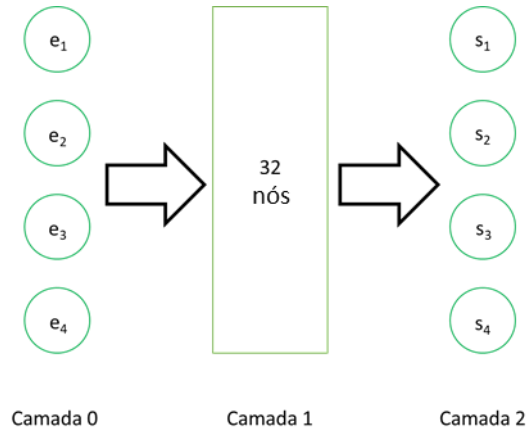


O experimento abaixo visa a comparar o desempenho do Matlab com o Python no treinamento de uma rede neuronal artificial totalmente construída sem o uso de “toolboxes” ou “libraries” para essa finalidade. A rede é composta de 4 entradas (camada 0), uma camada oculta com 32 nós, e 4 saídas (camada 2).



Como se trata de uma rede meramente demonstrativa, para “treiná-la” foram usados 16 registros binários que, juntos, “exaurem” as 16 combinações possíveis das quatro entradas. Mas, dependendo do caso, bons resultados também podem ser obtidos se apenas uma parte das possíveis combinações das entradas forem usadas para treinamento.

Os dois códigos (em Matlab e Python) são praticamente uma cópia um do outro, com as devidas adaptações de sintaxe.

A Figura 1 mostra o resultado da execução do treinamento e teste da rede em Matlab R2019b, com duração de 0,07 segundos.

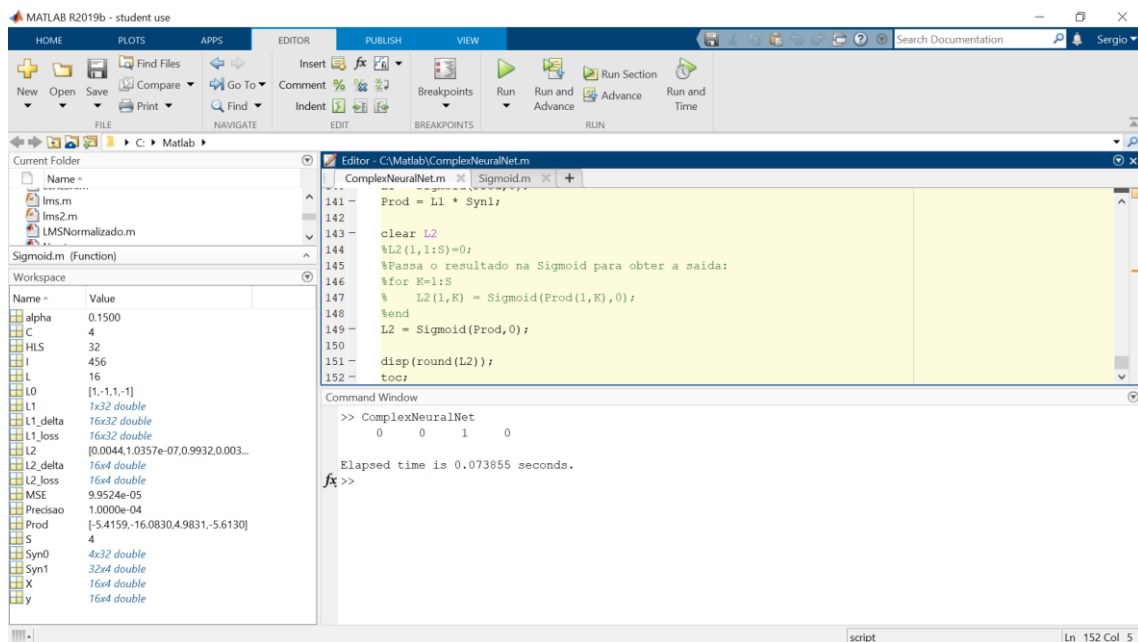
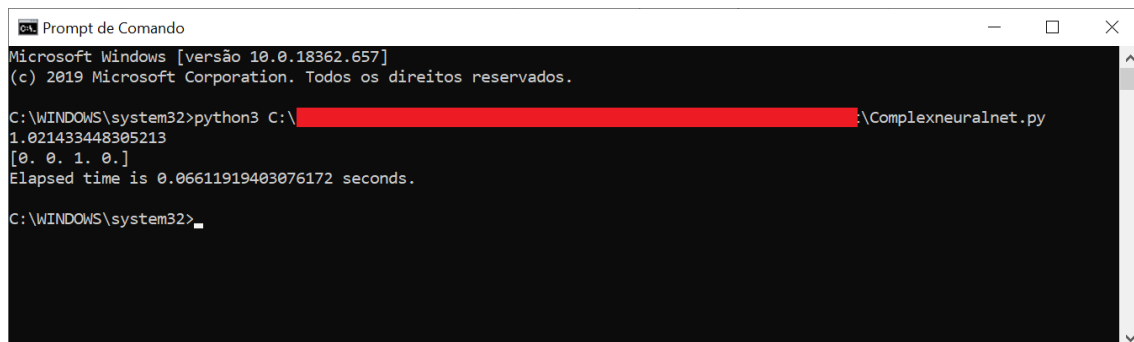


Figura 1: resultado de uma execução em Matlab R2019b

As Figuras 2 e 3 mostram os resultados da execução do treinamento e teste da rede em Python 3.7, respectivamente em linha de comando e no Visual Studio, com durações respectivas de 0,07 segundos e 0,06 segundos. Os experimentos foram repetidos várias vezes no Matlab e Python, e os resultados não variaram significativamente em relação a estes.

