
0.10311,0.057028,-0.10426,-0.081221,0.28744,0.39689,0.63882,0.82316,0.67339,
1.0709,-0.046659,-0.23675,-0.15035,-0.49021,-0.46717,-0.28859,-0.61118,
-0.66302,-0.59965,-0.72638,-0.83007,-0.72062,-0.59389,-0.48445,-0.0063364,
0.63265]

x2=[0.69956,0.68494,0.69225,0.50219,0.46564,0.2098,0.034357,
-0.19225,-0.40424,-0.51389,-0.56506,-0.5212,-0.24342,-0.18494,0.48757,
0.5826,0.53874,0.81652,0.69956,0.63377,0.59722,0.33406,0.005117,
-0.27266,-0.39693,-0.60161,-0.53582,-0.53582,0.54605,0.77997,0.96272,
0.8019,0.64839,0.47295,0.31213,0.027047,-0.21418,-0.18494,-0.16301,
-0.41155,-0.2288,-0.18494,-0.14108,0.012427,0.15863,0.26827,0.44371,
```

```
0.52412,0.67032,0.69225,0.57529,0.39985,0.55336,0.35599,0.17325,0.21711,
-0.016813,-0.27266,0.93348,0.77997,0.61915,0.75804,0.7288,0.59722,0.50219,
0.3633,0.27558,0.085526,0.012427,-0.082602,-0.20687,-0.36769,-0.5212,
-0.55775,-0.7405,-0.5943,-0.41886,-0.57968,-0.76974,-0.75512,-0.57968,
-0.4481,-0.41155,-0.25804,-0.25804,0.041667,0.2902,0.68494,0.70687,0.91886,
0.90424,0.70687,0.77997,0.91886,0.99196,1.1089,1.087,0.82383,0.88962,
0.66301,0.64108,0.10015,-0.57968,-0.63816,-0.36769,-0.3019,-0.13377,
-0.060673,-0.067982,-0.21418,-0.41886,-0.082602,0.31213,0.53874,0.49488,
0.99927,0.99927,-0.030612]

y_out=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0]

custo = []

##########################################################
#Multi layer Perceptron - Rede Neural

#inicializando a rede neural
def create_network(i, h, o):
    hidden  = []
    outputs = []
    for j in range(h): hidden.append(1)
    for k in range(o): outputs.append(1)
    return i, hidden, outputs

#camada de entrada, camada oculta (dois neuronios), camada de saida (um
    neuronio)
net = create_network([x1[0],x2[0]],2,1)
print("estrutura da rede eh:", net)

#lista de pesos
weights = []

#metodo de Xavier para inicializacao randomica dos pesos
def weights_ini(n,m):
    #n = nro de entradas e m= nro de neuronios da camada oculta
    for i in range(n*m+m):
        weights.append(random.uniform(-(1/math.sqrt(n)), 1/math.sqrt(n)))
    print("Pesos: ", weights)
    return weights

```

```
#inicializa aleatoriamente os pesos de acordo com o numero de entradas e
    numero de neuronios da camada oculta
weights = weights_ini(2, len(net[1]))

bias = []
#n eh o numero de layers da rede
def bias_ini(n):
    for i in range(n):
        bias.append(np.random.uniform(0, 1))
    return bias

bias = bias_ini(2)
print("Lista de bias: ", bias_ini(2))


#funcao generica de ativacao dos neuronios da camada oculta
def activation(neuron):
    for j in range(len(neuron)):
            gu.append(1/(1+math.exp(-neuron[j])))
    return gu

#derivada do Etotal em relacao a go
def derivada_go(y, go, i):
    return -(y[i] - go[i])

#derivada do go em relacao a o
def derivada_o(go, i):
    return (2/(np.exp(go[i]) + np.exp(-go[i])))**2

#funcao ativacao camada de saida
def act_tanh(z, i):
    for i in range(len(z)):
            gu.append((np.exp(z[i]) - np.exp(-z[i])) / (np.exp(z[i]) + np.
                exp(-z[i])))
    return gu

