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:

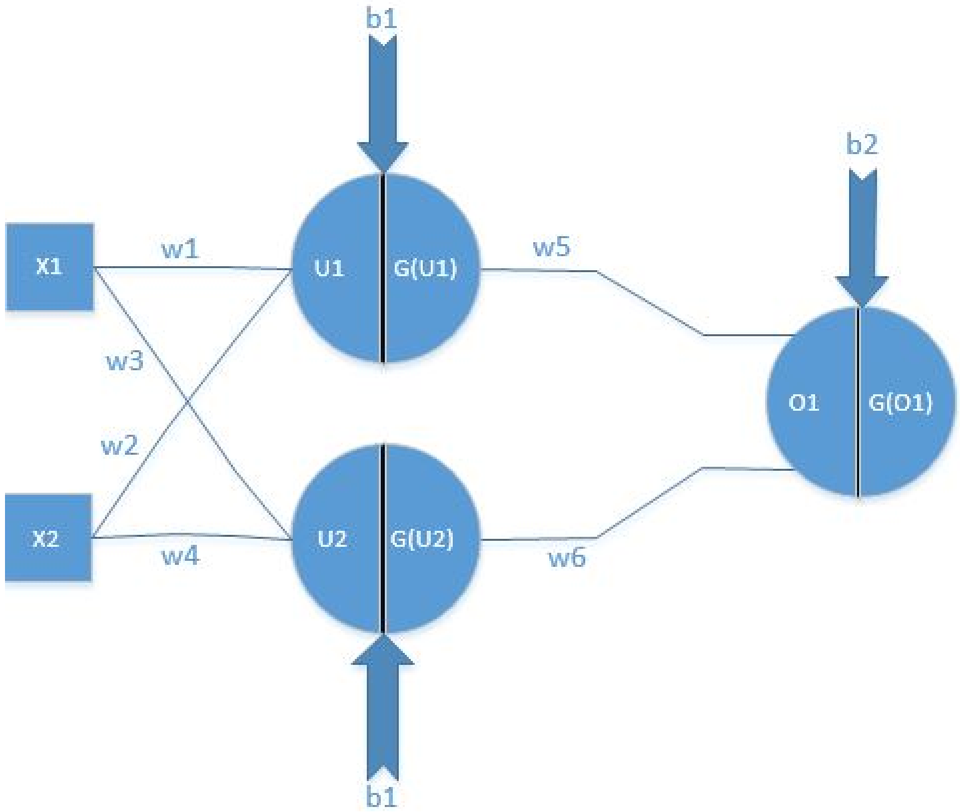

Figura 1: Estrutura da Rede Neural Multicamadas.

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:

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

```
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,
0,0,0,0,0,0,0,0]

custo = []

######################################################
#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]))
```

```python

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)

#funcao generica de ativacao do neuronio da camada de saida
def activation(neuron):
    return (1/(1+math.exp(-neuron[j])))

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

#derivada do go em relacao a o
def derivada_o(go):
    return go*(1-go)

#funcao ativacao camada oculta
def act_tanh(neuron):
    for j in range(len(neuron)):
            gu.append((np.exp(neuron[j]) - np.exp(-neuron[j])) / (np.exp(
                neuron[j]) + np.exp(-neuron[j])))
    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])):
        gh.append(w[i+4] - a*dtot[i])
    return gh

#derivada do gu em relacao a u
def derivada_gu(guu, i):
    return 1 - np.power(np.tanh(guu[i]), 2)
```

```
#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
c = 0
epochs = 10000
saida = []
erroEpoca=[]
erroHist = []
w1 = []
w2 = []
w3 = []
w4 = []
w5=  []
w6 = []
saida_ver = []
taxa = 0
lista_taxa=[]


while c <= epochs-1:

    TaxaEpoca=0
    taxa=0
    saida_ver=[]

    #loop: corrigindo os pesos para os 118 exemplos do classification2
    for cont in range(np.size(x1)):

        x = [x1[cont],x2[cont]]
        y = [y_out[cont]]
        u  = []
        gu = []

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

```

```
        g = gu

        #mostra ativacao dos neuronios da camada oculta
        act_tanh(u)
        #print("Ativacao neuronios da camada oculta: ", act_tanh(u))

        o = []

        #propagacao camada oculta para camada de sa da
        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 sa da: ", o)

        go = activation(o) #ativacao sigmoide
        #print("Ativacao neuronio da camada de sa da: ", go)
        saida.append(go)

        gaux = []
        gaux.append(go)
        g.extend(gaux)
        #print("historico ativacoes", g)

