EPC4: Perceptron Multicamadas

Guilherme Rocha Gonçalves[1], Rodolfo Coelho Dalapicola[1]

Universidade de São Paulo
São Carlos, Brasil

1 Introdução

1.1 Contextualização

Quarta atividade do estudo sobre Redes Neurais Artificiais desenvolvida durante o curso da disciplina SEL5712 Redes Neurais Artificiais, oferecida no primeiro semestre do ano de 2019 para os alunos do programa de mestrado em Engenharia Elétrica e oferecido pelo Professor Dr. Ivan Nunes da Silva.

1.2 Objetivo

Esta atividade têm como objetivo a criação de uma rede neural Perceptron multicamadas para solução de um problema de engenharia. O trabalho desenvolverá toda a etapa de treinamento *backpropagation*, além de usar a rede para calcular a saída de um subconjunto de teste.

2 Materiais e Métodos

2.1 Problema

Neste problema, é preciso calcular a energia absorvida por um sistema de ressonância magnética através de três variáveis de entrada. Como a solução da energia não é trivial, é proposto o desenvolvimento de uma rede neural multicamadas conforme será descrito no corpo deste trabalho.

2.2 Bases de treino e de teste

O treinamento da rede neural será feito usando o *backpropagation*. Esse método, também conhecido como *Regra Delta Generalizada*, se baseia nos desvios entre as respostas desejadas em relação às produzidas pelos neurônios de saída (isso significa que o aprendizado é supervisionado).

O *backpropagation* é dividido em duas fases, a fase *adiante*, ou *forward* e a *reversa*, ou *backward*. A primeira calcula a saída da rede para uma amostra, e a segunda regula os pesos de acordo com o erro da saída da primeira fase.

O ajuste de pesos é feito de trás para frente (daí o nome), levando em consideração o erro estimado de cada camada/neurônio.

As bases de treino e de teste foram fornecidas pelo EPC4.

2.3 Linguagem

Os códigos deste trabalho foram feitos usando *python*. Nenhuma biblioteca de Redes Neurais foram usadas, apenas bibliotecas para manipulação de matrizes e visualização de gráficos. O código está no Apêndice e também pode ser acessado através do link https://github.com/grgoncal/SEL5712—Redes-Neurais-Artificiais.

3 Resultados e discussão

3.1 Treinamento da Rede Perceptron

Foram feitos 5 treinamentos para a rede. O vetor de pesos foi iniciado com valores aleatórios entre 0 e 1, a taxa de aprendizagem foi definida como 0.1 e a tolerância do erro foi definida como 0.000001. O resultado dos treinamentos estão documentado na Tabela 1.

Treino	Erro Quadrático Médio	Épocas
1º (T1)	0.00077004	92
2º (T2)	0.00083253	83
3º (T3)	0.00079728	97
4º (T4)	0.00081356	87
5º (T5)	0.00079291	82

Tabela 1: Treinamento

Nota-se que o número de épocas entre os treinamentos muda. O mesmo vale para o Erro Quadrático Médio. Isso ocorre porque o Perceptron multicamadas possui várias soluções que podem não coincidir com o erro mínimo global. A convergência da rede alcançará sempre um ponto de mínimo local (não necessariamente o global) pelo fato de seu treinamento ser baseado na Regra Delta (que é baseada no Gradiente). Portanto, dependendo dos valores iniciais dos pesos, a convergência irá tender para mínimos diferentes.

É completamente possível portanto que uma solução seja melhor que a outra pelo simples fato de os valores de pesos iniciais estarem mais próximos de um mínimo global na convergência. Assim obtêm-se vários valores de erro que diferem entre sí. O mesmo ocorre para as épocas: um chute inicial pode cair distante do mínimo local mais próximo, resultando em mais épocas. Da mesma maneira o oposto pode ocorrer, resultando em menos épocas.

Em relação aos treinamentos desse trabalho, pode-se considerar que o primeiro treinamento obteve a melhor solução, já que obteve o menor Erro Quadrático final.

3.2 Erro Quadrático Médio

As Figuras 1 e 2 mostram a diminuição do Erro Quadrático Médio ao passar das épocas, para os dois treinamentos com mais épocas. É nítido que o erro diminuiu com as épocas, provando que o ajuste dos pesos foi adequado e resultou na redução do erro.

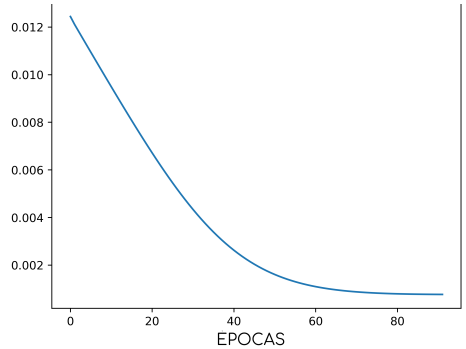

Figura 1: Erro Quadrático Médio: Teste 1.