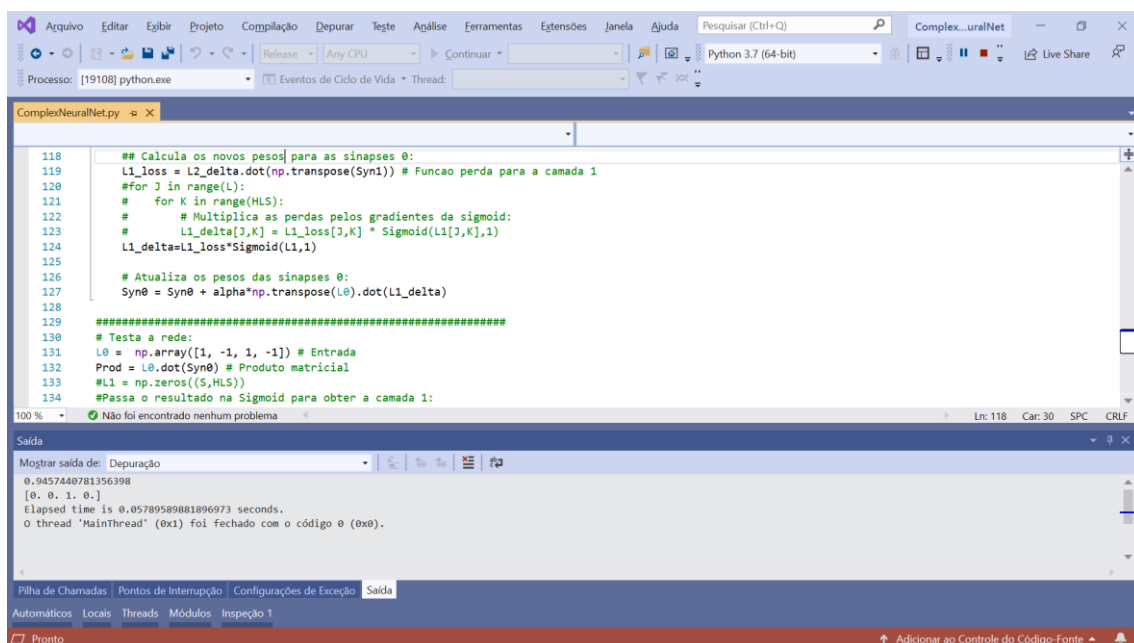


```
Prompt de Comando
Microsoft Windows [versão 10.0.18362.657]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\WINDOWS\system32>python3 C:\[redacted]:\Complexneuralnet.py
1.021433448305213
[0. 0. 1. 0.]
Elapsed time is 0.06611919403076172 seconds.

C:\WINDOWS\system32>
```

Figura 2: resultado de uma execução em Python 3.7 em linha de comando



```
ComplexNeuralNet.py
118     ## Calcula os novos pesos para as sinapses 0:
119     L1_loss = L2_delta.dot(np.transpose(Syn1)) # Funcao perda para a camada 1
120     #for J in range(L):
121     #    for K in range(HLS):
122     #        # Multiplica as perdas pelos gradientes da sigmoid:
123     #        L1_delta[J,K] = L1_loss[J,K] * Sigmoid(L1[J,K],1)
124     L1_delta=L1_loss*Sigmoid(L1,1)
125
126     # Atualiza os pesos das sinapses 0:
127     Syn0 = Syn0 + alpha*np.transpose(L0).dot(L1_delta)
128
129     #####
130     # Testa a rede:
131     L0 = np.array([1, -1, 1, -1]) # Entrada
132     Prod = L0.dot(Syn0) # Produto matricial
133     #L1 = np.zeros((S,HLS))
134     #Passa o resultado na Sigmoid para obter a camada 1:

Saída
0.9457440781356398
[0. 0. 1. 0.]
Elapsed time is 0.05789589881896973 seconds.
O thread 'MainThread' (0x1) foi fechado com o código 0 (0x0).
```

Figura 3: resultado de uma execução em Python 3.7 no Visual Studio

As Figuras de 4 a 8 serão usadas para explicar o funcionamento do código em Python, e as mesmas explicações se aplicam ao código em Matlab.

```

1 import numpy as np
2 import time
3
4 def Sigmoid(x, flag):
5     s = 1/(1+np.exp(-x))
6     if (flag==1):
7         return s*(1-s) # Retorna o valor da derivada no ponto x
8     else:
9         return s # Retorna o valor da funcao no ponto x
10
11
12 ## Implementa uma rede neuronal
13 # com 4 entradas, uma camada oculta e 4 saidas
14 #
15 # Para o treinamento:
16 #
17 # L0          L1          L2
18 # +-----+ Syn0 +-----+ Syn1 +-----+
19 # | 4 entradas |----->| 32 neuronios |----->| 4 Saidas |
20 # | 16x4      | 4x32    | 16x32      | 32x4    | 16x4      |
21 # +-----+ +-----+ +-----+
22 # 16 registros de treino
23 # com 4 entradas cada
24 #
25 # Entradas de treinamento e ...
26 X = np.array([[[-1,-1,-1,-1],
27               [-1,-1, 1,-1],
28               [-1, 1, 1, 1],
29               [-1, 1,-1,-1],
30               [-1, 1, 1, 1],
31               [-1, 1,-1,-1],
32               [-1, 1, 1, 1],
33               [-1, 1, 1, 1]]])

