UNIVERSIDADE​ ​FEDERAL​ ​DO​ ​RIO​ ​GRANDE​ ​DO​ ​SUL

DEPARTAMENTO​ ​DE​ ​ENGENHARIA​ ​ELÉTRICA

ENG04019​ ​–​ TÓPICOS ESPECIAIS EM INSTRUMENTAÇÃO

MATHEUS QUEVEDO SIVELLI

**TRABALHO​ 5 – REDES NEURAIS**

Porto alegre

2020

SUMÁRIO

[1 INTRODUÇÃO 4](#_Toc42555744)

[2. EXERCÍCIOS 5](#_Toc42555745)

[2.1 ESTUDO DO DATASET 5](#_Toc42555746)

[2.2 NORMALIZAÇÃO DOS DADOS 8](#_Toc42555747)

[2.3 HEATMAP DE CORRELAÇÃO 8](#_Toc42555748)

[2.4 OTIMIZAÇÃO DE HIPERPARÂMETROS 10](#_Toc42555749)

[2.5 ANÁLISE DE CORRELAÇÃO 11](#_Toc42555750)

[2.6 AVALIAÇÃO COM CONJUNTO TESTE 12](#_Toc42555751)

[2.7 OPERAÇÕES COM MATEMÁTICA INTERVALAR 13](#_Toc42555752)

[2.8 CALCULANDO A ÁREA DO RETÂNGULO COM MATEMÁTICA INTERVALAR 14](#_Toc42555753)

[3. CONCLUSÃO 16](#_Toc42555754)

LISTA DE FIGURAS

[Figura 1 - Descrição dos atributos de entrada do dataset 5](#_Toc42555839)

[Figura 2 - Descrição das variavéis de saída 7](#_Toc42555840)

[Figura 3 - Resultado da normalização 8](#_Toc42555841)

[Figura 4 - Mapa de correlação das variáveis utilzadas no sistema 9](#_Toc42555842)

[Figura 5 - Resultado do GridSearch 11](#_Toc42555843)

[Figura 6 - Resultado do GridSearch com atributos otimizados 12](#_Toc42555844)

[Figura 7 - Resultado com o conjunto de teste 13](#_Toc42555845)

[Figura 8 - Resultado das operações matemáticas 14](#_Toc42555846)

[Figura 9 - Resultado do problema 8 15](#_Toc42555847)

1 INTRODUÇÃO

A tecnologia avançou muito com o passar do tempo e com isso, as técnicas de inteligência artificial também. Foi em 1986 quando se anunciaram um avanço extremamente importante na área das redes neurais, que foi o forjado o termo e o método de *‘backpropagation’.* Esse método consiste em a partir de um erro conhecido na camada de saída de uma rede neural encontrar o erro localmente em cada nó e, portanto, atualizar cada peso, aprimorando a rede neural para o melhor resultado do problema.

Se atemos em estudar a implementação de métodos de redes neurais *feedfoward*, que é a construção da rede na direção da camada de saída, e do *backpropagation* aliado com o *stochastic gradient descent* para encontrarmos os pesos atualizados em cada neurônio das camadas ocultas. Aplicamos os métodos estudados nos problemas clássicos de regressão e classificação.

2. EXERCÍCIOS

2.1 INTRODUÇÃO TÉORICA E IMPLEMENTAÇÃO COM VALIDAÇÃO

Redes neurais são baseadas em neurônios. Os conceito de neurônio é relacionado a uma unidade de processamento que possui, basicamente três elementos, são eles:

* Entradas.
* Somador
* Função de ativação

O somador tem como principal função de multiplicar as entradas pelos respectivos pesos do neurônio e acrescentar o bias, posteriormente, é aplicado o resultado do somador na função de ativação que resulta na resposta do neurônio para essas condições.

