Andrés Eduardo Marchetti Colognesi - 9838161

Relatório Atividade 4 - Inteligência Computacional

1 Questão 1)

1.1 Itens a) e b)

O código que implementa a construção, treinamento com gradiente descendente, e predição de uma rede neural de L camadas, com ativações *tanh()*, está implementado no Algoritmo 1. Além disso, o código plota a evolução da função de custo MSE, assim como a evolução dos valores dos pesos e viés para cada neurônio. Também plota a região de classificação e a região de regressão. O Algoritmo 1 está no final do relatório.

2 Questão 2)

Para este exercício, utilizou-se a estrutura programada para a Questão 1, para programar a rede neural 2-2-1 e treiná-la para realizar a operação XOR. A parte que implementa esta questão e faz a plotagem dos gráficos está no final do Algoritmo 1.

2.1 Item a)

As regiões de glassificação e de regressão determinadas pelo algoritmo treinado estão representadas, respectivamente, nas Figuras 1 e 2. A região de classificação determina o pertencimento de cada ponto do espaço às classes 0 ou 1, sendo estas as saídas lógicas do XOR. Já a região de regressão retrata a gradação de valores de probabilidade de cada ponto. Esta probabilidade é levada em conta na hora de realizar a classificação final.

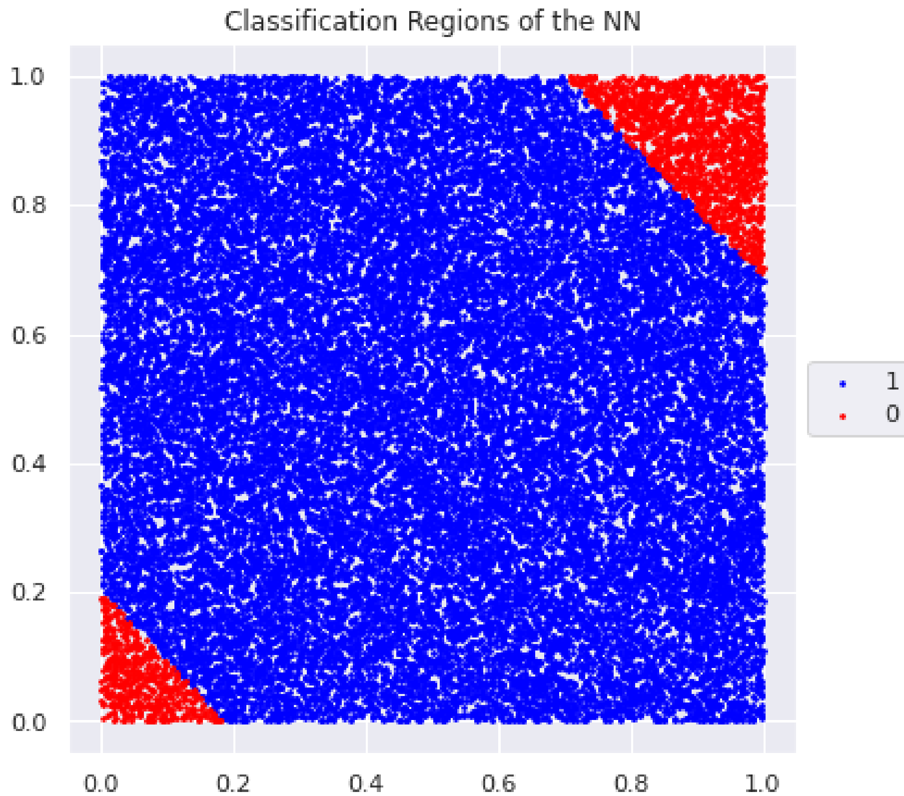

Figura 1: Regiões de classificação determinadas pelo algoritmo treinado.