        #calculando erro total
        erro = 0.5*(y_out[cont] - go)**2
        #print("ERRO *********", erro)
        erroEpoca.append(erro)

        #Corrigindo os pesos da camada oculta para camada de sa da

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

        do = derivada_o(go)
        #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 = 0.9 #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]
        w5.append(w_updated[4])
        w_updated[5] = grad[1]
        w6.append(w_updated[5])
        #print("Pesos atualizados: ", w_updated)

        #corrigindo os pesos camada1->camada oculta

        #derivada do erro em relacao a O1
        de_o1 = dgo*do
        do_gu = weights[4]

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

        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]
            w1.append(w_updated[0])
            w_updated[1] = grad_2[1]
            w2.append(w_updated[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]
            w3.append(w_updated[2])
            w_updated[3] = grad_3[1]
            w4.append(w_updated[3])

            #print("Todos os pesos corrigidos: ", w_updated)
            if go>0.5:
                go = 1
            else:
                go=0
            saida_ver.append(go)

        weights = [] #limpa para receber os atuais
        w_updated = []
        w_1 = np.mean(w1)
        w_2 = np.mean(w2)
        w_3 = np.mean(w3)
        w_4 = np.mean(w4)
        w_5 = np.mean(w5)
```

```python
    w_6 = np.mean(w6)
    w_updated = [w_1, w_2, w_3, w_4, w_5, w_6]
    weights = w_updated  #atualiza peso
    #print("PESOS atualizados", weights)
    erroHist.append(np.mean(erroEpoca))
    w1 = []
    w2 = []
    w3 = []
    w4 = []
    w5 = []
    w6 = []
    erroEpoca = []
    c+=1
    for i in range(118):
        if y_out[i]==saida_ver[i]:
            taxa+=1
    TaxaEpoca = taxa/118
    lista_taxa.append(TaxaEpoca)
    print('Taxa na  poca : ',TaxaEpoca)

#funcao custo ao longo das epocas
plt.figure()
x = [f for f in range(epochs)]
y = [e for e in erroHist]
plt.plot(x, y, 'm', label = "epochs vs Cost function" )
plt.legend()
plt.show()

#taxa de acerto ao longo das epocas
plt.figure()
x = [f for f in range(epochs)]
y = [e for e in lista_taxa]
