EPC5: Perceptron Multicamadas Classificador

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

Universidade de São Paulo
São Carlos, Brasil

1 Introdução

1.1 Contextualização

Quinta 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 classificação em 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 exercício proposto, é preciso calcular o conservante ideal durante o processamento de uma determinada bebida. Sabe-se que existem três conservantes categorizados por tipos A, B e C. A partir de 4 variáveis de entrada definidas pelo teor de ágia (x1), grau de acidez (x2), temperatura (x3) e tensão superficial (x4), deve-se aplicar o conservante ideal.

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 EPC5 e consiste de 148 dados de treinamento e 18 dados de teste.

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. Na tabela estão resumidos os resultados da função logística de saída, mas também foi aplicada uma função de pós processamento sobre a saída, onde valores maiores que 0.5 assumiram valor 1 e valores menores que 0.5 assumiram valor 0.

Amostra	x1	x2	x3	x4	d1	d2	d3	y1p	y2p	y3p	y1	y2	y3
1	0.8622	0.7101	0.6236	0.7894	0	0	1	0	0	1	1e-7	0.0421	0.9834
2	0.2741	0.1552	0.1333	0.1516	1	0	0	1	0	0	0.9956	0.0277	3e-5
3	0.6772	0.8516	0.6543	0.7573	0	0	1	0	0	1	2e-7	0.0396	0.9803
4	0.2178	0.5039	0.6415	0.5039	0	1	0	0	1	0	0.0182	0.8219	0.0035
5	0.7260	0.7500	0.7007	0.4953	0	0	1	0	0	1	1e-6	0.1480	0.8849
6	0.2473	0.2941	0.4248	0.3087	1	0	0	1	0	0	0.9071	0.1539	1e-4
7	0.5682	0.5683	0.5054	0.4426	0	1	0	0	1	0	0.0017	0.9082	0.014
8	0.6566	0.6715	0.4952	0.3951	0	1	0	0	1	0	0.00029	0.8865	0.05347
9	0.0705	0.4717	0.2921	0.2954	1	0	0	1	0	0	0.95707	0.1371	8e-5
10	0.1187	0.2568	0.3140	0.3037	1	0	0	1	0	0	0.9823	0.0635	6e-5
11	0.5673	0.7011	0.4083	0.5552	0	1	0	0	1	0	0.0002	0.891	0.05806
12	0.3164	0.2251	0.3526	0.2560	1	0	0	1	0	0	0.9618	0.08945	9e-5
13	0.7884	0.9568	0.6825	0.6398	0	0	1	0	0	1	1e-7	0.02834	0.9872
14	0.9633	0.7850	0.6777	0.6059	0	0	1	0	0	1	1e-7	0.0323	0.9857
15	0.7739	0.8505	0.7934	0.6626	0	0	1	0	0	1	9e-8	0.0273	0.9905
16	0.4219	0.4136	0.1408	0.0940	1	0	0	1	0	0	0.97688	0.0924	6e-5
17	0.6616	0.4365	0.6597	0.8129	0	0	1	0	0	1	6e-6	0.6083	0.6103
18	0.7325	0.4761	0.3888	0.5683	0	1	0	0	1	0	0.0012	0.9091	0.0195

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. O número de épocas do melhor treinamento foi 897.

3.2 Erro Quadrático Médio

A Figura 1 mostra a diminuição do Erro Quadrático Médio ao passar das épocas, para o treinamento com melhor desempenho. É nítido que o erro diminuiu com as épocas, provando que o ajuste dos pesos foi adequado e resultou na redução do erro.

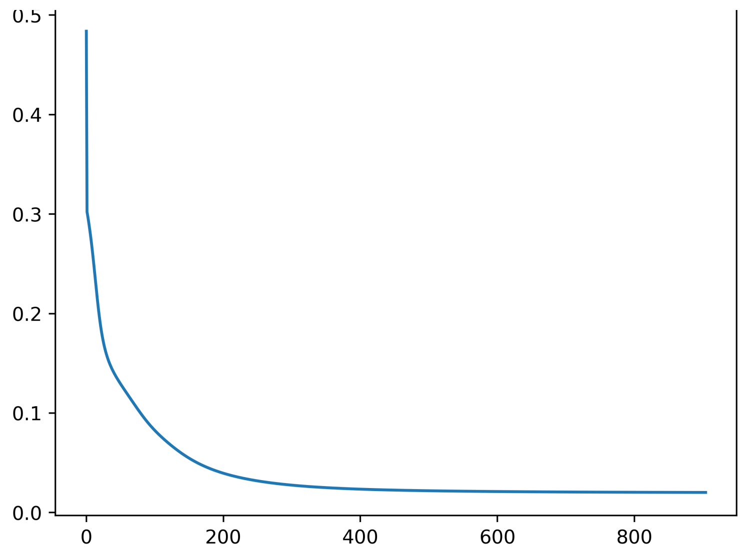

Figura 1: Erro Quadrático Médio do melhor treinamento.