```

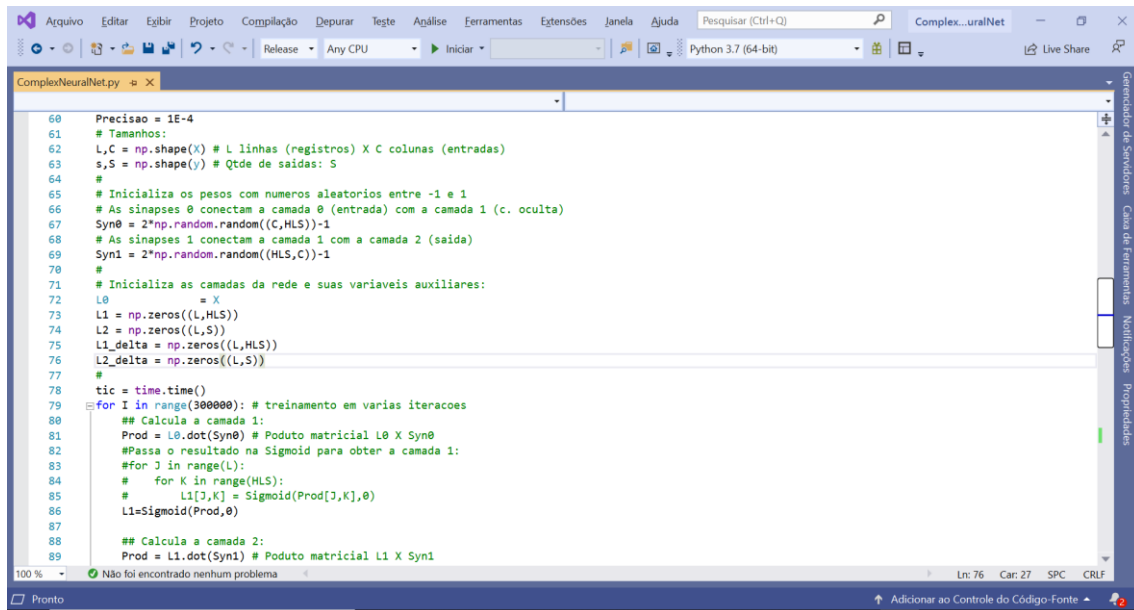
Figura 4: Código em Python (1/5)

```

30
31
32
33
34
35
36
37
38
39
40 # ... suas respectivas saidas:
41 y = np.array([[1,0,0,0],
42               [1,0,0,0],
43               [1,0,0,0],
44               [0,1,0,0],
45               [0,1,0,0],
46               [0,1,0,0],
47               [0,1,0,0],
48               [0,1,0,0],
49               [0,0,1,0],
50               [0,0,1,0],
51               [0,0,1,0],
52               [0,0,1,0],
53               [0,0,0,1],
54               [0,0,0,1],
55               [0,0,0,1],
56               [0,0,0,1]])
57
58 alpha = 0.15 # Tamanho do passo
59 HLS = 32 # Tamanho da camada oculta
60

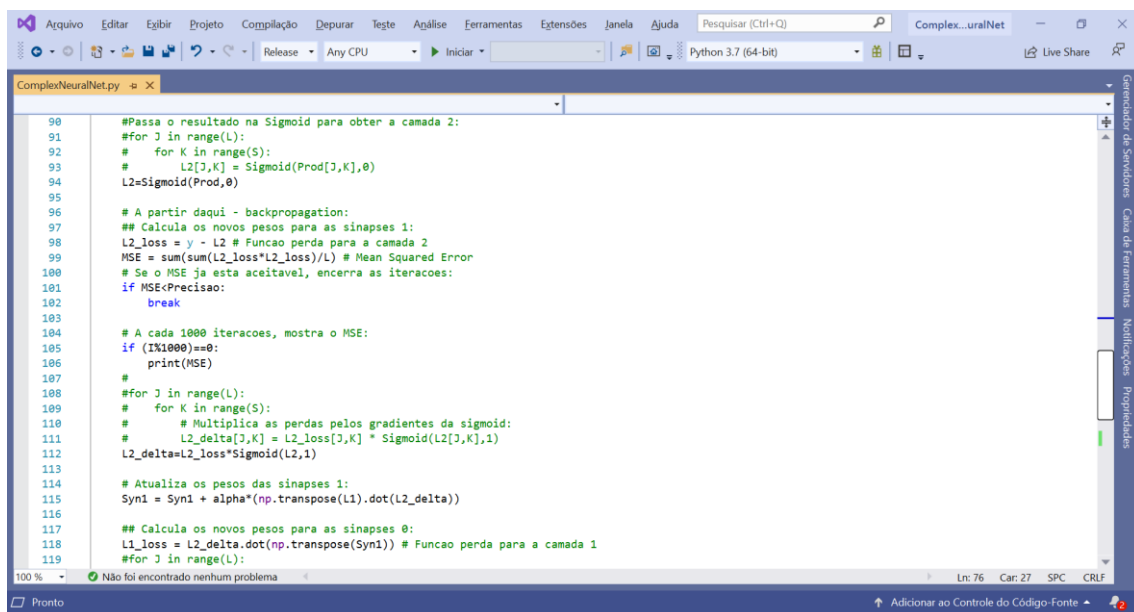
```