plt.plot(x, y, 'm', label = "epochs vs Taxa" )
plt.legend()
plt.show()


saida = []

#fazendo uma validacao da rede
for i in range (np.size(x1)):
    x = [x1[i],x2[i]]
    u  = []
    gu = []
    #propagacao entrada para camada
```

```
        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
        act_tanh(u)
        #print("Ativacao neuronios da camada oculta: ", act_tanh(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 sa da: ", o)

        go = activation(o) #ativacao sigmoide
        if go>0.5:
            go=1
        else:
            go=0
        print('Sa da: ', go)
        saida.append(go)

#plotando as 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 0 na saída. A figura 2 ilustra a disposição dos dados:

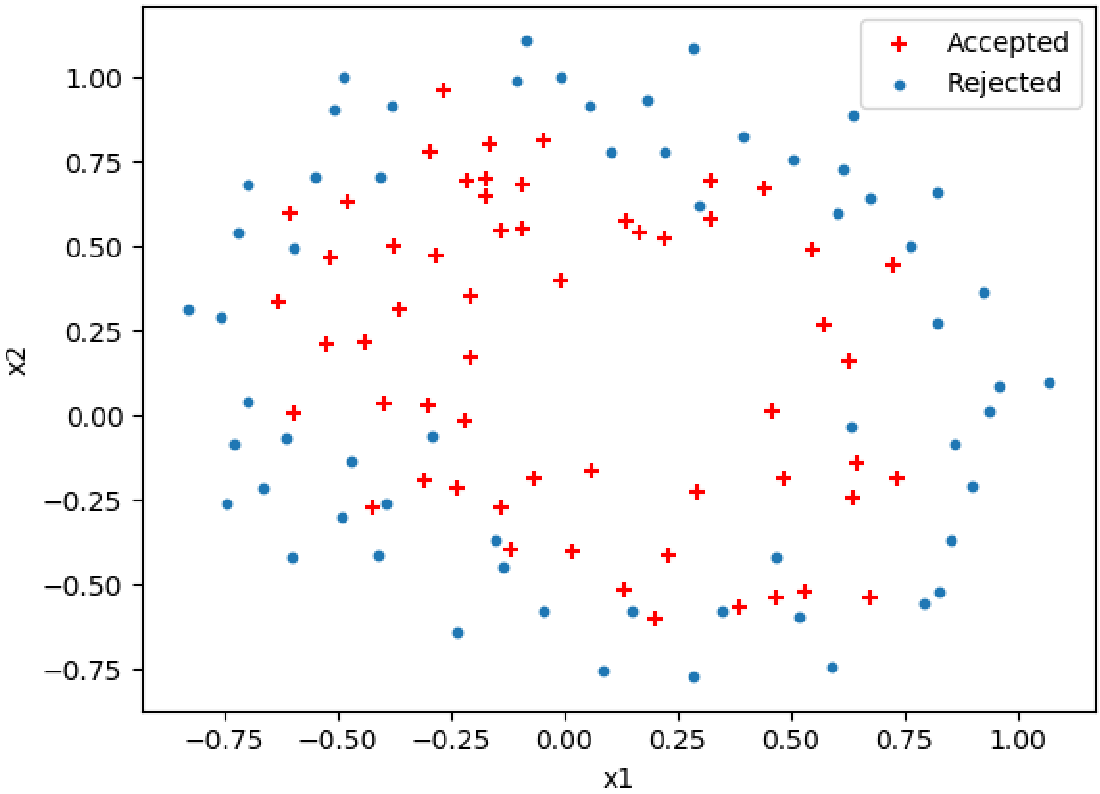

Figura 2: Pontos do classification2.

Antes de abordar os resultados obtidos, é necessário pontuar que, com relação ao primeiro relatório enviado, foram feitas algumas correções a pedido do professor, sendo elas:

- A mudança da função de ativação da camada oculta para ser a função tangente hiperbólica (tanh);
- A mudança da derivada da função de ativação da camada oculta, sendo agora a derivada da função tangente hiperbólica (tanh);
- A mudança da função de ativação do neurônio de saída para a função sigmoide;
- A mudança da derivada da função de ativação da saída, sendo agora a derivada da sigmoide;
- A realização do treinamento de todo o conjunto disponibilizado pelo arquivo *classification2.txt* para cada época, fazendo a atualização dos pesos por época.
- A contabilização da taxa de acertos da rede neural, fazendo posteriormente um plote dos dados obtidos.

Para visualizar a minimização da função de custo ao longo das épocas, 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 0.2, 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. Então, foram observados, nos três cenários de simulação, a convergência do erro para um valor próximo a zero. Porém, a rapidez com que o erro converge muda conforme o valor da taxa de aprendizagem, como será mostrado a seguir.

No primeiro ensaio de simulação, onde utilizou-se uma taxa de aprendizagem de 0.2, nota-se, pelo resultado mostrado na figura 3, uma taxa de convergência do erro rápida obtida pela rede, estagnando em um valor aproximadamente igual a 0.124.