#derivada de o em relacao ao peso w
def derivada_w(gu):
    for i in range(len(gu)):
        dw.append(gu[i])
    return dw

#calculando gradiente do w da camada oculta
def gradient_hidden(w, a, dtot):
    for i in range(len(net[1])):
```

```python
            gh.append(w[i+4] - a*dtot[i])
        return gh

#derivada do gu em relacao a u
def derivada_gu(guu, i):
        return guu[i]*(1-guu[i])

#calculando gradiente do w da camada de entrada
def gradient_in(w, a, dtot,j):
        for i in range(len(net[1])):
            ghs.append(w[i+j] - a*dtot[i])
        return ghs

#inicializa iteracao em 0
cont = 0

saida = []

print("Primeiro exemplo", [x1[cont],x2[cont]])


while cont <=  np.size(x1) - 1:

        print("VALOR DO CONTADOR", cont)

        x = [x1[cont],x2[cont]]
        print("X", x)
        y = [y_out[cont]]
        print("Y",y)

        print("***************************************************************")
        print("*******************************NOVA ITERACAO***************")
        print("***************************************************************")

        u  = []
        gu = []

        #propagacao entrada para camada oculta
        for j in range(len(net[1])):
                u.append(np.dot(x, weights[j*2:2*(j+1)]) + bias[0]*1)
        print("Propagacao para camada oculta: ", u)

        g = gu

        #mostra ativacao dos neuronios da camada oculta
```

```python
    print("Ativacao neuronios da camada oculta: ", activation(u))

    o = []

    #propagacao camada oculta para camada de saida
    for j in range(len(net[2])):
        o.append(np.dot(gu, weights[len(net[1])*len(x)+j*2:2*(j+1)+len(net[1])*len(x)]) + bias[1]*1)

    print("Propagacao para camada de saida: ", o)

    gu = []

    go = act_tanh(o, 0)
    print("Ativacao neuronios da camada de saida: ", go)
    saida.append(go)

    g.extend(go)
    print("historico ativacoes", g)

    #calculando erro total
    print("Y cont", y)
    print("Go cont", go[0])
    erro = 0.5*(y - go[0])**2
    custo.append(erro)

    print("Erro total eh: ", erro)


    #Corrigindo os pesos da camada oculta para camada de saida

    dgo = derivada_go(y, go, 0)
    print("Derivada erro em relacao a go1: ", dgo)

    do = derivada_o(go, 0)
    print("Derivada de go em relacao a o: ", do)

    delta1 = dgo*do
    print("Delta 1 eh: ", delta1)

    #seleciono somente as ativacoes da camada oculta
    gu = g[0:len(net[1])]

    dw = [ ]

```

```
    dw = derivada_w(gu)
    print("As derivadas em relacao aos pesos da camada oculta  sao: ", dw)

    dtot = np.dot(delta1, dw)

    print("As derivadas do erro em relacao aos pesos da camada oculta sao:", dtot)

    gh = []  #vetor para guardar os gradientes da camada oculta
    a = 0.1 #taxa de aprendizagem

    #calculando gradiente do w da camada oculta

    grad = gradient_hidden(weights, a, dtot)
    print("Novos pesos w5 e w6 da camada oculta em funcao do gradiente do erro sao: ", grad)

    #vetor dos pesos atualizados
    w_updated = [0, 0, 0, 0, 0, 0]
    w_updated[4] = grad[0]
    w_updated[5] = grad[1]

    print("Pesos atualizados: ", w_updated)

    #corrigindo os pesos camada1->camada oculta

    #derivada do erro em relacao a O1
    de_o1 = dgo*do
    #print("Deo1", de_o1)
    do_gu = weights[4]
    #print("Do gu", do_gu)

    #derivada do gu em relacao a u
    dg_u  = derivada_gu(g, 0)
    #print("Dgu ", dg_u)

    h1 = de_o1*do_gu*dg_u
    print("O valor de h1 eh", h1)

    dw = []
    dw_first = derivada_w(x)

    dtot_w1 = np.dot(h1, dw_first)

    print("Derivadas parciais do erro em relacao a w1 e w2: ", dtot_w1)
```

```
    do_g    = weights[5]
    dg_u2   = derivada_gu(g, 1)

    h2 = de_o1*do_g*dg_u2
    print("O valor de h2 eh: ", h2)

    dtot_w2 = np.dot(h2, dw_first)

    print("Derivadas parciais do erro em relacao a w3 e w4: ", dtot_w2)

    ghs = []