Dentro da teoria de redes neurais, possuímos alguns métodos que constituem um processo de desenvolvimento de uma aplicação baseada em redes neurais. O primeiro deles é a construção da rede *feedfoward,* que é a parte onde construímos o caminho da rede em direção a saída da mesma, com os pesos, bias e funções de ativações já inicializados. Posteriormente, é aplicado o método de backpropagation para conseguirmos ir atualizando os pesos a fim de encontrar uma resposta mais assertiva para o contexto da aplicação. Além disso, utilizamos o stochastic grandient descent para irmos atualizando os pesos de acordo com cada elemento do conjunto de treinamento, não precisando processar todo o treinamento para depois aferir os pesos de cada neurônio.

2.2 EXEMPLO MANUAL E BACKPROPAGATION

Utilizamos o código fornecido pelo professor como base para implementarmos técnicas de backpropagation e stochastic gradient descent. O código é apresentado abaixo.

import numpy as np

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.metrics import accuracy\_score, mean\_squared\_error

class NeuralNetwork:

    '''simple feedforward neural network class'''

    def \_\_init\_\_(self,num\_inputs,num\_outputs,num\_neurons,tf\_functions,epochs=100,learning\_rate=0.1):

        self.learning\_rate = learning\_rate

        self.epochs = epochs

        self.num\_inputs = num\_inputs

        self.num\_outputs = num\_outputs

        self.num\_neurons = num\_neurons

        self.min\_weight\_value = -1

        self.max\_weight\_value = 1

        self.tf\_functions = []

        #self.tf\_functions\_derivatives = []

        for i in range(0,len(tf\_functions)):

            if tf\_functions[i] == "logistic":

                self.tf\_functions.append(self.tf\_logistic)

                #self.tf\_functions\_derivatives.append(self.tf\_logistic\_derivative)

            elif tf\_functions[i] == "linear":

                self.tf\_functions.append(self.tf\_linear)

                #self.tf\_functions\_derivatives.append(self.tf\_linear\_derivative)

            else:

                print("Unknown transfer function: %s" %(tf\_function[i]))

                exit()

        self.outputs = None # will be updated after calculation is called

        #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        # Initialize weights and bias

        #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        weights\_per\_layer = []

        biases\_per\_layer = []

        wsums\_per\_layer = []  # for helping later activation function approximate derivative calculation

        for l in range(0,len(num\_neurons)): # for all layers

            if (l == 0):

                previous\_layer\_size = len(inputs[0])

            else:

                previous\_layer\_size = num\_neurons[l-1]

            layer\_size = num\_neurons[l]

            layer\_weights = np.zeros((layer\_size,previous\_layer\_size))

            weights\_per\_layer.append(layer\_weights)

            biases\_per\_layer.append(np.zeros(layer\_size))

            wsums\_per\_layer.append(np.zeros(layer\_size))

self.randomize\_weights()

         self.randomize\_biases()

    def set\_last\_layer\_weights(self,weights):

        self.weights\_per\_layer[-1] = [weights]

    def randomize\_weights(self):

        a = self.min\_weight\_value

        b = self.max\_weight\_value

        for i, layer in enumerate(self.weights\_per\_layer):

            #self.weights\_per\_layer[i] = np.random.random(self.weights\_per\_layer[i].shape)

            self.weights\_per\_layer[i] = a+(b-a)\*np.random.random(self.weights\_per\_layer[i].shape)

    def randomize\_biases(self):

        a = self.min\_weight\_value

        b = self.max\_weight\_value

        for i, layer in enumerate(self.biases\_per\_layer):

            self.biases\_per\_layer[i] = a+(b-a)\*np.random.random(self.biases\_per\_layer[i].shape)

    def predict(self,X):

        Ypredicted = []

        for i in range(0,X.shape[0]):  #for every pattern

            predicted = model.calc\_output(X[i]) # calculating outputs for a given pattern

            Ypredicted.append(predicted)

        return np.array(Ypredicted)

    def calc\_output(self,inputs):

        '''calculates the output of the neural network'''

        #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        # creates empty output array

        #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        outputs = []

        for i in range(0,len(self.num\_neurons)):

            outputs.append(np.zeros(num\_neurons[i]))

        outputs = np.array(outputs) #convert to numpy array

        #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        # calculates output for each layer