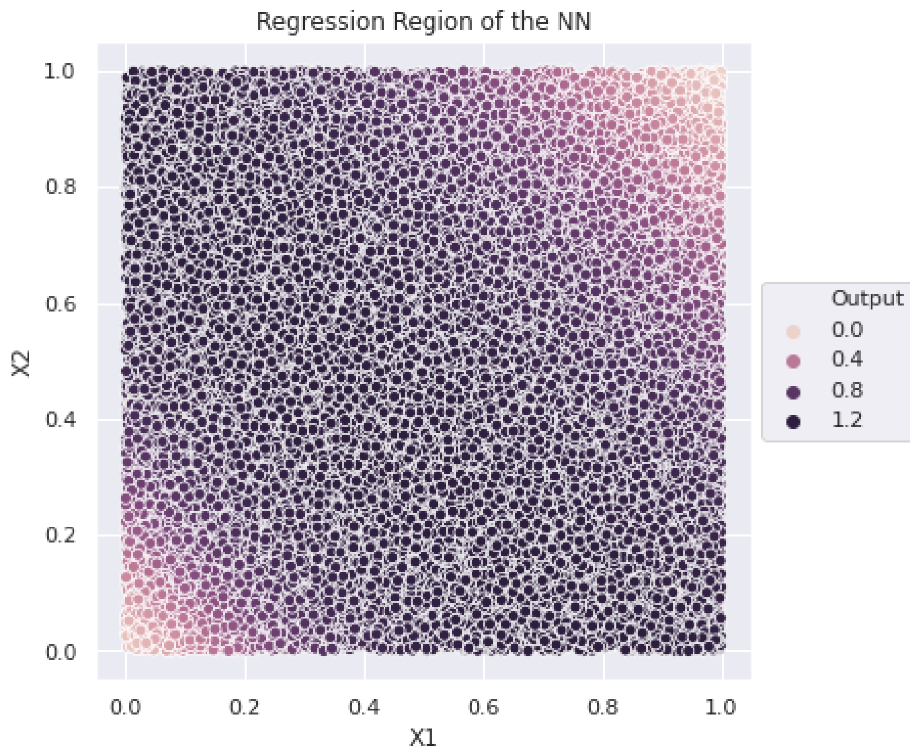

Figura 2: Regiões de regressão determinadas pelo algoritmo treinado.

2.2 Item b)

A evolução da função de custo MSE com as épocas de iteração está representada na Figura 3. Note que o custo atinge a tolerância de 0.001 em 372 iterações.

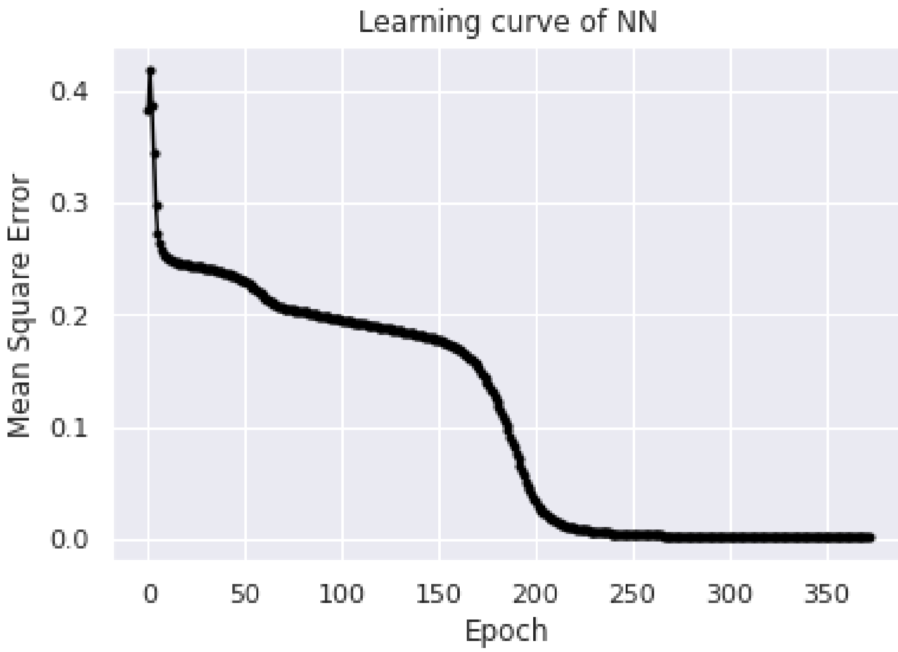

Figura 3: Evolução da função de custo MSE com as iterações de aprendizagem.

2.3 Item c)

Nas Figuras 4, 5 e 6 está exposto a evolução dos pesos e do viés para cada neurônio da rede, em relação ao número de iterações de aprendizagem do algoritmo.

