EPC7: Redes com função de base radial

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

Universidade de São Paulo
São Carlos, Brasil

1 Introdução

1.1 Contextualização

Sétima 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 Multicamada para solução de um problema deprevisão de funções, porém, a primeira camada da rede contém uma função de base radial. O trabalho desenvolverá toda a etapa de treinamento *backpropagation* e também do treinamento usando o *k-means*, 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 determinar se determinada substância é ou não radioativa através de duas variáveis de entrada. A topologia usada será uma rede com dois neurônios na primeira camada e um na saída. Será usada uma função de base radial na primeira camada.

2.2 Bases de treino e de teste

O treinamento da rede neural tem dois estágios: o primeiro, que envolve o treinamento dos neurônios da camada intermediária e um segundo, onde ocorre o ajuste de pesos da camada de saída.

Na primeira etapa ocorre um treinamento auto organizado (não supervisionado) usando o *k-means*. Nessa etapa são calculadas as distâncias entre as amostras e os neurônios da primeira camada. A cada iteração do *k-means*, as coordenadas dos neurônios se aproximam cada vez mais do subconjunto de dados (*cluster*) com mais amostras próximas. A variância da vizinhança do neurônio entra na equação radial, assim como sua "posição"(em duas dimensões pode-se falar em coordenadas, mas nada impede que se existam n dimensões).

A segunda etapa cuida do ajuste dos pesos da segunda camada, usando a *Regra Delta Generalizada*, comuns aos PMCs.

A função de ativação da entrada é uma gaussiana definida conforme o material da disciplina (slides RNA da aula 8) e função de saída é a função logística.

As bases de treino e de teste foram fornecidas pelo EPC7 e consiste de 40 dados de treinamento e 10 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 K-means

Foi feito o treinamento da primeira camada por meio do algoritmo de clusterização *k-means* e a Tabela 1 abaixo mostra as coordenadas e a variância dos clusters encontrados.

Cluster	Centro	Variância
1	0.52479, 0.75013	0.1012617
2	0.45313, 0.2109	0.097376

Tabela 1: Treinamento da primeira camada

Em seguida, foi realizado o treinamento da camada de saída usando o *backpropagation* com uma taxa de aprendizagem de 0.01 e uma precisão de 0.0000001. O resultado do treinamento retornou os pesos da camada de saída conforme a Tabela 2.

Peso	Valor
w1,0	-0.92978
w1,1	1.7984
w1,2	0.38978

Tabela 2: Pesos camada de saída

3.2 Casos de teste

A rede treinada calculou a saída para 10 casos de testes. A saída desses casos pode ser vista na Tabela 3.

A rede teve uma taxa de acerto de 90% e levou 463 épocas para convergir.

Para elevar a taxa de acerto desta RBF deveria-se ter mais amostras de teste para a rede. Outra estratégia é usar outras funções radiais que podem representar melhor o conjunto de dados.

Amostra	x1	x2	d	y	y(pos)
1	0.8705	0.9329	-1	-1	-0.776
2	0.0388	0.2703	1	1	0.248
3	0.8236	0.4458	-1	-1	-0.099
4	0.7075	0.1502	1	1	0.742
5	0.9587	0.8663	-1	-1	-0.679
6	0.6115	0.9365	-1	-1	-1.039
7	0.3534	0.3646	1	1	0.7378
8	0.3268	0.2766	1	1	0.9776
9	0.6129	0.4518	-1	1	0.2046
10	0.9948	0.4962	-1	-1	-0.3544

Tabela 3: Resultados da rede

4 Conclusão

As redes neurais com funções de base radial são muito versáteis e reduzem muito a quantidade de neurônios necessária para classificar ou realizar determinada tarefa. Neste EPC, um problema pode ser resolvido com apenas dois neurônios de entrada e um de saída. O fato de requerer menos neurônios reduz a complexidade e o custo de processamento.

Foi possível com uma topologia simples uma taxa de acerto de 90% com uma base de teste de apenas 40 amostras.

O método de aprendizagem não supervisionado do *k-means* para a primeira camada funcionou bem e conseguiu separar as amostras em clusters bem organizados. Em relação a camada de saída, foi usado o *backpropagation*, que novamente foi capaz de suprir as necessidades da rede.

5 Apêndice

5.1 Código fonte

O código fonte dos algoritmos desenvolvidos está disponível no repositório público https://github.com/grgoncal/SEL5712—Redes-Neurais-Artificiais para livre acesso.