Figura 5: Código em Python (2/5)



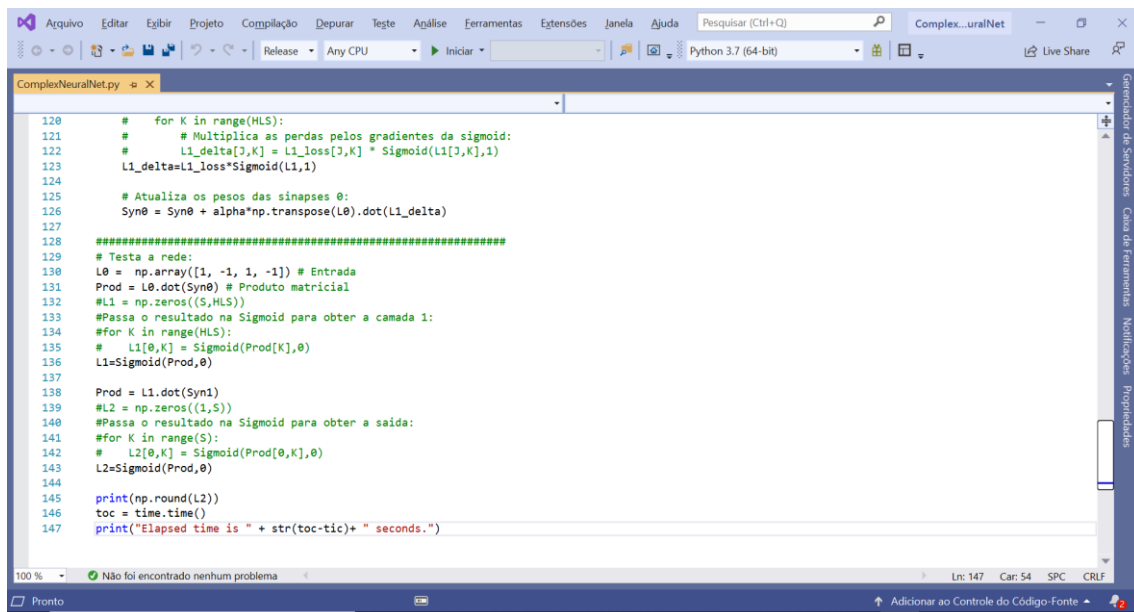
```
60 Precisao = 1E-4
61 # Tamanhos:
62 L,C = np.shape(X) # L linhas (registros) X C colunas (entradas)
63 s,S = np.shape(y) # Qtde de saidas: S
64 #
65 # Inicializa os pesos com numeros aleatorios entre -1 e 1
66 # As sinapses 0 conectam a camada 0 (entrada) com a camada 1 (c. oculta)
67 Syn0 = 2*np.random.random((C,HLS))-1
68 # As sinapses 1 conectam a camada 1 com a camada 2 (saida)
69 Syn1 = 2*np.random.random((HLS,C))-1
70 #
71 # Inicializa as camadas da rede e suas variaveis auxiliares:
72 L0 = X
73 L1 = np.zeros((L,HLS))
74 L2 = np.zeros((L,S))
75 L1_delta = np.zeros((L,HLS))
76 L2_delta = np.zeros((L,S))
77 #
78 tic = time.time()
79 for I in range(30000): # treinamento em varias iteracoes
80     ## Calcula a camada 1:
81     Prod = L0.dot(Syn0) # Produto matricial L0 X Syn0
82     #Passa o resultado na Sigmoid para obter a camada 1:
83     for J in range(L):
84         for K in range(HLS):
85             L1[J,K] = Sigmoid(Prod[J,K],0)
86     L1=Sigmoid(Prod,0)
87     ## Calcula a camada 2:
88     Prod = L1.dot(Syn1) # Produto matricial L1 X Syn1
```

Figura 6: Código em Python (3/5)



```
90 #Passa o resultado na Sigmoid para obter a camada 2:
91 for J in range(L):
92     for K in range(S):
93         L2[J,K] = Sigmoid(Prod[J,K],0)
94     L2=Sigmoid(Prod,0)
95
96 # A partir daqui - backpropagation:
97 ## Calcula os novos pesos para as sinapses 1:
98 L2_loss = y - L2 # Funcao perda para a camada 2
99 MSE = sum(sum(L2_loss*L2_loss)/L) # Mean Squared Error
100 # Se o MSE ja esta aceitavel, encerra as iteracoes:
101 if MSE<Precisao:
102     break
103
104 # A cada 1000 iteracoes, mostra o MSE:
105 if (IX1000)==0:
106     print(MSE)
107
108 #for J in range(L):
109     for K in range(S):
110         # Multiplica as perdas pelos gradientes da sigmoid:
111         L2_delta[J,K] = L2_loss[J,K] * Sigmoid(L2[J,K],1)
112     L2_delta=L2_loss*Sigmoid(L2,1)
113
114 # Atualiza os pesos das sinapses 1:
115 Syn1 = Syn1 + alpha*(np.transpose(L1).dot(L2_delta))
116
117 ## Calcula os novos pesos para as sinapses 0:
118 L1_loss = L2_delta.dot(np.transpose(Syn1)) # Funcao perda para a camada 1
119 for J in range(L):
```