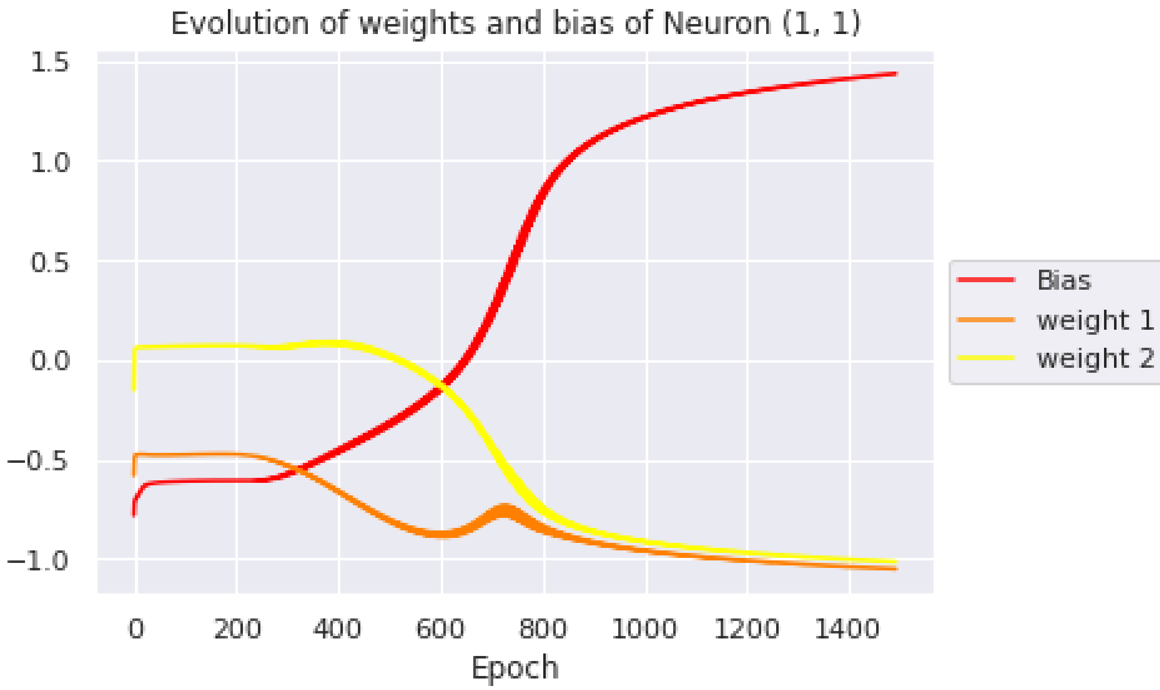

Figura 4: Evolução dos pesos e do viés para o primeiro neurônio da camada 1.

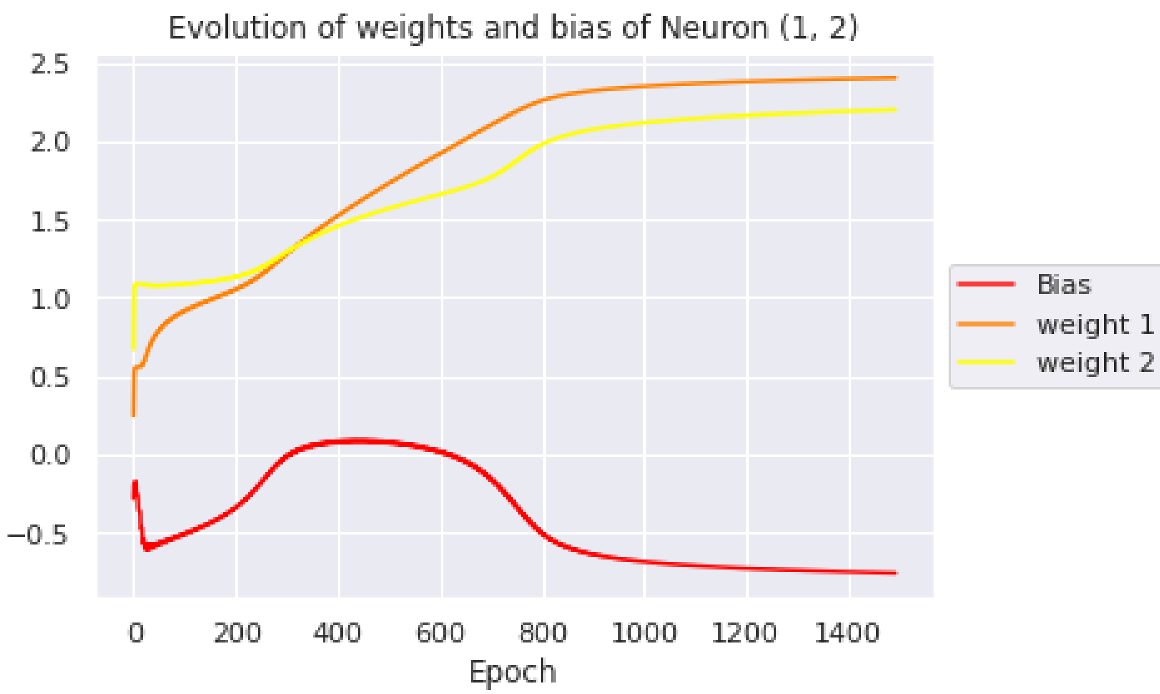

Figura 5: Evolução dos pesos e do viés para o segundo neurônio da camada 1.