    #calculando gradiente do w da camada de entrada
    grad_2 = gradient_in(weights, a, dtot_w2,0)
    print("Novos pesos w1 e w2 da camada oculta em funcao do gradiente do
        erro sao: ",  grad_2)

    w_updated[0] = grad_2[0]
    w_updated[1] = grad_2[1]


    ghs = []
    grad_3 = gradient_in(weights, a, dtot_w2,2)
    print("Novos pesos w3 e w4 da camada oculta em funcao do gradiente do
        erro sao: ",  grad_3)


    w_updated[2] = grad_3[0]
    w_updated[3] = grad_3[1]

    print("Todos os pesos corrigidos: ", w_updated)

    weights = [] #limpa para receber os atuais
    weights = w_updated  #atualiza pesos
    print("Pesos para a nova iteracao: ", weights)
    g=[]
    cont+=1

#funcao custo ao longo das iteracoes
plt.figure()
x = [f for f in range(118)]
y = [e for e in custo]
plt.plot(x, y, 'm', label = "iters v Cost fcn" )
plt.legend()
```

```
plt.show()

#plot saidas
plt.figure()
x = [f for f in range(118)]
y = [e for e in saida]
plt.scatter(x, y, alpha=0.5)
plt.legend()
plt.show()
```

3 Resultados

Nesta seção serão mostrados os resultados obtidos com a execução do algoritmo. Num primeiro momento, plotamos as entradas da rede de maneira a verificar os valores de entrada treinados para obter 1 na saída da rede e, da mesma forma, os valores de entrada treinados para obter 1 na saída. A figura 2 ilustra a disposição dos dados:

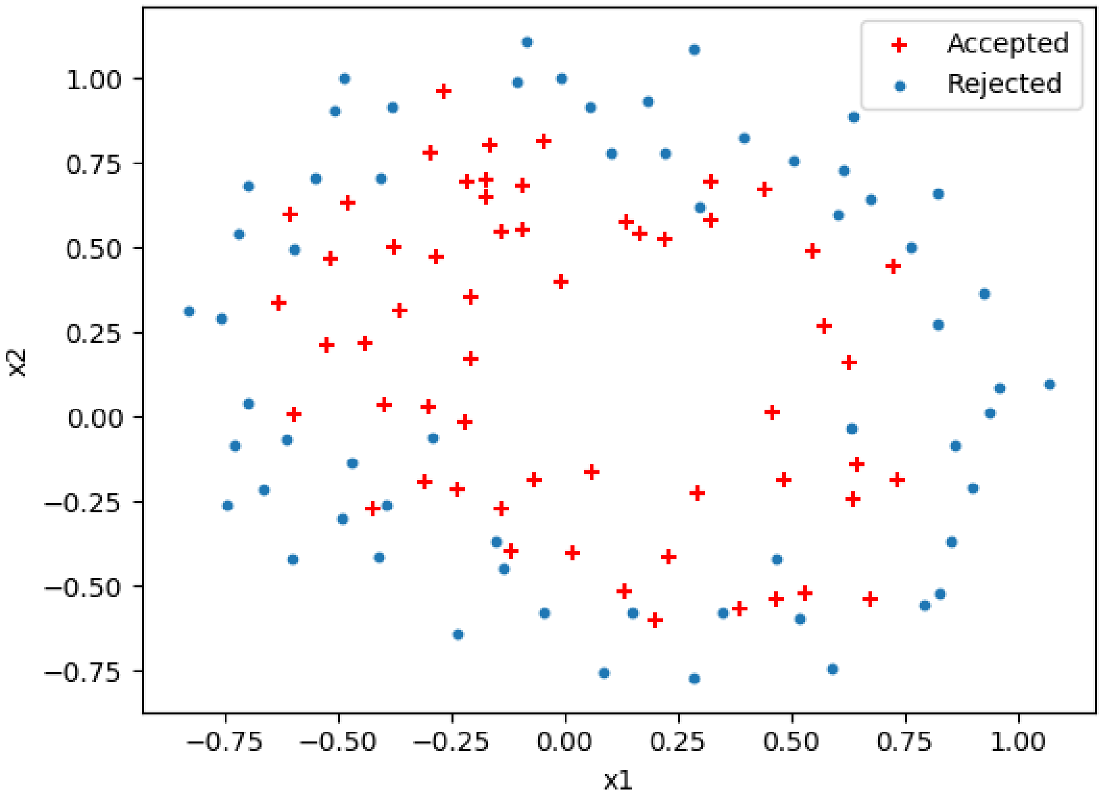


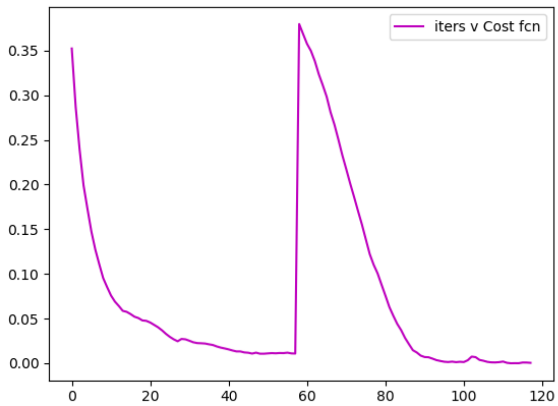