        #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        for i in range(0,len(self.num\_neurons)):

            if (i == 0): # first layer

                outputs[i] = self.calc\_layer(inputs,self.weights\_per\_layer[i],self.biases\_per\_layer[i],self.tf\_functions[i],i)

            else:

                outputs[i] = self.calc\_layer(outputs[i-1],self.weights\_per\_layer[i],self.biases\_per\_layer[i],self.tf\_functions[i],i)

        self.outputs = outputs

        return outputs[-1]

def calc\_layer(self,inputs,weights,biases,tf\_function,layer\_index):

        outputs = np.zeros(len(weights))

        for i in range(0,len(weights)): #for all neurons

            outputs[i] = self.neuron(weights[i],biases[i],inputs,tf\_function,layer\_index,i)

        return outputs

    def tf\_logistic(self,x):

        return 1/(1+np.exp(-x\*2))

        #return 1/(1+np.exp(-x))

    def tf\_logistic\_derivative(self,y):

        return (1-y)\*(y)

    def tf\_linear(self,x):

        return x

    def tf\_linear\_derivative(self,y):

        return 1

    def neuron(self,weights,bias,inputs,tf\_function,layer\_index,neuron\_index):

        weighted\_sum = np.dot(weights,inputs)+bias

        output = tf\_function(weighted\_sum)

        # update for later calculate activation function approximate derivative

        self.wsums\_per\_layer[layer\_index][neuron\_index] = weighted\_sum

        return output

    def fit(self,X,Y):

        # sequential approach

        self.ssr\_total\_list = []

        self.mse\_total\_list = []

        for epc in range(0,self.epochs+1):

            ssr\_total = 0

            mse\_total = 0

            idxlist = np.arange(0,X.shape[0])

            #on first evaluation, local gradients are not updated, just to save the initial solution

            np.random.shuffle(idxlist)  #randomize index\_list

            for i in range(0,X.shape[0]):

                predicted = model.calc\_output(X[idxlist[i]]) # calculating outputs for a given pattern

                output\_error = Y[idxlist[i]] - predicted  # error for each output

                ssr = 0.5\*sum(output\_error\*\*2) #sum of squared residuals

                ssr\_total += ssr

                mse\_total += sum(output\_error\*\*2)

                # calculate local gradients

                if (epc != 0):

                    self.calculate\_local\_gradients(output\_error,ssr)

                    self.update\_weights(X[idxlist[i]])

            mse = mse\_total/X.shape[0]

            self.ssr\_total\_list.append(ssr\_total)

            self.mse\_total\_list.append(mse)

            #print("Epoch: %d \t SSR\_total = %f" %(epc,ssr\_total))

def calculate\_local\_gradients(self,output\_error,ssr):

        ''' calculates local gradients '''

        # initializes local gradients

        self.local\_gradients = [0]\*len(num\_neurons) # will be a list for each element

        for i in range(0,len(self.local\_gradients)):

            self.local\_gradients[i] = np.zeros(num\_neurons[i])

        # calculates local gradients for output layer

        for j in range(0,self.num\_neurons[-1]): # for all neurons in the output layer

            partiald\_E\_y = -(output\_error[j])

            #local\_gradient = -partiald\_E\_y \* self.tf\_functions\_derivatives[-1](self.outputs[-1][j])

            tf = self.tf\_functions[-1]

            local\_gradient = -partiald\_E\_y \* self.num\_derivative(tf,self.wsums\_per\_layer[-1][j])

            self.local\_gradients[-1][j] = local\_gradient

        # calculates local gradients for hidden layers

        for l in range(len(num\_neurons)-2,-1,-1):  # for all hidden layers, from last to first

            for j in range(0,num\_neurons[l]):

                outsum = 0

                w\_from\_this\_neuron\_to\_next\_layer = self.weights\_per\_layer[l+1][:,j]

                for o in range(0,num\_neurons[l+1]): # for all neurons on the next layer

                    wok = w\_from\_this\_neuron\_to\_next\_layer[o]

                    outsum+= self.local\_gradients[l+1][o]\*wok

                #self.local\_gradients[l][j] = self.tf\_functions\_derivatives[l](self.outputs[l][j])\*outsum

                tf = self.tf\_functions[l]

                self.local\_gradients[l][j] = self.num\_derivative(tf,self.wsums\_per\_layer[l][j])\*outsum

def update\_weights(self,inputs):