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

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

error_tolerance = 0.0000001
learning_rate = 0.01
number_neurons_1st_layer = 2
number_neurons_2nd_layer = 1

#/////////////////////////////////////////////////////////////////////////
# K-MEANS /////////////////////////////////////////////////////////////////
#/////////////////////////////////////////////////////////////////////////

def kMeans(train, w1):

```

```
    last_groups = [[2],[0]]
    groups = [[],[]]

    while(last_groups != groups):

        last_groups = groups
        euclidian_distance = pd.DataFrame(np.zeros((len(train),
    number_neurons_1st_layer)))
        groups = [[],[]]

        for i in range(len(train)):
            for j in range(number_neurons_1st_layer):
                euclidian_distance.iloc[i,j] = math.sqrt(math.pow(train.iloc[i
    ,0] - w1.iloc[j,0], 2) + math.pow(train.iloc[i,1] - w1.iloc[j,1], 2))
            groups[euclidian_distance.iloc[i,:].idxmin()].append(i)

        w1 = pd.DataFrame(np.zeros((2,2)))

        for i in groups[0]:
            for j in range(number_neurons_1st_layer):
                w1.iloc[0,j] += (train.iloc[i,j])/len(groups[0])

        for i in groups[1]:
            for j in range(number_neurons_1st_layer):
                w1.iloc[1,j] += (train.iloc[i,j])/len(groups[1])

    variance = [0,0]

    for i in groups[0]:
        variance[0] +=  math.pow(train.iloc[i,0] - w1.iloc[0,0], 2) + math.pow(
    train.iloc[i,1] - w1.iloc[0,1], 2)
    variance[0] = variance[0]/len(groups[0])

    for i in groups[1]:
        variance[1] +=  math.pow(train.iloc[i,0] - w1.iloc[1,0], 2) + math.pow(
    train.iloc[i,1] - w1.iloc[1,1], 2)
    variance[1] = variance[1]/len(groups[1])

    return w1, variance


#////////////////////////////////////////////////////////////////
# OUTPUT LAYER ////////////////////////////////////////////////////
#////////////////////////////////////////////////////////////////

def outputLayer(train,w1,w2, variance):
    d = train.iloc[:,2]
    x = train.iloc[:,:-1]

    z = pd.DataFrame(np.zeros((len(train),3)))
    for i in range(len(z)):
        z.iloc[i,2] -= 1

    for i in range(len(train)):
        for j in range(number_neurons_1st_layer):
```

```
            z.iloc[i,j] = math.exp(-(math.pow(x.iloc[i,0] - w1.iloc[j,0], 2) +
    math.pow(x.iloc[i,1] - w1.iloc[j,1], 2))/(2*variance[j]))

    epoch = 0
    Eqm = 0
    lEqm = 1
    while(abs(Eqm - lEqm) > error_tolerance and epoch < 1000):
        lEqm = Eqm

        # SECOND LAYER WEIGHTS
        for i in range(len(train)):
            y = (z.iloc[i,:] * w2.iloc[:,0]).sum()

            for j in range(number_neurons_1st_layer + 1):
                gradient = (d.iloc[i] - y) * dlogistic(z.iloc[i,j])
                w2.iloc[j,0] += learning_rate * gradient * z.iloc[i,j]

        # QUADRATIC ERROR
        Eqm = 0
        for i in range(len(train)):
            y = (z.iloc[i,:] * w2.iloc[:,0]).sum()
            Eqm += math.pow(d.iloc[i] - y ,2)/2
        Eqm = Eqm/len(train)
        print epoch
        print "[QUADRATIC ERROR] " + str(abs(Eqm - lEqm))

        epoch += 1

    return w2

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

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

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

def dlogistic(x):
    return logistic(x) * (1 - logistic(x))

#////////////////////////////////////////////////////////////////////
# TEST SAMPLES /////////////////////////////////////////////////////
#////////////////////////////////////////////////////////////////////

def runTest(train,w1,w2, variance):
    x = train.iloc[:,:-1]

    z = pd.DataFrame(np.zeros((len(train),3)))
    for i in range(len(z)):
        z.iloc[i,2] -= 1

    for i in range(len(train)):
```

```
        for j in range(number_neurons_1st_layer):
            z.iloc[i,j] = math.exp(-(math.pow(x.iloc[i,0] - w1.iloc[j,0], 2) +
    math.pow(x.iloc[i,1] - w1.iloc[j,1], 2))/(2*variance[j]))

    for i in range(len(train)):
        y = (z.iloc[i,:] * w2.iloc[:,0]).sum()
        if y > 0:
            print "1 " + str(y)
        else:
            print "-1 " + str(y)

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

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

w1 = pd.DataFrame(train.iloc[:number_neurons_1st_layer,:-1])
w2 = pd.DataFrame(np.random.rand(number_neurons_1st_layer + 1,
    number_neurons_2nd_layer))

# CALCULATE CLUSTERS
w1, variance = kMeans(train, w1)
w2, outputLayer(train,w1,w2,variance)
runTest(test, w1, w2, variance)
print "[COORDINATES W1] " + str(w1)
print "[W2] " + str(w2)
print "[VARIANCE] " + str(variance)
```

Listing 1.1: Python code