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

LISTA DE FIGURAS

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. O conceito de neurônio é relacionado a uma unidade de processamento que possui 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

Faremos o calculo para apenas uma época e iniciando os pesos em zero. Inicialmente calcularemos a saída do somador em ambos neurônios apresentados na figura 1.

Figura 1 - Calculando saída do somador

```
Formula:
Saída do somador = Entrada*pesos + bias

Neuronio 1 = 0*0 + 0 = 0
Neuronio 1 = 1*0 + 0 = 0
Neuronio 2 = 0*0 + 0 = 0
Neuronio 2 = 1*0 + 0 = 0
```

Fonte: Elaboração própria

Após isso, calcularemos a saída da célula passando o valor na função de ativação, que é a função logística. A figura 2 exemplifica o método.

Figura 2 - Calculando saída da célula

```
Função logística = 1/(1+exp(x*2))
Aplicando valor = 0 na função, obtemos

1 / 1 e^0 = 1 / 2 = 0.5

Saida do neurônio = 0.5
```

Fonte: Elaboração própria

Em seguida, calcularemos o somador da camada de saída, como apresentado na figura 3.

Figura 3 - Calculando somador de saída

```
Somador Saida = Somador 1 + Somador 2
Somador 1 = 0.5 * 0 + 0
Somador 2 = 0.5 * 0 + 0
Somador Saida = 0
```

Fonte: Elaboração própria

Após o calculo do somador na última célula, aplicamos na função de ativação, linear nesse caso, e calculamos o erro, como é apresentado na figura 4.

Figura 4 - Calculando saída e erro

```
f(x) = x
Saida obtida  = 0

erro = saida esperada - saida obtida
erro = 1 - 0
erro = 1
```

Fonte: Elaboração própria

Com o erro da camada de saída em mãos, calcula-se o gradiente local. A figura 5 representa o cálculo.

Figura 5 - Calculando gradiente da saída

```
Gradiente local = erro * derivada da função de ativação
Gradiente Local = 1 * 1
gradiente local da saída = 1
```

Fonte: Elaboração própria

Com o gradiente local na camada de saída, calcula-se o gradiente local em cada neurônio. Para fazer isso, calcula-se a derivada da função de ativação local e multiplica-se pelo gradiente posterior e o peso anterior, conforme ilustra a figura 6. O procedimento é igual em ambos os neurônios.

Figura 6 - Calculando gradiente local nos neurônios

```
Gradiente local da camada oculta = Derivada da função de ativação * (Gradiente de saída * peso)
Derivada no ponto zero = (1/2)/2 = 0.25
Gradiente local do neurônio = 0.25 * (1*0)

-> esse procedimento é repetido para o outro neurônio <-
```

Fonte: Elaboração própria

Após achar o gradiente local do neurônio, atualiza-se os pesos conforme a figura 7 demonstra.

Figura 7 - Atualizando os pesos

```
taxa de aprendizado = 0.1
delta peso = taxa de aprendizado * gradiente da célula * saída da celula

novo peso = peso anterior + delta peso
novo peso (bias saída) = 0 + (0.1 * 1 * 1) = 0.1
novo peso (neurônios) = 0 + (0.1 * 1 *0.5) = 0.05
novo peso (bias) = 0 + (0.1 * 0 * 1) = 0

novos pesos de entrada = 0 + (0.1 * 0 * 1) = 0
novos pesos de entrada = 0 + (0.1 * 0 * 0) = 0
```

Fonte: Elaboração própria

Com os novos pesos, aplica-se o mesmo método acima para obtenção da nova saída e posteriormente executar novamente os passos aqui demonstrado. A figura 8 ilustra a obtenção da nova saída com os pesos atualizados.

Figura 8 - Calculando nova saída com novos pesos

```
Neuronio 1 = 0*0 + 0 = 0
Neuronio 1 = 1*0 + 0 = 0
Neuronio 2 = 0*0 + 0 = 0
Neuronio 2 = 1*0 + 0 = 0

Saida do neurônio = 0.5

Entrada 1 na camada de saída = 0.5 * 0.05 = 0.025
Entrada 2 na camada de saída = 0.5 * 0.05 = 0.025

Somador saida = 0.1 + 0.25 + 0.25 = 0.15

Função linear: f(x) = x
f(0.15) = 0.15

Saida obtida = 0.15
erro = 1 - 0.15
erro = 0.85
```

Fonte: Elaboração própria

Para compararmos a nossa implementação, executaremos os mesmos parâmetros no nosso código e obtivemos o seguinte resultado.

Figura 9 - Resultado da implementação com os mesmos parâmetros

```
Biases per layer:  [array([0., 0.]), array([0.1])]
Pesos por layer:  [array([[0., 0.],
       [0., 0.]]), array([[0.05, 0.05]])]
Local Gradients:  [array([0., 0.]), array([1.])]

Out[21]: array([0.15])
```

Fonte: Elaboração própria

Resultado que condiz com o que calculamos manualmente e corrobora a assertividade da implementação do nosso método.

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, learn
ing_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 10.

Figura 10 - Taxa de aprendizado adequada sem cross-validation

```
Indice de menor erro foi:  10
Menor erro encontrado no conjunto treinamento:  0.004647158838610101
Learning_rate adequado foi:  0.41000000000000003
```

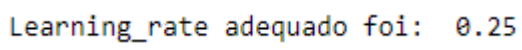
Fonte: Elaboração própria

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, rando
m_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, learn
ing_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 11.

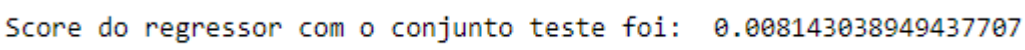
Figura 11 - Resultado com o cross-validation



Fonte: Elaboração própria

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 12.

Figura 12 - Score com o conjunto de teste



Fonte: Elaboração própria

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_s
tate=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, learnin
g_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 13.

Figura 13 - Otimização da taxa de aprendizado com cross-validation

```
Indice de menor erro foi:  2
Menor erro encontrado no conjunto treinamento:  0.43345532705937034
Learning_rate adequado foi:  0.09
```

Fonte: Elaboração própria

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 14.

Figura 14 - Resultado do conjunto de teste

```
Acurácia do modelo com conjunto de teste:  0.9824561403508771
```

Fonte: Elaboração própria

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

Analisando os resultados obtidos dos nossos problemas de regressão e classificação, conseguimos notar um desempenho excelente da nossa rede neural, apesar de utilizarmos um código caseiro. Também conseguimos reforçar toda a teoria por trás das redes neurais e do método de *backpropagation*, que trouxe a viabilidade de implementação a uma rede neural.

Além disso, notamos que o poder computacional já começa a ser uma variável em nossas construções, por exemplo, a otimização dos parâmetros teve seu tempo médio de execução em torno de 45 minutos. Uma das explicações para o prolongado tempo de execução é a utilização de métodos caseiros, caso tivesse sido abordado bibliotecas consolidadas, a eficiência do problema poderia ter sido incrementada.