Figura 7: Código em Python (4/5)



```
120 # for K in range(HLS):
121 # # Multiplica as perdas pelos gradientes da sigmoid:
122 # L1_delta[K] = L1_loss[K] * Sigmoid(L1[K],1)
123 L1_delta=L1_loss*Sigmoid(L1,1)
124
125 # Atualiza os pesos das sinapses 0:
126 Syn0 = Syn0 + alpha*np.transpose(L0).dot(L1_delta)
127
128 #####
129 # Testa a rede:
130 L0 = np.array([1, -1, 1, -1]) # Entrada
131 Prod = L0.dot(Syn0) # Produto matricial
132 #L1 = np.zeros((S,HLS))
133 #Passa o resultado na Sigmoid para obter a camada 1:
134 #for K in range(HLS):
135 # L1[K] = Sigmoid(Prod[K],0)
136 L1=Sigmoid(Prod,0)
137
138 Prod = L1.dot(Syn1)
139 #L2 = np.zeros((1,S))
140 #Passa o resultado na Sigmoid para obter a saída:
141 #for K in range(S):
142 # L2[K] = Sigmoid(Prod[K],0)
143 L2=Sigmoid(Prod,0)
144
145 print(np.round(L2))
146 toc = time.time()
147 print("Elapsed time is " + str(toc-tic)+ " seconds.")
```

Figura 8: Código em Python (5/5)

Os parágrafos I a XXIV abaixo referem-se às Figuras 4 a 8 acima.

- I. As linhas de 4 a 8 implementam a função Sigmoid (a função de ativação), que retorna o valor da função de mesmo nome no ponto de interesse ou o valor da derivada (gradiente) da função no ponto de interesse, dependendo do valor do parâmetro de entrada chamado “flag”. O ponto de interesse é o parâmetro de entrada chamado “x”. Usou-se a função sigmoid porque é contínua e diferenciável em todo o seu domínio. Ver [este artigo](#).
- II. As linhas de 24 a 39 definem os 16 registros de treinamento, e as linhas 42 a 56 definem as suas 16 respectivas saídas. Cada registro tem 4 colunas, onde cada coluna representa uma das 4 entradas da rede. Portanto, tanto as entradas de treinamento quanto as saídas de treinamento são representadas no código em matrizes 16x4.
- III. A linha 58 define o tamanho do passo para o algoritmo Gradient Descent. O valor $\alpha=0,15$ foi achado empiricamente, por tentativa. Esse valor também pode ser chamado de “taxa de aprendizado”.
- IV. A linha 59 define o tamanho da camada oculta (camada 2). Nestes experimentos usamos $HLS=32$ (total de nós na camada 2).
- V. A linha 60 define o valor da precisão desejada para atingir a convergência no Gradient Descent. Quanto menor esse valor, maior o tempo de processamento.
- VI. As linhas 62 e 63 inicializam as variáveis L (total de registros de treinamento), C (total de entradas), e S (total de saídas). Nestes experimentos, $L=16$, $C=4$, $S=4$.
- VII. As linhas 67 e 69 inicializam os pesos das sinapses com valores aleatórios entre -1 e 1, retirados de uma distribuição Uniforme de probabilidades. Nestes experimentos, $Syn0$

é inicializada como uma matriz aleatória 4x32, e Syn1 é inicializada como uma matriz aleatória 32x4.

- VIII. As linhas 72, 73 e 74 inicializam as variáveis que representam as camadas 0, 1 e 2. A camada 0 é representada pela variável L0 que é uma matriz 16x4 (16 registros com 4 entradas cada); a camada 1 é representada pela variável L1 que é uma matriz 16x32 (16 registros com 32 nós cada); e a camada 2 é representada pela variável L2 que é uma matriz 16x4 (16 registros com 4 saídas cada). Mais adiante se verá que essas dimensões decorrem de produtos matriciais, e portanto devem obedecer à regra que determina que, numa multiplicação de duas matrizes, o número de colunas da primeira matriz deve ser igual ao número de linhas da segunda matriz, e a dimensão do resultado é dada pelo número de linhas da primeira matriz versus o número de colunas da segunda matriz.
- IX. A linha 78 serve para inicializar um temporizador que mede o tempo de processamento.
- X. Entre as linhas 79 e 126 está contido um laço que executa até 300000 iterações. Este laço pode ser interrompido pelas linhas 101 e 102 se a precisão desejada for atingida antes de 300000 iterações.
- XI. A linha 81 define uma variável auxiliar (Prod) apenas para facilitar a legibilidade do código. Curiosamente, o método usado no Python para executar multiplicação matricial se chama “dot”, que é uma abreviação de “dot product” (“produto interno”, no Português), muito embora aqui isso não seja um produto interno no conceito rigoroso da Matemática. O fato é que o resultado da linha 81 é a multiplicação da matriz L0 pela matriz Syn0. No Matlab essa operação é representada pelo mesmo operador de multiplicação “*”, o qual verifica se os operandos são compatíveis para serem multiplicados um pelo outro.
- XII. A linha 86 calcula a camada 1 passando cada valor da variável auxiliar Prod pela função Sigmoid. Note que estão comentadas as linhas 83, 84 e 85. Essas três linhas fazem exatamente o que a linha 86 faz, só que usando o conceito de “escalar programming”, ao passo que a linha 86 usa o conceito de “array programming”, o qual toma proveito de otimizações implementadas pelo interpretador (ver [este artigo](#)). Isso deixa o código MUITO mais rápido. Experimente substituir a linha 86 pelas linhas 83, 84 e 85 e veja como fica lenta a execução.
- XIII. A linha 89 redefine a variável auxiliar (Prod) apenas para facilitar a legibilidade do código (ver observação contida no parágrafo XI acima). O resultado da linha 89 é a multiplicação da matriz L1 pela matriz Syn1.
- XIV. A linha 94 calcula a camada 2 passando cada valor da variável auxiliar Prod pela função Sigmoid. Note que estão comentadas as linhas 91, 92 e 93: o motivo é o mesmo que foi explicado no parágrafo XII acima.
- XV. A linha 98 calcula os “erros”, que são as diferenças entre os valores desejados (i.e.: as saídas de treinamento) e as saídas que foram calculadas a cada iteração do “laço for”.