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 (como neste problema), otimização de sistemas, 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 EPC5.

5 Apêndice

5.1 Código fonte

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

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

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

learningRate = 0.1
maxEpochs = 1000
errTol = 0.000001
trainNumber = 5
numberNeurons1stLayer = 15
numberNeurons2ndLayer = 3

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

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

    # CHART LIST ------------------------------------------------------
    chart = [[],[]]

    while abs(Eqm - lEqm) > errTol and epoch < maxEpochs:
        lEqm = Eqm
        Eqm = 0

        out = []

        for inputIndex in range(x.shape[0]):

            # INITIALIZE INPUTS
            I1 = pd.DataFrame(np.zeros((numberNeurons1stLayer,1)))
            I2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer,1)))
            Y1 = I1
            Y2 = I2
            D1 = pd.DataFrame(np.zeros((numberNeurons1stLayer, 1)))
            D2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer, 1)))

            # FORWARD
            for neuronIndex1stLayer in range(numberNeurons1stLayer):
                I1.iloc[neuronIndex1stLayer,0] += (w1.iloc[:,
    neuronIndex1stLayer] * x.iloc[inputIndex,:]).sum()
                Y1.iloc[neuronIndex1stLayer,0] = logistic(I1.iloc[
    neuronIndex1stLayer,0])

            for name in reversed(list(Y1.index)):
                Y1.rename(index = {name : name + 1}, inplace = True)
            Y1.loc[0] = -1
```

```
            Y1 = Y1.sort_index()

            for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
                I2.iloc[neuronIndex2ndLayer,0] += (Y1.iloc[:,0] * w2.iloc[:, neuronIndex2ndLayer]).sum()
                Y2.iloc[neuronIndex2ndLayer,0] = logistic(I2.iloc[neuronIndex2ndLayer,0])

            # BACKWARD
            for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
                D2.iloc[neuronIndex2ndLayer,0] = (d.iloc[inputIndex, neuronIndex2ndLayer] - Y2.iloc[neuronIndex2ndLayer,0]) * logistic(I2.iloc[neuronIndex2ndLayer,0]) * (1 - logistic(I2.iloc[neuronIndex2ndLayer,0]))
                for wIndex2ndLayer in range(w2.shape[0]):
                    w2.iloc[wIndex2ndLayer, neuronIndex2ndLayer] += learningRate * D2.iloc[neuronIndex2ndLayer, 0] * Y1.iloc[wIndex2ndLayer, 0]

            for neuronIndex1stLayer in range(numberNeurons1stLayer):  # Neuronio 0, 1, 2 .... 14 de entrada
                for neuronIndex2ndLayer in range(numberNeurons2ndLayer): # Neuronio 0, 1 e 2 de saida
                    D1.iloc[neuronIndex1stLayer, 0] += D2.iloc[neuronIndex2ndLayer, 0] * w2.iloc[neuronIndex1stLayer, neuronIndex2ndLayer]
                D1.iloc[neuronIndex1stLayer, 0] = D1.iloc[neuronIndex1stLayer, 0] * logistic(I1.iloc[neuronIndex1stLayer,0]) * (1 - logistic(I1.iloc[neuronIndex1stLayer,0]))
                for inputIndex1stLayer in range(x.shape[1]):
                    w1.iloc[inputIndex1stLayer, neuronIndex1stLayer] += learningRate * D1.iloc[neuronIndex1stLayer,0] * x.iloc[inputIndex, inputIndex1stLayer]

            # REPEAT FORWARD STEP

            I1 = pd.DataFrame(np.zeros((numberNeurons1stLayer,1)))
            I2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer,1)))
            Y1 = I1
            Y2 = I2

            for neuronIndex1stLayer in range(numberNeurons1stLayer):
                I1.iloc[neuronIndex1stLayer,0] += (w1.iloc[:, neuronIndex1stLayer] * x.iloc[inputIndex,:]).sum()
                Y1.iloc[neuronIndex1stLayer,0] = logistic(I1.iloc[neuronIndex1stLayer,0])

            for name in reversed(list(Y1.index)):
                Y1.rename(index = {name : name + 1}, inplace = True)
            Y1.loc[0] = -1
            Y1 = Y1.sort_index()

            for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
                I2.iloc[neuronIndex2ndLayer,0] += (Y1.iloc[:,0] * w2.iloc[:, neuronIndex2ndLayer]).sum()
                Y2.iloc[neuronIndex2ndLayer,0] = logistic(I2.iloc[neuronIndex2ndLayer,0])
                out.append(Y2.iloc[neuronIndex2ndLayer,0])
```

