UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL (UFRGS)
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA (DELET)
TÓPICOS ESPECIAIS EM INSTRUMENTAÇÃO (ENG04019)

MATHEUS QUEVEDO SIVELLI

TRABALHO 6 – RADIAL BASIS FUNCTION NEURAL NETWORK

Porto alegre
2020

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

Inteligência artificial e o aprendizado de máquina são usualmente definidos como o futuro da humanidade. Dentro desse pilar, o aprendizado profundo – deep learning, em inglês – tem se destacado e criado espaço em diversas aplicações como reconhecimento facial, visão computacional, processamento natural de linguagem e outras.

As redes neurais são os principais elementos entre a vertente de deep learning e sua abordagem mais clássica como as famosas MLP são excelente métodos para utilizarmos em problemas de classificação, porém não apresentam o mesmo nível de performance em problemas de regressão, e para solucionar esse problema existem as redes neurais de base radial, que fazem uso da distância dos centros – variáveis definidas por algum método, seja aleatório ou via algoritmos de clusterização – e seus respectivos valores de entrada. As RBF's podem ser definidas como um caso especial das MLP's pois possuem, em sua essência, apenas uma camada oculta e com pesos apenas na camada de saída.

Esse trabalho está estruturado em quatro partes. A primeira, é a abordagem de um clássico problema de interpolação, onde não aplicamos técnicas de aprendizado de máquina. Já a segunda parte consiste em aplicar métodos de aprendizado de máquina para estimar uma função que seja eficiente em representar o conjunto de dados. Por fim, as partes finais possuem o objetivo em comum de analisar os conteúdos recentemente publicados dentro da comunidade cientifica sobre o tema.

2. EXERCÍCIOS

2.1 INTRODUÇÃO TÉORICA

As redes neurais de base radial podem ser classificadas como um caso específico das MLP's contendo apenas uma camada oculta e a na camada de saída contendo apenas um neurônio de saída. Além disso, as RBF's não apresentam biases em sua camada escondida e não utilizam o conceito de pesos, por outro lado, observamos o uso de centros, que são medidas arbitrárias que podem ser definidas aleatoriamente ou por algum algoritmo de agrupamento como o K-Means, os centros são utilizados para medir a distância entre o valor de entrada e o centro respectivo do neurônio. Por fim, os pesos são apenas aplicados na camada de saída e as funções de ativação da camada oculta são, geralmente, funções gaussianas, também conhecida como distribuição normal. O código base para todo trabalho 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 RBFNeuralNetwork:
    '''simple feedforward neural network class'''
    def __init__(self,num_inputs,num_outputs,num_neurons,tf_functions,epochs=100,learning_rate=
0.1, mode = 'sequential'):
        self.mode = mode
        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] == "gaussian":
                self.tf_functions.append(self.tf_gaussian)
                #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
        #*******************************
        centers_per_layer = []
        weights_per_layer = []
        biases_per_layer = []
        wsums_per_layer = []  # for helping later activation function approximate derivative ca
lculation

        for l in range(0,len(num_neurons)):
            previous_layer_is_input = 0
            if (l == 0):
                previous_layer_size = len(inputs[0])
                previous_layer_is_input = 1
            else:
                previous_layer_size = num_neurons[l-1]
```

```
 previous_layer_size = num_neurons[l-1]

            layer_size = num_neurons[l]

            if (not previous_layer_is_input):
                layer_weights = np.zeros((layer_size,previous_layer_size))
                layer_biases = np.zeros(layer_size)
            else:
                layer_weights = np.zeros((layer_size,0))
                layer_biases = np.zeros((0))
                layer_centers = np.zeros((layer_size, previous_layer_size))

                self.centers_per_layer = layer_centers

            weights_per_layer.append(layer_weights)
            biases_per_layer.append(layer_biases)
            wsums_per_layer.append(np.zeros(layer_size))

        self.weights_per_layer = weights_per_layer
        self.biases_per_layer = biases_per_layer
        self.wsums_per_layer = wsums_per_layer

        self.randomize_weights()
        self.randomize_biases()
        self.randomize_centers()
```