        '''update weights and bias for backpropagation'''

        # weight update

        for l in range(0,len(num\_neurons)): # for all layers

            for j in range(0,num\_neurons[l]): # for all neurons in layer

                if (l == 0): # first hidden layer

                    for p in range(0,self.num\_inputs): # for all inputs

                        self.weights\_per\_layer[l][j,p] += self.learning\_rate \* self.local\_gradients[l][j] \* inputs[p]

                else:

                    for p in range(0,num\_neurons[l-1]): # for all neurons in previous layer

                        self.weights\_per\_layer[l][j,p] += self.learning\_rate \* self.local\_gradients[l][j] \* self.outputs[l-1][p]

                # bias update

                self.biases\_per\_layer[l][j] += self.learning\_rate \* self.local\_gradients[l][j] \* 1

        print("Biases per layer: ", self.biases\_per\_layer)

        print("Pesos por layer: ", self.weights\_per\_layer)

        print("Local Gradients: ", self.local\_gradients)

    def num\_derivative(self,f,x,delta=1e-6):

        return (f(x+delta)-f(x))/delta

Usaremos o exemplo mais simples para validaremos a implementação. Os parâmetros que serão utilizados são apresentados abaixo:

* Entrada: [0,1]
* Pesos: Todos zerados
* Bias: Zerados
* Função logística na camada oculta
* Função linear na camada de saída.
* Épocas: 1

Primeiramente, calcularemos manualmente.

2.3 REGRESSOR COM REDES NEURAIS

Nesta seção, abordamos a construção de um regressor utilizando as redes neurais para o método preditivo. Estudaremos o dataset fornecido pelo SKLearn, Boston, que fornece o preço das casas em Boston. O conjunto de dados possui 506 amostras com 13 atributos em cada elemento. Inicialmente foi preciso normalizar todo o conjunto de dados apresentados, tanto as entradas quanto as saídas, pois a diferente escala influência diretamente na construção da rede neural.

Definimos as propriedades de nossa rede neural baseada na estrutura do dataset, portanto, incrementamos 13 neurônios de entrada com apenas um neurônio de saída, que é o valor alvo.

Foi separado 20% do dataset para ser nosso conjunto de teste e o restante para utilizar nos processos de treinamento e validação da rede.

Mantemos as mesmas funções de ativações que o exemplo passado e com os pesos inicializando de maneira randômica, além disso, delimitamos a otimização de hiperparametros apenas a taxa de aprendizado pois entendemos que o poder computacional é um limitador de nossos estudos e, consequentemente, o tempo que o modelo necessita para otimizar os outros hiperparametros, portanto, foi escolhido um valor default de 100 épocas para cada problema analisado nesse documento.

Por fim, otimizamos a taxa de aprendizado com todo o conjunto de treinamento e outra aplicando o cross-validation. A segunda abordagem de otimização aumentou consideravelmente a complexidade de nosso modelo e o tempo de execução foi bastante impactado, porém, como julgamos ter sido mais verossímil, foi a abordagem escolhida. O intervalo de valores de taxa de aprendizado utilizado foi delimitado para que o tempo de execução não saturasse 60 minutos. O código apresentado abaixo é a primeira abordagem a otimização de parâmetros.

from sklearn.datasets import load\_boston

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import cross\_val\_score

data = load\_boston()

target = data['target']

data = data['data']

target = target.reshape(len(target), 1)

num\_outputs = 1

num\_inputs = np.size(data[0])

num\_neurons = np.array([num\_inputs, 1])

num\_layers = len(num\_neurons)

scaler = MinMaxScaler()

scaler.fit(data)

inputs = scaler.transform(data)

scaler.fit(target)

target = scaler.transform(target)