Figura 2: Pontos do classification2.

Agora, para visualizar a minimização da função de custo ao longo das iterações, ou seja, a convergência do erro para um valor próximo a zero, foram simulados três cenários distintos: o primeiro, com uma taxa de aprendizagem pequena de 0.1, o segundo com uma taxa de aprendizagem média de 0.5 e, por fim, o último com uma taxa de aprendizagem alta de 0.9.

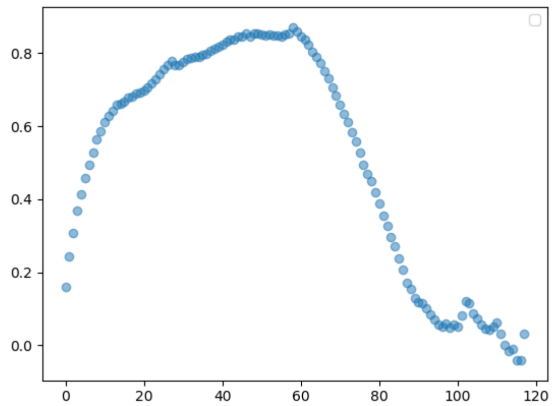
De maneira geral, foram observados nos resultados do treinamento a convergência do erro ao longo das iterações. Entretanto, ressalta-se que aproximadamente na metade do número de iterações, a saída desejada na rede muda de um para zero, gerando um pico na curva de erros que, após alguns passos de treinamento, volta a convergir para erro nulo.

Feita a análise geral das curvas de erro, tem-se que no primeiro ensaio de simulação, onde utilizou-se uma taxa de aprendizagem de 0.1, ocorre um maior número de iterações para convergência da rede, caracterizando que, ao final de treinamento, os pesos da rede são reajustados de maneira que obtenhamos um número muito próximo de zero.

A figura 3 mostra a convergência da curva do erro, bem como a descontinuidade (denotada pelo pico) repentina gerada quando se tem a mudança do valor da saída desejada. Além disso, são mostradas as saídas obtidas em cada iteração, onde se vê claramente que o último valor de saída obtido antes da mudança da saída desejada de um para zero, estava bem próxima de um e, depois, ao final do treinamento, está bem próxima de zero.