```
    def set_last_layer_weights(self,weights):
        self.weights_per_layer[-1] = [weights]

    def set_weights_interpolation_matrix(self, inputs, outputs):
        interp_matrix = np.zeros((inputs.shape[0], self.centers_per_layer.shape[0]))
        for j in range (0,len(inputs)):
            for i in range(0,len(self.centers_per_layer)):
                phi = self.neuron_rbf(self.centers_per_layer[i], inputs[j], self.tf_functions[0
], 0, i)
                interp_matrix[j,i] = phi

        interp_matrix = np.array(interp_matrix)

        W = np.linalg.pinv(interp_matrix).dot(outputs)
        self.weights_per_layer[1][0] = W

    def set_centers(self, centers):
        self.centers_per_layer = centers
```

```
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)
            self.biases_per_layer[i] = self.biases_per_layer[i]*0

    def randomize_centers(self):
        a = self.min_weight_value
        b = self.max_weight_value
        for i, layer in enumerate(self.centers_per_layer):
            self.centers_per_layer[i] = a+(b-
a)*np.random.random(self.centers_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_rbf(inputs,self.weights_per_layer[i],self.biases_p
er_layer[i],self.centers_per_layer,self.tf_functions[i],i)
                #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_rbf(self,inputs,weights,biases,centers,tf_function,layer_index):
        outputs = np.zeros(len(weights))
        for i in range(0,len(weights)): #for all neurons
            outputs[i] = self.neuron_rbf(centers[i],inputs,tf_function,layer_index,i)
        return outputs


    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_gaussian(self, x, sigma=5):
        return np.exp(-((x)/sigma)**2)

    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 neuron_rbf(self,centers,inputs,tf_function,layer_index,neuron_index):
        summing_section = inputs - centers
        norm_section = np.linalg.norm(summing_section, ord=2)
        output = tf_function(norm_section, sigma = 1)

        # update for later calculate activation function approximate derivative
        self.wsums_per_layer[layer_index][neuron_index] = norm_section

        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 sol
ution
            #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

        return 1

    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
                   break
                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_gradi
ents[l][j] * self.outputs[l-1][p]

                # bias update
                self.biases_per_layer[l][j] += self.learning_rate * self.local_gradients[l][j]
* 1

    def num_derivative(self,f,x,delta=1e-6):
        return (f(x+delta)-f(x))/delta
```

2.2 PROBLEMA 1 – INTERPOLAÇÃO

Neste problema são criados 13 pontos aleatórios com seus respectivos valores e é utilizado uma RBFNN para fazer a interpolação desses pontos, que é encontrar a curva que melhor representa o conjunto. Para que o problema seja de interpolação, devemos utilizar os centros exatamente iguais aos pontos de entradas, fazendo com que cada entrada se conecte em cada neurônio da camada oculta e cada neurônio possua um centro que seja equivalente a uma entrada.

Aplicando a técnica de interpolação, encontramos uma matriz quadrada que possuem exatamente a saída da camada oculta. Com a inversa da matriz quadrada e a multiplicação da saída, conseguimos obter exatamente os pesos que representam a aproximação da função, caracterizando assim, a interpolação.

Utilizamos o código apresentado na seção 2.1 e aplicamos a seguinte técnica para provar as constatações acima. O código e resultados são obtidos abaixo.