```
                Eqm += 0.5 * (d.iloc[inputIndex, neuronIndex2ndLayer] - Y2.iloc[neuronIndex2ndLayer ,0]) * (d.iloc[inputIndex, neuronIndex2ndLayer] - Y2.iloc[neuronIndex2ndLayer ,0])


        # CALCULATE NEW ERROR
        Eqm = (Eqm/x.shape[0])

        print abs(Eqm - lEqm)

        result = []
        for value in out:
            if value > 0.5:
                result.append(1)
            else:
                result.append(0)

        print result

        chart[0].append(epoch)
        chart[1].append(Eqm)

        # INCREMENT EPOCH
        epoch += 1

    plt.figure(1)
    plt.plot(chart[0], chart[1])
    plt.savefig("./" + str(testNumber) + ".png", dpi = 500)
    plt.close()

    return w1, w2

#//////////////////////////////////////////////////////////////////////
# TEST ////////////////////////////////////////////////////////////////
#//////////////////////////////////////////////////////////////////////

def test(w1, w2):
    x = pd.read_csv('./test.csv', header = None)
    for name in reversed(x.columns.values):
        x.rename(columns = {name : name + 1}, inplace = True)
    x.insert(loc = 0, column=0, value = np.full((x.shape[0],1), -1))

    y = []

    # REPEAT FORWARD STEP
    for inputIndex in range(x.shape[0]):

        I1 = pd.DataFrame(np.zeros((numberNeurons1stLayer,1)))
        I2 = pd.DataFrame(np.zeros((numberNeurons2ndLayer,1)))
        Y1 = I1
        Y2 = I2

        for neuronIndex1stLayer in range(numberNeurons1stLayer):
            I1.iloc[neuronIndex1stLayer,0] += (w1.iloc[:,neuronIndex1stLayer] * x.iloc[inputIndex,:]).sum()
```

```
            Y1.iloc[neuronIndex1stLayer,0] = logistic(I1.iloc[neuronIndex1stLayer,0])

        for name in reversed(list(Y1.index)):
            Y1.rename(index = {name : name + 1}, inplace = True)
        Y1.loc[0] = -1
        Y1 = Y1.sort_index()

        for neuronIndex2ndLayer in range(numberNeurons2ndLayer):
            I2.iloc[neuronIndex2ndLayer,0] += (Y1.iloc[:,0] * w2.iloc[:,neuronIndex2ndLayer]).sum()
            Y2.iloc[neuronIndex2ndLayer,0] = logistic(I2.iloc[neuronIndex2ndLayer,0])
            y.append(Y2.iloc[neuronIndex2ndLayer,0])

    return y

#//////////////////////////////////////////////////////////////////////
# LOGISTIC FUNCTION ///////////////////////////////////////////////////
#//////////////////////////////////////////////////////////////////////

def logistic(x):
    return 1 / (1 + math.exp(-x))

#//////////////////////////////////////////////////////////////////////
# MAIN ////////////////////////////////////////////////////////////////
#//////////////////////////////////////////////////////////////////////

# TRAIN DATASET -------------------------------------------------------
data = pd.read_csv('./train.csv', header = None)

# GET INPUTS, OUTPUTS AND WEIGTHS -------------------------------------
x = data.iloc[:,1:(data.shape[1] - 3)]                                  # GET INPUTS
x.insert(loc = 0, column=0, value = np.full((x.shape[0],1), -1))     # ADD -1 INPUT

d = data.iloc[:,(data.shape[1] - 3):(data.shape[1])]

for i in range(1, trainNumber + 1):
    w1 = pd.DataFrame(np.random.rand((data.shape[1] - 3), numberNeurons1stLayer))                 # GENERATE WEIGHTS
    w2 = pd.DataFrame(np.random.rand(numberNeurons1stLayer + 1, numberNeurons2ndLayer))

    print("[TRAINING NUMBER " + str(i) + "]")
    w1, w2 = train(x, d, w1, w2, i)
    y = test(w1, w2)
    print("[RESULT OF TRAINING NUMBER " + str(i) + "] " + str(y) + "\n")
    result = []
    for value in y:
        if value > 0.5:
            result.append(1)
        else:
            result.append(0)
```

```
    print str(result)
```

Listing 1.1: Python code