- XVI. A linha 99 calcula o erro médio quadrático (MSE), que é a soma dos quadrados dos erros dividida pelo total de registros. Para mais detalhes veja [este artigo](#).
- XVII. As linhas 101 e 102 testam se o MSE já está adequado ao que se deseja (i.e.: se está menor que a precisão desejada); se estiver, não há necessidade de continuar com o restante das iterações, e o “laço for” descrito no parágrafo X é encerrado. Note que o “laço for” objetiva minimizar o MSE. Usou-se o MSE porque é uma função que tem apenas um ponto de mínimo, que é global (seu gráfico é um paraboloide com concavidade voltada para cima), portanto é adequada para o algoritmo Gradient Descent puro, usado aqui. Ver [este artigo](#) e [este](#).
- XVIII. As linhas 105 e 106 escrevem na tela o MSE a cada 1000 iterações.
- XIX. A linha 112 define uma variável auxiliar (L2_delta) que contém o resultado da multiplicação escalar dos erros (calculados na linha 98) pelos gradientes (derivadas) da função Sigmoid em cada ponto de interesse. Os pontos de interesse são as saídas (camada L2) que foram calculados na linha 94 (descrito no parágrafo XIV acima). Note que as linhas 109, 110 e 111 estão comentadas pelo mesmo motivo descrito no parágrafo XII acima.
- XX. A linha 115 atualiza os pesos das sinapses 1 (variável Syn1), somando seus valores anteriores com o resultado da expressão “alpha*(np.transpose(L1).dot(L2_delta))”. Vamos começar pelo que está dentro do parêntese: trata-se de uma multiplicação matricial entre a matriz transposta da camada 1 ($L1^T$) com a matriz auxiliar obtida na linha 112. A matriz L1 tem dimensão 16x32 (16 registros de treinamento versus 32 nós na camada 2), logo sua transposta tem dimensão 32x16. Multiplicando a transposta de L1 por L2_delta (que tem o formato mostrado abaixo), cada uma das 32 linhas de $L1^T$ (na qual cada linha tem 16 colunas) irá multiplicar cada uma das 4 colunas de L2_delta (onde cada coluna tem 16 linhas), somar, e produzir um resultado. Ao todo serão produzidos 128 resultados, numa matriz 32x4. Essa matriz é multiplicada de forma escalar pela taxa de aprendizado (alpha). E o resultado é somado com o valor anterior de Syn1 (a qual também é uma matriz 32x4), para atualizar os pesos de cada sinapse. Este resultado será usado na linha 118.

Formato da matriz L2_delta (16x4):

erro[1.1]*gradiente[1.1]	erro[1.2]*gradiente[1.2]	... erro[1.4]*gradiente[1.4]	
erro[2.1]*gradiente[2.1]	erro[2.2]*gradiente[2.2]	... erro[2.4]*gradiente[2.4]	
....	
....	
erro[16.1]*gradiente[16.1]	erro[16.2]*gradiente[16.2]	... erro[16.4]*gradiente[16.4]	

- XXI. A linha 118 calcula a “perda” ou “erro” na camada 1 fazendo uma multiplicação matricial de L2_delta (16x4) com $Syn1^T$ (4x32). Lembrando que $Syn1^T$ é a transposta do resultado obtido na linha 115. O resultado dessa multiplicação é uma matriz 16x32, chamada L1_loss.

- XXII. Similarmente ao que foi feito na linha 112, a linha 123 define uma variável auxiliar ($L1_delta$) que contém o resultado da multiplicação escalar dos erros (calculados na linha 118) pelos gradientes (derivadas) da função Sigmoid em cada ponto de interesse. Os pontos de interesse agora são os da camada L1 que foram calculados na linha 86 (descrito no parágrafo XII acima). Note que as linhas 120, 121 e 122 estão comentadas pelo mesmo motivo descrito no parágrafo XII.
- XXIII. Finalmente, a linha 126 atualiza os pesos das sinapses 0 (variável $Syn0$) similarmente ao que foi descrito no parágrafo XX acima. E o “laço for” é repetido em seguida.
- XXIV. As linhas 129 em diante são apenas para testar a rede usando um valor arbitrário para cada uma das 4 entradas, para verificar se a saída produzida por essa entrada coincide com a saída esperada.

Os códigos completos em Python e Matlab estão mostrados a seguir. Um bom exercício é treinar a rede com apenas um subconjunto dessas entradas, e verificar se as entradas que não foram usadas no treinamento produzem saídas corretas quando alimentam a rede depois do treinamento.