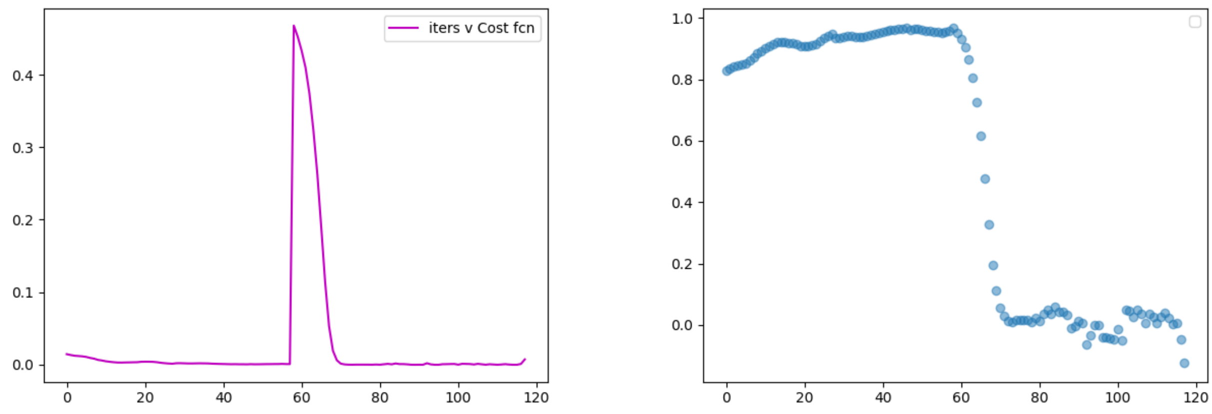
(a) Curva do erro.



(b) Saídas obtidas.

Figura 3: Simulação com taxa de aprendizagem 0.1.

No segundo cenário, com taxa de aprendizagem de 0,5 (um valor razoavelmente alto visto que eleva demais o peso da derivada do erro total na atualização dos pesos da rede e, com isso, abre a possibilidade para divergência) observamos um valor bem menor do número de iterações para convergência ao valor de saída desejado. Obviamente que, na mudança de saída desejado, o valor do erro acaba sendo maior, pois o erro anterior era bem pequeno (próximo a zero). No gráfico das saídas obtidas, se vê uma separação melhor dos valores obtidos pela rede neural, denotando uma proximidade bastante grande dos valores desejados (um e zero).

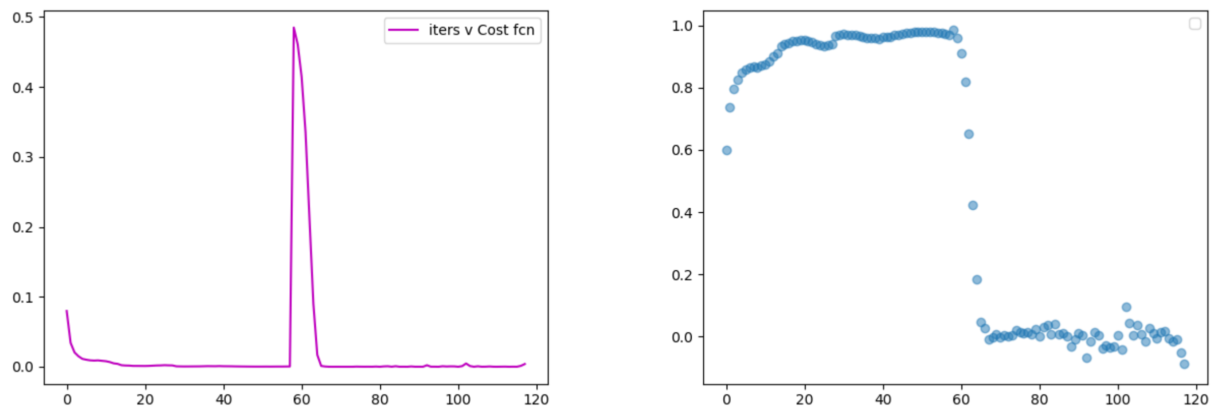


(a) Curva do erro.

(b) Saídas obtidas.

Figura 4: Simulação com taxa de aprendizagem 0.5.

Por fim, com taxa de aprendizagem de 0.9 observamos uma grande velocidade na convergência da rede, obtendo os valores desejados na saída.



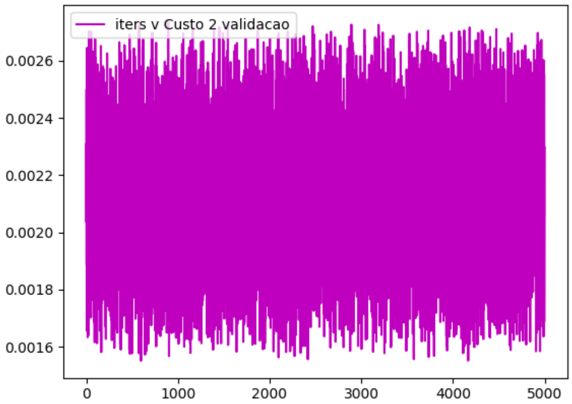
(a) Curva do erro.

(b) Saídas obtidas.

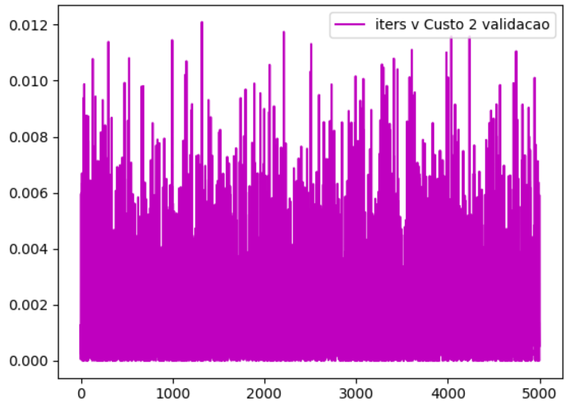
Figura 5: Simulação com taxa de aprendizagem 0.9.