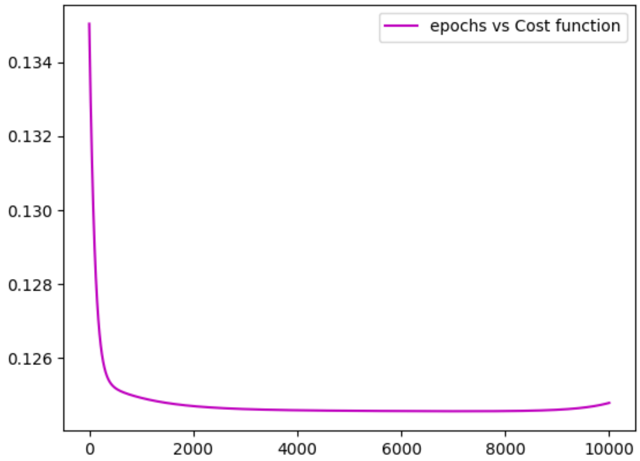

Figura 3: Simulação com taxa de aprendizagem 0.2.

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, na figura 4, que a rede levou mais tempo para convergir a um patamar de erro similar ao obtido no primeiro ensaio.

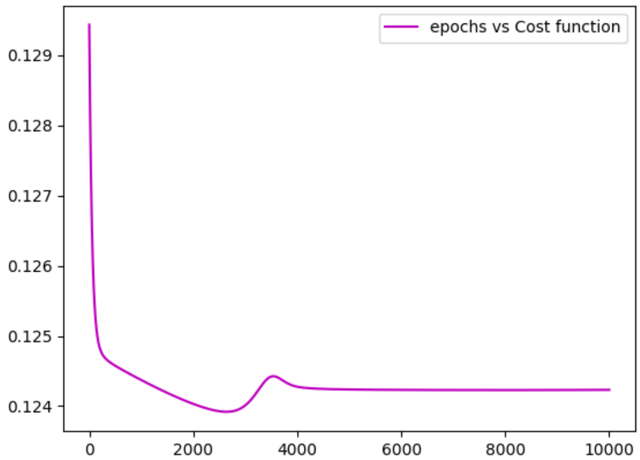

Figura 4: Simulação com taxa de aprendizagem 0.5.

Por fim, com taxa de aprendizagem de 0.9, observamos na figura 5 uma taxa de convergência mais lenta que as observadas anteriormente e, além disso, observamos que a rede converge para um patamar de erro maior que os anteriores.

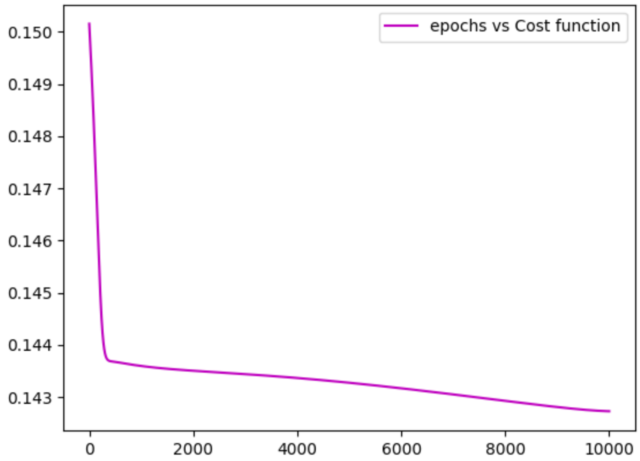

Figura 5: Simulação com taxa de aprendizagem 0.9.

Agora, a medida em que avançava o treinamento da rede, foi necessário observar a evolução da taxa de acertos da rede neural ao longo das épocas. Para isso, da mesma forma que foi feito para avaliar o erro, considerou-se taxas de aprendizagem de 0.2, 0.5 e 0.9, respectivamente. Os resultados podem ser vistos nas figura 6, 7 e 8, onde claramente se percebe que, a medida em que se aumenta a taxa de aprendizagem, o valor médio da taxa de acertos da rede aumenta. Entretanto, vale ressaltar que uma taxa de aprendizagem muito alta pode causar a divergência e não a convergência da rede para um valor de mínimo.