Arquivo: ComplexNeuralNetwork.py

```
import numpy as np
import time

def Sigmoid(x, flag):
    s = 1/(1+np.exp(-x))
    if (flag==1):
        return s*(1-s) # Retorna o valor da derivada no ponto x
    else:
        return s # Retorna o valor da funcao no ponto x

## Implementa uma rede neuronal
# com 4 entradas, uma camada oculta e 4 saidas
#
# Para o treinamento:
#      L0              L1              L2
# +-----+ Syn0 +-----+ Syn1 +-----+
# | 4 entradas |----->| 32 neuronios|----->| 4 Saidas |
# | 16x4      | 4x32  | 16x32  | 32x4  | 16x4  |
# +-----+ +-----+ +-----+
# 16 registros de treino
# com 4 entradas cada
#
# Entradas de treinamento e ...
X = np.array([[ -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, 1]])
# ... suas respectivas saidas:
y = np.array([[1,0,0,0],
               [1,0,0,0],
               [1,0,0,0],
               [0,1,0,0],
               [0,1,0,0],
               [0,1,0,0],
               [0,1,0,0],
               [0,1,0,0],
               [0,0,1,0],
               [0,0,1,0],
               [0,0,1,0],
               [0,0,1,0],
               [0,0,0,1],
               [0,0,0,1],
               [0,0,0,1],
               [0,0,0,1]])

alpha = 0.15 # Tamanho do passo
HLS = 32 # Tamanho da camanda oculta
Precisao = 1E-4
# Tamanhos:
L,C = np.shape(X) # L linhas (registros) X C colunas (entradas)
```

```

s,S = np.shape(y) # Qtde de saidas: S
#
# Inicializa os pesos com numeros aleatorios entre -1 e 1
# As sinapses 0 conectam a camada 0 (entrada) com a camada 1 (c. oculta)
Syn0 = 2*np.random.random((C,HLS))-1
# As sinapses 1 conectam a camada 1 com a camada 2 (saida)
Syn1 = 2*np.random.random((HLS,C))-1
#
# Inicializa as camadas da rede e suas variaveis auxiliares:
L0 = X
L1 = np.zeros((L,HLS))
L2 = np.zeros((L,S))
L1_delta = np.zeros((L,HLS))
L2_delta = np.zeros((L,S))
#
tic = time.time()
for I in range(300000): # treinamento em varias iteracoes
    ## Calcula a camada 1:
    Prod = L0.dot(Syn0) # Poduto matricial L0 X Syn0
    #Passa o resultado na Sigmoid para obter a camada 1:
    #for J in range(L):
    #    for K in range(HLS):
    #        L1[J,K] = Sigmoid(Prod[J,K],0)
    L1=Sigmoid(Prod,0)

    ## Calcula a camada 2:
    Prod = L1.dot(Syn1) # Poduto matricial L1 X Syn1
    #Passa o resultado na Sigmoid para obter a camada 2:
    #for J in range(L):
    #    for K in range(S):
    #        L2[J,K] = Sigmoid(Prod[J,K],0)
    L2=Sigmoid(Prod,0)

    # A partir daqui - backpropagation:
    ## Calcula os novos pesos para as sinapses 1:
    L2_loss = y - L2 # Funcao perda para a camada 2
    MSE = sum(sum(L2_loss*L2_loss)/L) # Mean Squared Error
    # Se o MSE ja esta aceitavel, encerra as iteracoes:
    if MSE<Precisao:
        break

    # A cada 1000 iteracoes, mostra o MSE:
    if (I%1000)==0:
        print(MSE)
    #
    #for J in range(L):
    #    for K in range(S):
    #        # Multiplica as perdas pelos gradientes da sigmoid:
    #        L2_delta[J,K] = L2_loss[J,K] * Sigmoid(L2[J,K],1)
    L2_delta=L2_loss*Sigmoid(L2,1)

    # Atualiza os pesos das sinapses 1:
    Syn1 = Syn1 + alpha*(np.transpose(L1).dot(L2_delta))

    ## Calcula os novos pesos para as sinapses 0:
    L1_loss = L2_delta.dot(np.transpose(Syn1)) # Funcao perda para a camada 1
    #for J in range(L):
    #    for K in range(HLS):
    #        # Multiplica as perdas pelos gradientes da sigmoid:
    #        L1_delta[J,K] = L1_loss[J,K] * Sigmoid(L1[J,K],1)
    L1_delta=L1_loss*Sigmoid(L1,1)

    # Atualiza os pesos das sinapses 0:
    Syn0 = Syn0 + alpha*np.transpose(L0).dot(L1_delta)

#####

```

```

# Testa a rede:
L0 = np.array([1, -1, 1, -1]) # Entrada
Prod = L0.dot(Syn0) # Produto matricial
#L1 = np.zeros((S,HLS))
#Passa o resultado na Sigmoid para obter a camada 1:
#for K in range(HLS):
#    L1[0,K] = Sigmoid(Prod[K],0)
L1=Sigmoid(Prod,0)

Prod = L1.dot(Syn1)
#L2 = np.zeros((1,S))
#Passa o resultado na Sigmoid para obter a saida:
#for K in range(S):
#    L2[0,K] = Sigmoid(Prod[0,K],0)
L2=Sigmoid(Prod,0)

print(np.round(L2))
toc = time.time()
print("Elapsed time is " + str(toc-tic)+ " seconds.")