```
inputs = np.array([[-2,-2], [-2,-1], [-2,0], [-1,-2], [-1,-1], [-
1,0], [0,0], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]])
outputs = np.array([-5, -3, 2, -1, 2, -4, -6, 2, 10, 7, -8,  4, 2])

num_outputs = np.size(outputs[0])
num_inputs = np.size(inputs[0])

num_neurons = np.array([13, num_outputs])

num_layers = len(num_neurons)

model = RBFNeuralNetwork(num_inputs, num_outputs, num_neurons, ["gaussian", "linear"], epo
chs = 5, learning_rate = 0.1, mode = 'sequential')

model.set_centers(inputs)
model.set_weights_interpolation_matrix(inputs, outputs)

model.calc_output(inputs[1])

for i in range (0,len(inputs)):
    print("real output: %f \t predicted output: %f" %(outputs[i], model.calc_output(inputs
[i])))
```

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

x0_sampled = np.arange(-2, 2, 0.1)
x1_sampled = np.arange(-2, 2, 0.1)

x0, x1 = np.meshgrid(x0_sampled, x1_sampled)
z = np.zeros_like(x0)

for i, x0i in enumerate(x0_sampled):
    for j, x1i in enumerate(x1_sampled):
        z[i,j] = model.calc_output([x0i, x1i])

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')

ax.set_xlabel("x0")
ax.set_ylabel("x1")
ax.set_zlabel("output")

surf = ax.plot_surface(x0, x1, z, rstride = 1, cstride = 1, cmap = 'viridis',
 linewidth = 0.1)

x0s = inputs[:,0]
x1s = inputs[:,1]
zs = outputs
ax.scatter(x0s, x1s, zs, marker = 'o', c = 'gray', s = 50, depthshade = 0)

plt.show()
```

A saída numérica da rede é apresentada na figura 1.

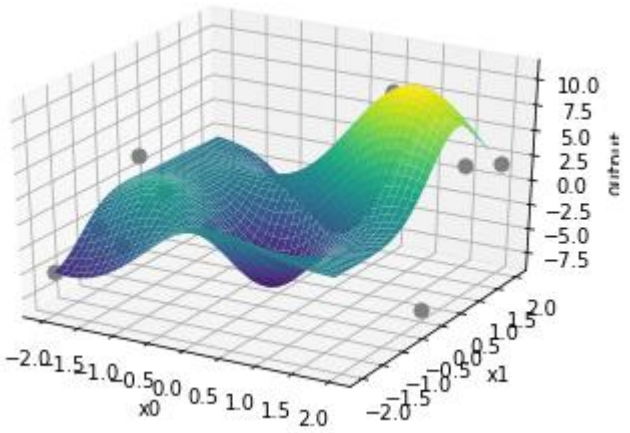
Figura 1 - Saída da RBF

```
real output: -5.000000   predicted output: -5.000000
real output: -3.000000   predicted output: -3.000000
real output: 2.000000    predicted output: 2.000000
real output: -1.000000   predicted output: -1.000000
real output: 2.000000    predicted output: 2.000000
real output: -4.000000   predicted output: -4.000000
real output: -6.000000   predicted output: -6.000000
real output: 2.000000    predicted output: 2.000000
real output: 10.000000   predicted output: 10.000000
real output: 7.000000    predicted output: 7.000000
real output: -8.000000   predicted output: -8.000000
real output: 4.000000    predicted output: 4.000000
real output: 2.000000    predicted output: 2.000000
```

Fonte: Elaboração própria

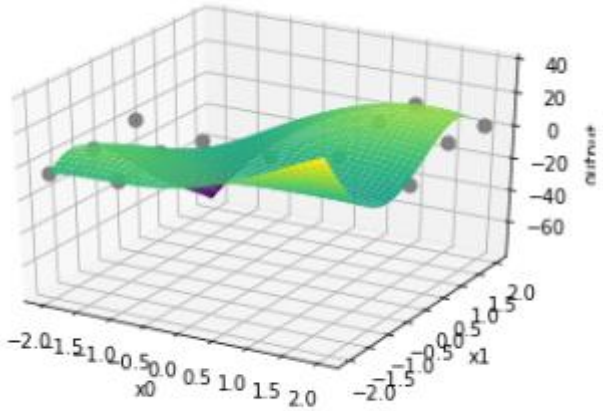
Os resultados gráficos são apresentados abaixo.