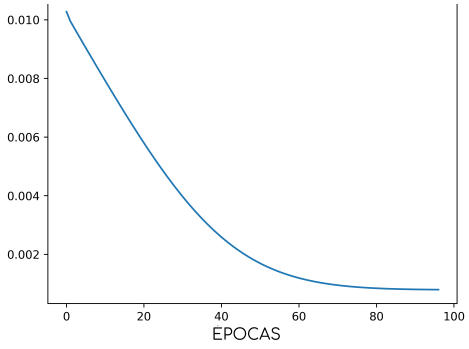

Figura 2: Erro Quadrático Médio: Teste 3.

3.3 Perceptron multicamada em casos de testes

Foram fornecidos alguns objetos de teste para que a rede possa classificar após cada iteração de treinamento. Os resultados foram próximos para todos os casos, como era esperado. A Tabela 2 mostra os resultados.

N	x1	x2	x3	d	y(1)	y(2)	y(3)	y(4)	y(5)
1	0.0611	0.286	0.7464	0.4831	0.48448	0.48723	0.48736	0.48308	0.48139
2	0.5102	0.7464	0.086	0.5965	0.59552	0.59919	0.60142	0.59667	0.59570
3	0.0004	0.6916	0.5006	0.5318	0.52784	0.53084	0.53329	0.52747	0.52458
4	0.943	0.4476	0.2648	0.6843	0.71103	0.71332	0.71135	0.52747	0.71269
5	0.1399	0.161	0.2477	0.2872	0.29033	0.28525	0.28241	0.28819	0.28892
6	0.6423	0.3229	0.8567	0.7663	0.75608	0.75804	0.75372	0.75805	0.75540
7	0.6492	0.0007	0.6422	0.5666	0.56671	0.57497	0.57068	0.57037	0.56985
8	0.1818	0.5078	0.9046	0.6601	0.67936	0.68705	0.68464	0.68274	0.68027
9	0.7382	0.2647	0.1916	0.5427	0.53172	0.53767	0.53691	0.53478	0.53590
10	0.3879	0.1307	0.8656	0.5836	0.60164	0.60964	0.60572	0.60492	0.60223
11	0.1903	0.6523	0.782	0.695	0.69131	0.69821	0.69739	0.69423	0.69266
12	0.8401	0.449	0.2719	0.679	0.67780	0.68172	0.68006	0.68049	0.67975
13	0.0029	0.3264	0.2476	0.2956	0.30019	0.29387	0.29075	0.29772	0.29611
14	0.7088	0.9342	0.2763	0.7742	0.78751	0.78591	0.78984	0.78730	0.78879
15	0.1283	0.1882	0.7253	0.4662	0.46365	0.46590	0.46563	0.46172	0.46105
16	0.8882	0.3077	0.8931	0.8093	0.82303	0.81840	0.81499	0.82143	0.82020
17	0.2225	0.9182	0.782	0.7581	0.78010	0.78373	0.78448	0.78128	0.78298
18	0.1957	0.8423	0.3085	0.5826	0.59334	0.59755	0.60063	0.59401	0.59223
19	0.9991	0.5914	0.3933	0.7938	0.80373	0.79988	0.80043	0.80294	0.80350
20	0.2299	0.1524	0.7353	0.5012	0.49416	0.49853	0.49764	0.49371	0.49306

Tabela 2: Resultado dos testes.

Os resultados foram próximos, mas não iguais. Nota-se uma certa variação dos resultados e isso acontece devido a existência de mínimos locais. Para cada treinamento, um conjunto de pesos foi calculado. Porém, cada conjunto de pesos refere-se a um mínimo do Erro Quadrático e não necessariamente os mínimos são os mesmos, como já foi discutido neste trabalho.

4 Conclusão

Os Perceptrons multicamadas são muito mais versáteis em suas aplicações e podem ser aplicados em problemas de aproximação de funcões, classificação de padrões, otimização de sistema (como no problema do EPC4), previsão de séries etc.

A sua arquitetura *feedforward* requer uma aprendizagem supervisionada e funciona de forma parecida com o ADALINE. O ajuste de pesos é feito com base no gradiente do erro em relação a saída esperada. O erro é propagado entre as camadas intermediárias, permitindo o treinamento dos neurônios das camadas intermetiárias e de entrada.

A solução de uma Rede Multicamadas pode possuir vários pontos de mínimo. Uma solução encontrará o mínimo mais próximo do conjunto de pesos inicial (que foi determinado de forma aleatória). Portanto, a rigor, uma solução pode ser melhor do que a outra, uma vez que uma solução pode ter convergido para o mínimo global, por ter seu conjunto de pesos inicial próximo deste ponto. A escolha do melhor treinamento pode ser feita com base do Erro Quadrático Médio do treinamento. E é por esse motivo que se é recomendável que se faça vários treinamentos.