#target = target.reshape(1, len(target))

train, test, train\_labels, test\_labels = train\_test\_split(inputs, target, test\_size = 0.2, random\_state=42)

learning\_rate = 0.1

#print(inputs.shape)

np.random.seed(0)

## Otimizando parametros com todo o conjunto de dados, otimizaremos o learning rate

learning\_rate\_param = np.arange(0.01, 1, 0.04) # => 25x

scores = []

for i in learning\_rate\_param:

    model = NeuralNetwork(num\_inputs, num\_outputs, num\_neurons, ["logistic", "linear"], epochs = 100, learning\_rate = i)

    model.fit(train,train\_labels)

    predicted = model.predict(train)

    predicted\_class = np.round(predicted)

    mse = mean\_squared\_error(train\_labels, predicted)

    scores.append(mse)

    print(mse)

scores = np.array(scores)

O resultado do código acima é apresentado na figura 1.

Figura 1 - Taxa de aprendizado adequada sem cross-validation



A segunda abordagem é apresentada abaixo.

train1, test1, train\_labels1, test\_labels1 = train\_test\_split(train, train\_labels, test\_size = 0.33, random\_state=42)

train2, test2, train\_labels2, test\_labels2 = train\_test\_split(train1, train\_labels1, test\_size = 0.5, random\_state=42)

def cross\_validation(x, y, x1, y1, x2, y2, model):

    treino = np.concatenate((x, x1))

    resultado = np.concatenate((y, y1))

    model.fit(treino, resultado)

    predicted = final\_model.predict(x2)

    mse = mean\_squared\_error(y2, predicted)

    return mse

scorex = []

for i in learning\_rate\_param:

    model = NeuralNetwork(num\_inputs, num\_outputs, num\_neurons, ["logistic", "linear"], epochs = 100, learning\_rate = i)

    a = cross\_validation(test1, test\_labels1, train2, train\_labels2, test2, test\_labels2, model)

    b = cross\_validation(test1, test\_labels1, test2, test\_labels2, train2, train\_labels2, model)

    c = cross\_validation(test2, test\_labels2, train2, train\_labels2, test1, test\_labels1, model)

    d = a+b+c/3

    scorex.append(d)

scorex = np.array(scorex)

idx = np.argmin(scorex)

min\_errorr = min(scorex)

learning\_ratee = 0.01 + idx\*0.04

print("Learning\_rate adequado foi: ", learning\_ratee)

O resultado é apresentado na figura 2.

Figura 2 - Resultado com o cross-validation



Com a taxa de aprendizado otimizada para nosso conjunto de treinamento, tratamos de testar a rede com o conjunto de teste. O resultado é apresentado na figura 3.

Figura 3 - Score com o conjunto de teste



Para o calculo do erro foi utilizado o método do *mean-squared-error*, ideal para problemas de regressão, que mede o quão próximo nosso resultado estimado está da resposta certa. Por fim, concluímos que obtivemos um resultado muito satisfatório dado a natureza do problema.

2.4 CLASSIFICADOR COM REDES NEURAIS

Em contraponto ao nosso regressor, construímos um classificador para notarmos as diferenças de técnicas com o uso de redes neurais. A primeira delas se diz em questão a normalização dos dados, como usaremos um dataset binário que, consequentemente, já está normalizado, não será necessário normalizar as saídas do conjunto de dados.

Foi utilizado o dataset do câncer de mama, fornecido pelo SKLearn, que possui 569 amostras com 30 atributos cada amostra. Assim como no regressor, normalizamos os atributos de entrada e caracterizamos nossa rede da seguinte maneira:

* Entradas: 30 (quantidade de atributos)
* Saídas: 1 saída ([0,1] ou [1,0] que determina qual tipo de câncer é)
* Neurônios: 2.