Figura 2 - Saída com a largura da função radial = 1



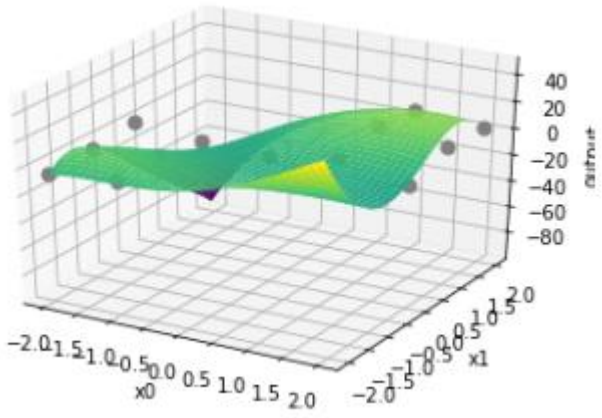
Fonte: Elaboração própria

Figura 3 - Saída com a largura da função radial = 5



Fonte: Elaboração própria

Figura 4 - Saída com a largura da função radial = 10



Fonte: Elaboração própria

Analisando os diferentes valores de sigma, conseguimos perceber que quanto menor a largura aplica na função radial, o comportamento é mais oscilatório e, portanto, a função tende a contemplar muito mais pontos do conjunto que está sendo aproximado a uma função. Já quando aumentamos essa largura, conseguimos perceber que o efeito oscilatório não aparece mais e, consequentemente, tende a não contemplar todos os pontos que estão sendo considerados no problema.

Esse é um clássico caso de otimização de parâmetros, idealmente, a largura da função radial deve ser extraída de um algoritmo não supervisionado de categorização como o ***k-means***.

Por fim, conseguimos extrapolar a ideia até a generalização dessa função, uma largura de base radial específica ao conjunto de dados, pode afetar pequena pode afetar diretamente o desempenho da rede em uma eventual generalização.

2.3 PROBLEMA 2 – APRENDIZADO DE MÁQUINA

Diferente do problema de interpolação, nessa abordagem usaremos técnicas de aprendizado de máquina para determinar os centros e os respectivos pesos entre a saída da camada oculta e a camada de saída. Na seção 2.2, foi preciso utilizar o número de neurônios na camada oculta igual a quantidade de valores na camada de entrada e os centros foram definidos iguais aos valores de entrada. Já nesta abordagem, usaremos um algoritmo não supervisionado, o kmeans, para encontrar os centros dos clusters que melhor representam o nosso conjunto de dados e um número inferior de neurônios na camada oculta.

Como a RBF clássica é uma rede neural feed forward e não possui pesos entre a camada de entrada e a camada oculta, não foi preciso implementar o backpropagation em sua completude. Utilizamos a técnica de backpropagation adaptada do clássico problema de MLP para atualizar os pesos. A adaptação se ateve a freiar o algoritmo de retroceder até a camada de entrada da rede neural, fazendo com que os somente os pesos existentes entre a camada oculta e a camada de saída fossem atualizados. A implementação da adaptação foi simples, abaixo destacamos o que foi preciso alterar.

- ➔ A função update weights não precisava mais acoplar a atualização dos pesos da camada de entrada.
- ➔ Atualizamos a função derivada da função de ativação para o caso da RBF.

Após ajustes finos dentro da nossa implementação, tratamos de testar a técnica em um problema clássico de regressão. Utilizamos o dataset fornecido pelo SKLearn, o load_boston, conjunto que determina o preço de um imóvel em Boston. Foram adotadas todas boas práticas em problemas de aprendizado de máquina, como separação dos conjuntos teste e treinamento, normalização dos dados, assim como o uso de uma otimização de hiperparametros simplista.