Feito o treinamento da rede, é necessário validá-la com um conjunto de dados desconhecidos e, então, verificar se, para este conjunto de dados, a rede neural fornece uma saída bem aproximada. Então, para a validação do treinamento realizado em nossa rede, gerou-se 5000 valores aleatórios no intervalo (-1,1) para as entradas X_1 e X_2 visando obter zero na saída. O fato de estabelecermos zero como condição de validação da saída da rede se dá pela estruturação de nosso código. Como o último conjunto de treinamentos possuía como resposta requerida o valor nulo, os pesos (w_1,...,w_6) foram ajustados para que obtivéssemos esse valor.

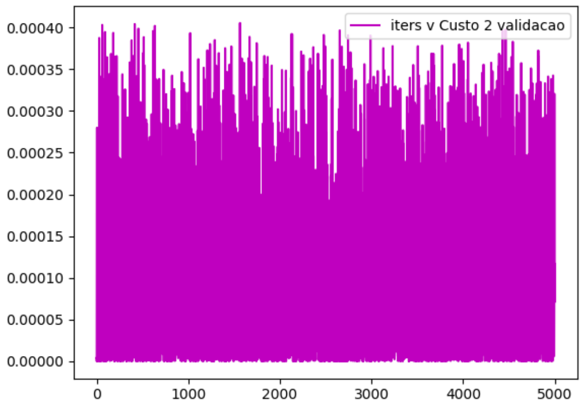
Então, com o valor dos pesos calculados, fazemos a propagação das entradas aleatórias geradas na rede atualizada. Os resultados podem ser vistos na figura 6, onde são plotadas os erros obtidos ao decorrer das iterações. Portanto, verifica-se que o treinamento da rede foi bem-sucedido, pois os valores de erros obtidos, em todos os cenários de simulação, ficaram bastante próximos de zero, denotando a proximidade da saída ao valor desejado (zero).



(a) Com taxa de aprendizagem 0.1.



(b) Com taxa de aprendizagem 0.5.



(c) Com taxa de aprendizagem 0.9.

Figura 6: Validação da rede com novo conjunto de treinamento.

Assim, considerando que foi realizado o treinamento da rede com apenas uma camada oculta, acredita-se que, com o acréscimo do número de camadas ocultas aliado ao aumento do conjunto de entradas de treinamento, obteríamos resultados ainda mais satisfatórios, o que ilustra o poder das redes neurais multicamadas.