O treinamento da rede é demorado, percebe-se um alto esforço computacional para o ajuste dos pesos. Porém, após o treinamento, a aplicação da rede em um passo de teste (*forward*) é bastante rápido.

O Perceptron multicamadas se mostrou muito versátil e resolveu bem o problema proposto pelo EPC4.

5 Apêndice

5.1 Código fonte

O código fonte dos algoritmos desenvolvidos está disponível no repositório público [**?**] para livre acesso.

```
# IMPORTS ------------------------------------------------------------
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt

#//////////////////////////////////////////////////////////////////
# CONSTANTS ////////////////////////////////////////////////////////
#//////////////////////////////////////////////////////////////////

learningRate = 0.1
maxEpochs = 10000
errTol = 0.000001
trainNumber = 5
numberNeurons1stLayer = 10
numberNeurons2ndLayer = 1

#//////////////////////////////////////////////////////////////////
# TRAIN ////////////////////////////////////////////////////////////
#//////////////////////////////////////////////////////////////////

def train(x, d, w1, w2, testNumber):
    # INITIALIZE EPOCHS AND ERR --------------------------------------
    epoch = 0
    Eqm = 1
    lEqm = 0
```

```
27
28      # CHART LIST ------------------------------------------------------------
29      chart = [[],[]]
30
31      while abs(Eqm - lEqm) > errTol and epoch < maxEpochs:
32          lEqm = Eqm
33          Eqm = 0
34
35          for inputIndex in range(x.shape[0]):
36
37              # INITIALIZE INPUTS
38              I1 = pd.DataFrame(np.zeros((numberNeurons1stLayer,1)))
39              I2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer,1)))
40              Y1 = I1
41              Y2 = I2
42              D1 = pd.DataFrame(np.zeros((numberNeurons1stLayer, 1)))
43              D2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer, 1)))
44
45              # FORWARD
46              for neuronIndex1stLayer in range(numberNeurons1stLayer):
47                  I1.iloc[neuronIndex1stLayer] += (w1.iloc[:,neuronIndex1stLayer]
    * x.iloc[inputIndex,:]).sum()
48                  Y1.iloc[neuronIndex1stLayer] = logistic(I1.iloc[
    neuronIndex1stLayer])
49
50              for name in reversed(list(Y1.index)):
51                  Y1.rename(index = {name : name + 1}, inplace = True)
52              Y1.loc[0] = -1
53              Y1 = Y1.sort_index()
54
55              for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
56                  I2.iloc[neuronIndex2ndLayer] += (Y1.iloc[:,neuronIndex2ndLayer]
    * w2.iloc[:,neuronIndex2ndLayer]).sum()
57                  Y2.iloc[neuronIndex2ndLayer] = logistic(I2.iloc[
    neuronIndex2ndLayer])
58
59              # BACKWARD
60              for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
61                  D2.iloc[neuronIndex2ndLayer] = (d.iloc[inputIndex] - Y2.iloc[
    neuronIndex2ndLayer]) * logistic(I2.iloc[neuronIndex2ndLayer]) * (1 -
    logistic(I2.iloc[neuronIndex2ndLayer]))
62                  for wIndex2ndLayer in range(w2.shape[0]):
63                      w2.iloc[wIndex2ndLayer, neuronIndex2ndLayer] +=
    learningRate * D2.iloc[neuronIndex2ndLayer, 0] * Y1.iloc[wIndex2ndLayer, 0]
64
65              for neuronIndex2ndLayer in range(numberNeurons2ndLayer): # Neuronio
     0 de saida
66                  for neuronIndex1stLayer in range(numberNeurons1stLayer): # 0 a
    9 neuronios de entrada
67                      D1.iloc[neuronIndex1stLayer] = D2.iloc[neuronIndex2ndLayer]
     * w2.iloc[neuronIndex1stLayer, neuronIndex2ndLayer] * logistic(I1.iloc[
    neuronIndex1stLayer]) * (1 - logistic(I1.iloc[neuronIndex1stLayer]))
68                      for inputIndex1stLayer in range(x.shape[1]):
69                          w1.iloc[inputIndex1stLayer, neuronIndex1stLayer] +=
    learningRate * D1.iloc[neuronIndex1stLayer, 0] * x.iloc[inputIndex,
    inputIndex1stLayer]
```

```
70
71          # REPEAT FORWARD STEP
72          for inputIndex in range(x.shape[0]):
73
74              I1 = pd.DataFrame(np.zeros((numberNeurons1stLayer,1)))
75              I2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer,1)))
76              Y1 = I1
77              Y2 = I2
78