Utilizamos o método do cotovelo, prática que calcula o ponto mais distante da reta traçada entre o maior e menor ponto possível para quantidade de cluster, para determinar qual é a quantidade de clusters que caracterizam nosso conjunto de dados. O código, junto com o tratamento dos dados e escolha dos centros é apresentado abaixo.

```
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
from sklearn.cluster import KMeans

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([4, 1])
num_layers = len(num_neurons)

scaler = MinMaxScaler()
scaler.fit(data)
inputs = scaler.transform(data)
scaler.fit(target)
target = scaler.transform(target)

wcss = []
for i in range(2,10):
    kmeans = KMeans(n_clusters = i, random_state=0).fit(inputs)
    wcss.append(kmeans.inertia_)

from math import sqrt
x1, y1 = 2, wcss[0]
x2, y2 = 8, wcss[len(wcss)-1]

distances = []
for i in range(len(wcss)):
    x0 = i+2
    y0 = wcss[i]
    numerator = abs((y2-y1)*x0 - (x2-x1)*y0 + x2*y1 - y2*x1)
    denominator = sqrt((y2 - y1)**2 + (x2 - x1)**2)
    distances.append(numerator/denominator)

kmeans = KMeans(n_clusters = (distances.index(max(distances))+2), random_state=0).fit(inputs)
kmeans.cluster_centers_
centers = kmeans.cluster_centers_
print(centers)
```

Os centros são apresentados na figura 5.

Figura 5 - Coordenadas dos centros de cada cluster

```
[[7.89155733e-04 4.48409091e-01 1.51163023e-01 4.54545455e-02
  8.66030677e-02 5.87548991e-01 2.61351933e-01 4.97829471e-01
  1.39130435e-01 2.26578765e-01 4.87911025e-01 9.80133322e-01
  1.30656733e-01]
 [1.43340318e-01 2.22044605e-16 6.46627566e-01 6.06060606e-02
  5.91392318e-01 4.71575305e-01 8.95008270e-01 8.47196744e-02
  1.00000000e+00 9.14122137e-01 8.08510638e-01 7.25627028e-01
  4.65524450e-01]
 [1.40612565e-02 9.71445147e-17 7.67085839e-01 1.52542373e-01
  6.02043663e-01 4.69152802e-01 9.46499328e-01 7.33451692e-02
  1.43699337e-01 4.21626342e-01 5.09556437e-01 8.77618896e-01
  3.98496838e-01]
 [3.30717295e-03 3.98780488e-02 2.47786281e-01 6.34146341e-02
  2.61305832e-01 5.34182326e-01 6.80520459e-01 2.55479626e-01
  1.57582185e-01 2.10603240e-01 6.08510638e-01 9.72187201e-01
  2.59415549e-01]]
```

Fonte: Elaboração própria

Abaixo é apresentado o código que utilizamos para otimizar a taxa de aprendizado.

```
train, test, train_labels, test_labels = train_test_split(inputs, target, test_size = 0.2,
 random_state=42)

np.random.seed(0)
learning_rate_param = np.arange(0.01, 1, 0.04)

scores = []
for i in learning_rate_param:
    model = RBFNeuralNetwork(num_inputs, num_outputs, num_neurons, ["gaussian", "linear"],
 epochs = 100, learning_rate = i, mode = 'sequential')
    model.set_centers(centers)
    model.fit(train, train_labels)
    predit = model.predict(train)
    mse = mean_squared_error(train_labels, predit)
    scores.append(mse)

scores = np.array(scores)

idx = np.argmin(scores)
min_error = min(scores)
learning_rate = 0.01 + idx*0.04

print("Menor erro encontrado no conjunto treinamento: ", min_error)
print("Learning_rate adequado foi: ", learning_rate)
```

Extraímos o seguinte valor, apresentado na figura 6.

Figura 6 - Resultado da otimização da taxa de aprendizado

```
Menor erro encontrado no conjunto treinamento:  0.031156530206982933
Learning_rate adequado foi:  0.01
```

Fonte: Elaboração própria

Com a melhor taxa de aprendizado avaliada dentro do nosso intervalo, avaliamos o modelo no conjunto de teste. O resultado é apresentado na figura 7.