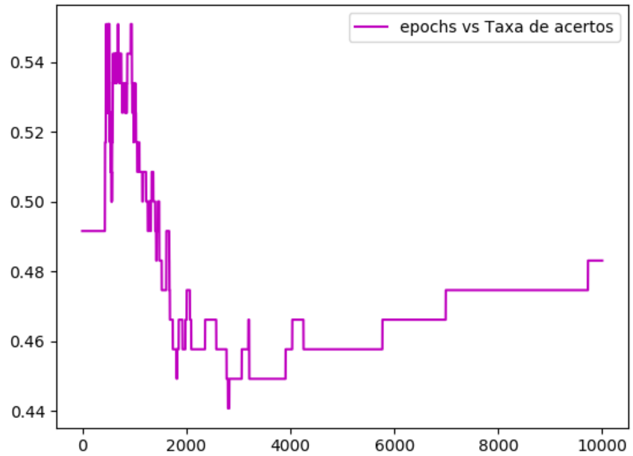

Figura 6: Taxa de acerto com taxa de aprendizagem 0.2.

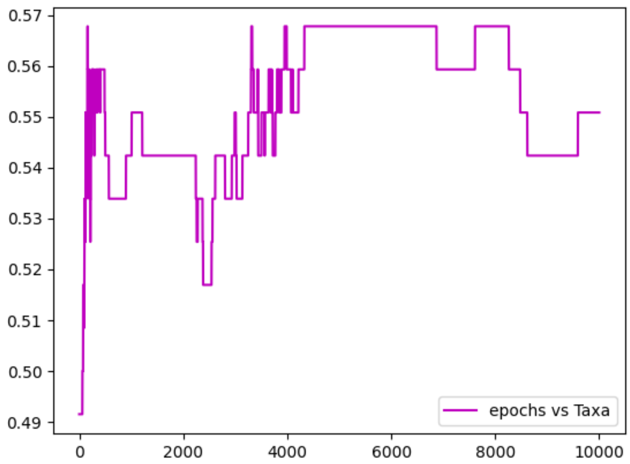

Figura 7: Taxa de acerto com taxa de aprendizagem 0.5.

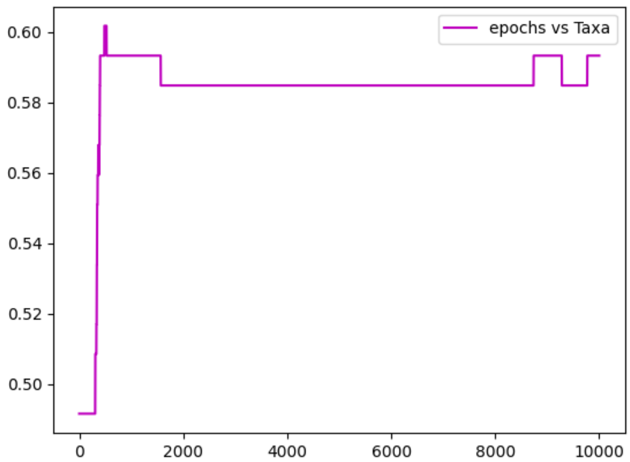

Figura 8: Taxa de acerto com taxa de aprendizagem 0.9.

4 Conclusão

Conforme visto em sala de aula, a criação de uma rede mais profunda (com maior número de neurônios e camadas) resultaria em uma taxa de acerto muito mais elevada e, consequentemente, o erro da função custo seria muito menor. Entretanto, conclui-se que, com este trabalho, foram compreendidos os conceitos teóricos mais importantes que regem as redes neurais segundo o algoritmo de retro-propagação.

5 Bibliografia

Sites consultados para a realização deste trabalho:

Fonte 1: <https://dev.to/shamdasani/build-a-flexible-neural-network-with-backpropagation-in-python>

Fonte 2: <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks>

Fonte 3: <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>