```

Arquivo: Sigmoid.m

```
function [s] = Sigmoid(x, flag)
    s = 1 ./ (1+exp(1) .^ (-x));
    if(flag==1)
        s = s .* (1-s); % Retorna o valor da derivada no ponto x
    % else
    % Retorna o valor da funcao no ponto x
    end
end
```

Arquivo: ComplexNeuralNetwork.m

```
% Implementa uma rede neuronal
% com 4 entradas, uma camada oculta e 4 saidas
%
clear variables
% Para o treinamento:
%      L0              L1              L2
% +-----+ Syn0 +-----+ Syn1 +-----+
% | 4 entradas |----->| 32 neuronios|----->| 4 Saidas |
% | 16x4      | 4x32  | 16x32  | 32x4  | 16x4  |
% +-----+ +-----+ +-----+
% 16 registros de treino
% com 4 entradas cada
%
% Entradas de treinamento e ...
X = [-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, 1];
% ... suas respectivas saidas:
y = [1,0,0,0;
     1,0,0,0;
     1,0,0,0;
     1,0,0,0;
     0,1,0,0;
     0,1,0,0;
     0,1,0,0;
     0,1,0,0;
     0,1,0,0;
     0,0,1,0;
     0,0,1,0;
     0,0,1,0;
     0,0,1,0;
     0,0,0,1;
     0,0,0,1;
     0,0,0,1;
     0,0,0,1];
```

```
alpha = 0.15; % Tamanho do passo
HLS = 32; % Tamanho da camanda oculta
```

```

Precisao = 1E-4;
% Tamanhos:
[L,C]=size(X); % L linhas (registros) X C colunas (entradas)
[~,S]=size(y); % Qtde de saidas: S
%
% Inicializa os pesos com numeros aleatorios entre -1 e 1
% As sinapses 0 conectam a camada 0 (entrada) com a camada 1 (c. oculta)
Syn0 = 2*rand(C,HLS)-1;
% As sinapses 1 conectam a camada 1 com a camada 2 (saida)
Syn1 = 2*rand(HLS,C)-1;
%
% Inicializa as camadas da rede e suas variaveis auxiliares:
L0 = X;
L1(1:L,1:HLS) = 0;
L2(1:L,1:S) = 0;
L1_delta(1:L,1:HLS) = 0;
L2_delta(1:L,1:S) = 0;
%
tic;
for I=1:300000 % treinamento em varias iteracoes
    %% Calcula a camada 1:
    Prod = L0 * Syn0; % Poduto matricial L0 X Syn0
    %Passa o resultado na Sigmoid para obter a camada 1:
    %for J=1:L
    %     for K=1:HLS
    %         L1(J,K) = Sigmoid(Prod(J,K),0);
    %     end
    %end
    L1 = Sigmoid(Prod,0);

    %% Calcula a camada 2:
    Prod = L1 * Syn1; % Poduto matricial L1 X Syn1
    %Passa o resultado na Sigmoid para obter a camada 2:
    %for J=1:L
    %     for K=1:S
    %         L2(J,K) = Sigmoid(Prod(J,K),0);
    %     end
    %end
    L2 = Sigmoid(Prod,0);

    % A partir daqui - backpropagation:
    %% Calcula os novos pesos para as sinapses 1:
    L2_loss = y - L2; % Funcao perda para a camada 2
    MSE = sum(sum(L2_loss.^2)/L); % Mean Squared Error
    % Se o MSE ja esta aceitavel, encerra as iteracoes:
    if MSE<Precisao
        break
    end
    % A cada 1000 iteracoes, mostra o MSE:
    if mod(I,1000)==0
        disp(MSE);
    end
    %
    %for J=1:L
    %     for K=1:S
    %         % Multiplica as perdas pelos gradientes da sigmoid:
    %         L2_delta(J,K) = L2_loss(J,K) * Sigmoid(L2(J,K),1);
    %     end
    %end
    L2_delta = L2_loss .* Sigmoid(L2,1);

    % Atualiza os pesos das sinapses 1:
    Syn1 = Syn1 + alpha*(transpose(L1) * L2_delta);

    %% Calcula os novos pesos para as sinapses 0:
    L1_loss = L2_delta * transpose(Syn1); % Funcao perda para a camada 1

```

```

%for J=1:L
%   for K=1:HLS
%       % Multiplica as perdas pelos gradientes da sigmoid:
%       L1_delta(J,K)=L1_loss(J,K)*Sigmoid(L1(J,K),1);
%   end
%end
L1_delta = L1_loss .* Sigmoid(L1,1);

% Atualiza os pesos das sinapses 0:
Syn0 = Syn0 + alpha*(transpose(L0) * L1_delta);
end
% %%%%%%%%%%%
%% Testa a rede:
L0 = [1, -1, 1, -1]; % Entrada
Prod = L0 * Syn0; % Produto matricial
clear L1;
%L1(1:S,1:HLS)=0;
%Passa o resultado na Sigmoid para obter a camada 1:
%for J=1:1
%   for K=1:HLS
%       L1(J,K) = Sigmoid(Prod(J,K),0);
%   end
%end
L1 = Sigmoid(Prod,0);
Prod = L1 * Syn1;

clear L2
%L2(1,1:S)=0;
%Passa o resultado na Sigmoid para obter a saida:
%for K=1:S
%   L2(1,K) = Sigmoid(Prod(1,K),0);
%end
L2 = Sigmoid(Prod,0);

disp(round(L2));
toc;

```