Figura 7 - Resultado do conjunto de teste na rede neural

```
0.025468385403748173
```

Fonte: Elaboração própria

Por fim, utilizamos um sigma equivale a cinco, por acreditarmos que quanto maior a largura de base da função de ativação, mais genérico o modelo é propenso a ser e, portanto, contém mais importância dentro do mundo corporativo. Idealmente, o valor de largura da gaussiana é extraído do próprio algoritmo de clusterização, porém, como não era o objetivo deste documento, decidimos não explorar essa vertente e focar na correto adaptação do backpropagation para uma rede neural de base radial.

Concluímos que a nossa rede funcionou de acordo com o esperado, atingindo um resultado extremamente satisfatório quando exposta a um conjunto de dados desconhecido, o conjunto de teste.

2.4 PERFORMANCE EVALUATION OF RADIAL BASIS FUNCTION NETWORKS BASED ON TREE SEED ALGORITHM

A presente seção apresentará um resumo do artigo “Performance Evaluation of Radial Basis Function Networks Based on Tree Seed Algorithm” escrito por V. Muneeswaran e DR.M.Pallikonda Rajasekaran[1]. No paper, os autores buscaram desenvolver uma Radial Basis Function Neural Network (RBFN) otimizada por um Tree Seed Algorithm (TSA).

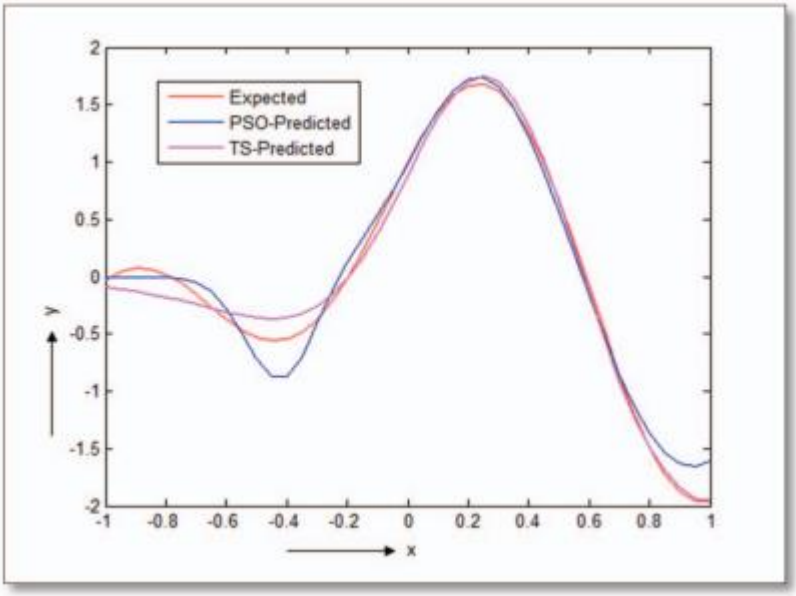
As RBFNN, radial basis functions neural network, não diferem de qualquer outro caso de inteligência artificial e grande partes de seus segredos estão no ato de otimização dos hiperparametros da rede. Portanto, o artigo em questão testa uma abordagem mista, utilizando o algoritmo ***tree seed algorithm,*** algoritmo baseado em 'populações' e idealmente usado para problemas de otimização, para determinar os parâmetros como centros, largura da função gaussiana e os pesos entre a camada de saída e a camada oculta.

Os autores iniciam o mesmo número de populações do TSA (tree seed algorithm) com a mesma quantidade de camadas ocultas da RBFNN, após isso, é definido o range máximo dos hiperparametros que serão otimizados, como largura da função radial, centros, pesos e inicia todos os valores. Com os hiperparametros inicializados, é determinado o valor da árvore e das folhas, que aplicando o processo de fit, neste resultado é determinado se existe convergência. Esse processo é repetido até encontrar a condição de terminação, depois é replicado para diferentes valores até que se encontre os melhores hiperparametros para o problema.

Para apurar os resultados é utilizado uma outra técnica de otimização de parâmetros, o ***particle swarm optimization*** e o TSA e fazendo a aproximação de uma função. A figura a baixo é a regressão da função com os dois métodos de otimização.