79              for neuronIndex1stLayer in range(numberNeurons1stLayer):
80                  I1.iloc[neuronIndex1stLayer] += (w1.iloc[:,neuronIndex1stLayer]
     * x.iloc[inputIndex,:]).sum()
81                  Y1.iloc[neuronIndex1stLayer] = logistic(I1.iloc[
    neuronIndex1stLayer])
82
83              for name in reversed(list(Y1.index)):
84                  Y1.rename(index = {name : name + 1}, inplace = True)
85              Y1.loc[0] = -1
86              Y1 = Y1.sort_index()
87
88              for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
89                  I2.iloc[neuronIndex2ndLayer] += (Y1.iloc[:,0] * w2.iloc[:,0]).
    sum()
90                  Y2.iloc[neuronIndex2ndLayer] = logistic(I2.iloc[
    neuronIndex2ndLayer])
91                  Eqm += 0.5 * (d.iloc[inputIndex] - Y2.iloc[neuronIndex2ndLayer
    ,0]) * (d.iloc[inputIndex] - Y2.iloc[neuronIndex2ndLayer ,0])
92
93          # CALCULATE NEW ERROR
94          Eqm = (Eqm/x.shape[0])
95          print Eqm
96
97          chart[0].append(epoch)
98          chart[1].append(Eqm)
99
100         # INCREMENT EPOCH
101         epoch += 1
102
103     plt.figure(1)
104     plt.plot(chart[0], chart[1])
105     plt.savefig("./" + str(testNumber) + ".png", dpi = 500)
106     plt.close()
107
108     return w1, w2
109
110 #////////////////////////////////////////////////////////////////
111 # TEST ////////////////////////////////////////////////////////////
112 #////////////////////////////////////////////////////////////////
113
114 def test(w1, w2):
115     x = pd.read_csv('./test.csv', header = None)
116     for name in reversed(x.columns.values):
117         x.rename(columns = {name : name + 1}, inplace = True)
118     x.insert(loc = 0, column=0, value = np.full((x.shape[0],1), -1))
119
120     y = []
```

```
121
122     # REPEAT FORWARD STEP
123     for inputIndex in range(x.shape[0]):
124
125         I1 = pd.DataFrame(np.zeros((numberNeurons1stLayer,1)))
126         I2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer,1)))
127         Y1 = I1
128         Y2 = I2
129
130         for neuronIndex1stLayer in range(numberNeurons1stLayer):
131             I1.iloc[neuronIndex1stLayer] += (w1.iloc[:,neuronIndex1stLayer] * x
    .iloc[inputIndex,:]).sum()
132             Y1.iloc[neuronIndex1stLayer] = logistic(I1.iloc[neuronIndex1stLayer
    ])
133
134         for name in reversed(list(Y1.index)):
135             Y1.rename(index = {name : name + 1}, inplace = True)
136         Y1.loc[0] = -1
137         Y1 = Y1.sort_index()
138
139         for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
140             I2.iloc[neuronIndex2ndLayer] += (Y1.iloc[:,0] * w2.iloc[:,0]).sum()
141             Y2.iloc[neuronIndex2ndLayer] = logistic(I2.iloc[neuronIndex2ndLayer
    ])
142             y.append(Y2.iloc[neuronIndex2ndLayer,0])
143
144     return y
145
146 #////////////////////////////////////////////////////////////////
147 # LOGISTIC FUNCTION ///////////////////////////////////////////////
148 #////////////////////////////////////////////////////////////////
149
150 def logistic(x):
151     return 1 / (1 + math.exp(-x))
152
153 #////////////////////////////////////////////////////////////////
154 # MAIN ////////////////////////////////////////////////////////////
155 #////////////////////////////////////////////////////////////////
156
157 # TRAIN DATASET ----------------------------------------------------
158 data = pd.read_csv('./train.csv', header = None)
159
160 # GET INPUTS, OUTPUTS AND WEIGTHS ----------------------------------
161 x = data.iloc[:,1:(data.shape[1] - 1)]                          # GET
    INPUTS
162 x.insert(loc = 0, column=0, value = np.full((x.shape[0],1), -1))    # ADD -1
    INPUT
163
164 d = data.iloc[:,(data.shape[1] - 1)]
165
166 for i in range(1, trainNumber + 1):
167     w1 = pd.DataFrame(np.random.rand((data.shape[1] - 1), numberNeurons1stLayer
    ))                  # GENERATE WEIGHTS
168     w2 = pd.DataFrame(np.random.rand(numberNeurons1stLayer + 1,
    numberNeurons2ndLayer))
169
```

```
170     print("[TRAINING NUMBER " + str(i) + "]")
171     w1, w2 = train(x, d, w1, w2, i)
172     y = test(w1, w2)
173     print("[RESULT OF TRAINING NUMBER " + str(i) + "] " + str(y) + "\n")
```

Listing 1.1: Python code