Assim como abordado no exemplo do regressor, otimizamos apenas a taxa de aprendizado do nosso classificador e utilizando apenas o método do cross-validation, visto que foi o método que utilizamos no exemplo anterior. O código é apresentado abaixo.

def cross\_validation\_class(x, y, x1, y1, x2, y2, model):

    treino = np.concatenate((x, x1))

    resultado = np.concatenate((y, y1))

    model.fit(treino, resultado)

    predicted = final\_model.predict(x2)

    predicted\_class = np.round(predicted)

    accuracy = accuracy\_score(y2,predicted\_class)

    return accuracy

data = load\_breast\_cancer()

target = data['target']

data = data['data']

outputs = []

for out\_num in target:

    outlist = [0, 0]

    outlist[out\_num] = 1

    outputs.append(outlist)

outputs = np.array(outputs)

num\_outputs = np.size(outputs[0])

num\_inputs = np.size(data[0])

num\_neurons = np.array([num\_inputs, num\_outputs])

num\_layers = len(num\_neurons)

scaler = MinMaxScaler()

scaler.fit(data)

inputs = scaler.transform(data)

learning\_rate\_param = np.arange(0.01, 1, 0.04)

train, test, train\_labels, test\_labels = train\_test\_split(inputs, outputs, test\_size = 0.2, random\_state=42)

train1, test1, train\_labels1, test\_labels1 = train\_test\_split(train, train\_labels, test\_size = 0.33, random\_state=42)

train2, test2, train\_labels2, test\_labels2 = train\_test\_split(train1, train\_labels1, test\_size = 0.5, random\_state=42)

np.random.seed(0)

## Otimizando parametros com todo o conjunto de dados, otimizaremos o learning rate

scorexx = []

for i in learning\_rate\_param:

    model = NeuralNetwork(num\_inputs, num\_outputs, num\_neurons, ["logistic", "linear"], epochs = 100, learning\_rate = i)

    a = cross\_validation\_class(test1, test\_labels1, train2, train\_labels2, test2, test\_labels2, model)

    b = cross\_validation\_class(test1, test\_labels1, test2, test\_labels2, train2, train\_labels2, model)

    c = cross\_validation\_class(test2, test\_labels2, train2, train\_labels2, test1, test\_labels1, model)

    d = (a+b+c)/3

    print(d)

    scorexx.append(d)

scorexx = np.array(scorexx)

idx = np.argmax(scorexx)

min\_errorr = min(scorexx)

learning\_ratee = 0.01 + idx\*0.04

print("Indice de menor erro foi: ", idx)

print("Menor erro encontrado no conjunto treinamento: ", min\_errorr)

print("Learning\_rate adequado foi: ", learning\_ratee)

O resultado da otimização é apresentado na figura 4.

Figura 4 - Otimização da taxa de aprendizado com cross-validation



Com o melhor valor que descreve a nossa taxa de aprendizado, executamos o nosso classificador com o conjunto de teste para testar a generalização do mesmo. O resultado é apresentado na figura 5.

Figura 5 - Resultado do conjunto de teste



Para aferirmos o resultado de nosso classificador, arredondamos o valor preditado para o número inteiro mais próximo e aplicamos o método de accuracy, fornecido pela biblioteca do SKLearn. Obtivemos um resultado muito satisfatório, visto que a nossa acurácia ficou em torno de 98,2%, ou seja, o nosso modelo conseguiu acertar em quase sua totalidade o conjunto de teste.

3. CONCLUSÃO

Por mais que não tivemos um resultado satisfatório, que fosse de fato, funcional, conseguimos extrair sólidos conhecimentos no processo de construção de um modelo de aprendizado de máquina.

Atuamos ativamente na prevenção das ocorrências conhecidas como overfitting e otimizando o nosso dataset para o resultado do nosso modelo, porém, como já foi dito anteriormente, o modelo de K-Neareast-Neighbors não foi satisfatório para o contexto de nosso problema.

Por fim, reforçamos alguns conceitos vinculados a aritmética intervalar, que é a introdução a lógica fuzzy, próximo conteúdo de nosso cronograma.