[1] O artigo pode ser encontrado em https://ieeexplore.ieee.org/document/7530267

Figura 8 - Resultado gráfico do experimento



Fonte: Muneeswaran e Rajasekaran (2016).

Como podemos perceber, o TSA teve uma performance melhor na regressão da respectiva função. A figura 2 evidencia ainda mais a diferença entre os dois métodos demonstrando os resultados numéricos de cada abordagem.

Figura 9 - Resultado numérico do experimento

unctions	No.of Hidden Layers	PSO based RBFN		TSA based RBFN	
		Fitness Values		Fitness Values	
		Mean	Standard Deviation	Mean	Standard Deviation
F1	3	3.844	3.151	0.268	0.177
	6	1.894	1.081	0.538	0.194
	9	1.172	0.981	0.298	0.161
	12	2.416	2.679	0.274	0.139

Fonte: Muneeswaran e Rajasekaran (2016).

Por fim, conseguimos analisar tanto graficamente e numericamente que o TSA se mostrou um recurso melhor para otimizar problemas de aproximação de funções do que o outro método comparado.

2.5 PARALLEL RADIAL BASIS FUNCTION NEURAL NETWORKS TO SOLVE THE POLYMONIALS EQUATIONS

A presente seção apresentará um resumo do artigo “Parallel Radial Basis Function Neural Networks to solve polymonials equations” escrito por Abed Ali H. Altaee, Haider K. Hoomod e Khalid Ali Hussein[2].

O problema de encontrar raízes polinomiais e aproximar funções são um ramo importante da matemática e tecnologia, para isto, o autor propõe um uso de RBFNN para abordar esse problema. As RBFNN são redes neurais feed forwards que possuem, em sua maioria, apenas uma camada oculta. Nesta camada oculta a função de ativação é a função gaussiana e não possui pesos associados ao núcleo de processamento do neurônio, este conceito é tomado pelos centros, valor que desempenha um papel semelhante aos pesos nas MLPS. Os centros são utilizados para medir, usualmente, a distância euclideana entre os valores de entrada e os centros. Dentro deste contexto, o autor trás abordagens modificadas para os problemas de otimização e para as funções de ativação.

A metodologia de implementação do autor se resume em uma abordagem mista entre o método clássico gaussiano e o método de Newton-Rapshon, técnica de estimar numericamente raízes de funções, aplicado nas funções de ativação, gradient descent e como na maneira de selecionar os centros.

Basicamente, o autor inicializa os parâmetros da rede sendo maiores que zero e atribui uma tolerância de erro. Aplica-se o grandient decent modificado para encontrar os valores otimizados até que a tolerância seja maior que o erro.

O autor ainda compara a abordagem de utilizar apenas uma RBFNN modificada com o uso delas em paralelo, aplicando exatamente a mesma função que deve ser solucionada com a mesma tolerância a ser permitida. Os resultados apresentam que as redes em paralelo foram 4 vezes mais velozes que apenas uma RBFNN e ainda apresentando um resultado mais assertivo.

Por fim, concluímos que a abordagem das RBFNN modificadas, em paralelo especialmente, podem ser uma alternativa interessante para a aproximação de funções não lineares, tendo registrado o erro muito próximo de zero.

[2] O artigo pode ser encontrado em https://ieeexplore.ieee.org/document/7759938

3. CONCLUSÃO

Por fim, notamos que em ambas abordagens exploradas neste documento tem suas peculiaridades, mas que, ao término, sempre caímos em um problema clássico de inteligência artificial, a otimização. Relembramos a extrema importância de utilizar uma largura de função radial assertiva, visto que tal resultado influência diretamente na operacionalização do método.

Também percebemos ao decorrer deste trabalho, que as redes neurais com funções de base radial continuam sendo um ramo a ser explorado dentro da ciência moderna, possuindo artigos recentes que tratam sobre os principais problemas em algoritmos de machine learning, a otimização.

Além disso, é possível ver o importantíssimo papel que as RBFNN desempenham no ramo matemático em aproximar funções complexas, tendo um desempenho invejável quando otimizado da maneira correta.