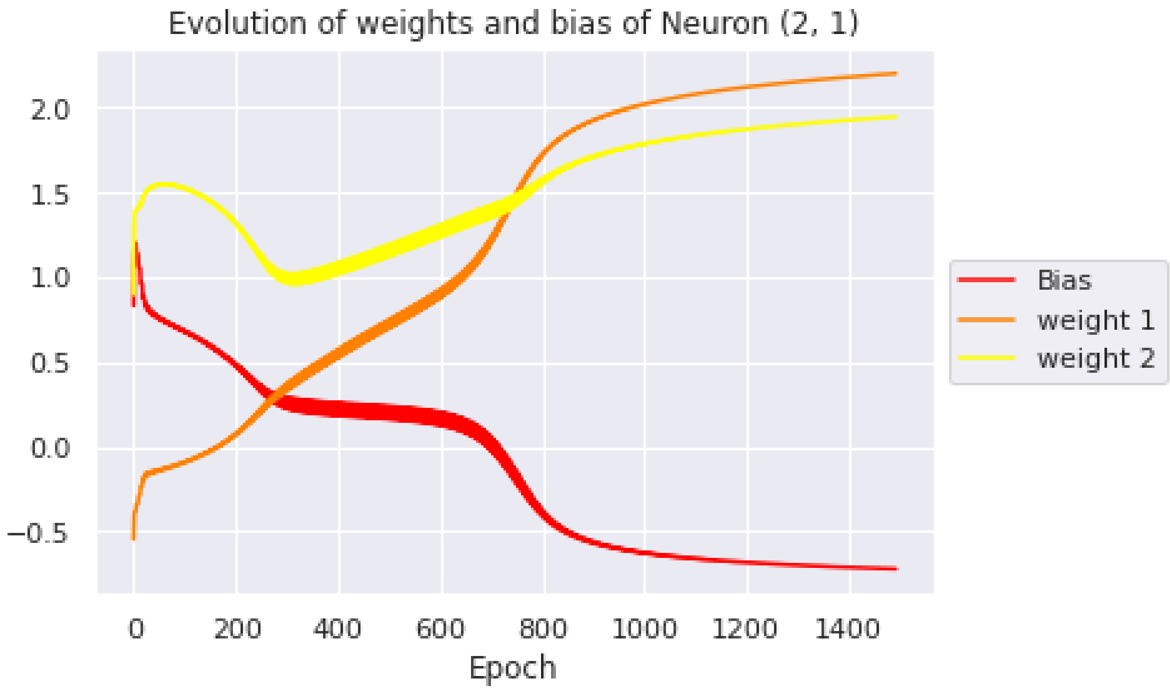

Figura 6: Evolução dos pesos e do viés para o neurônio da camada 2 (neurônio de saída).

3 Algoritmo

```
# Andres E. M. Colognesi - 9838161

#                    ##############################
#                    ## Atividade 4 - Int. Comp. ##
#                    ##############################

#                   ------  Importing libraries  ------

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()


#              ------  Exercise 1 - Helper functions  ------


def initialize_parameters_deep(layer_dims):
    """
```

```
    Arguments:
    layer_dims -- python array (list) containing the dimensions of
       each layer in our network

    Returns:
    parameters -- python dictionary containing your parameters
       "W1", "b1", ..., "WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l],
                       layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)    # number of layers in the network

    for l in range(1, L):
        parameters["W" + str(l)] = np.random.randn(layer_dims[l],
           layer_dims[l-1]) * 0.01
        parameters["b" + str(l)] = np.zeros((layer_dims[l], 1))
        #Verify dimensions:
        assert(parameters['W' + str(l)].shape == (layer_dims[l],
           layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l],
           1))

    return parameters


def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of
       previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current
       layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current
       layer, 1)

    Returns:
    Z -- the input of the activation function, also called
       pre-activation parameter
```

```
    cache -- a python dictionary containing "A", "W" and "b" ;
       stored for computing the backward pass efficiently
    """

    Z = np.dot(W, A) + b
    #Verify shape:
    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache


def my_tanh(Z):
    """
    Implements the tanh activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of tanhh(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = np.tanh(Z)
    cache = Z

    return A, cache


def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION
       layer

    Arguments:
    A_prev -- activations from previous layer (or input data):
       (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current
       layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current
       layer, 1)
    activation -- the activation to be used in this layer, stored
       as a text string: "tanh"
```

```
    Returns:
    A -- the output of the activation function, also called the
       post-activation value
    cache -- a python dictionary containing "linear_cache" and
       "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "tanh":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = my_tanh(Z)
    else:
        print("activation string error!")
    #Verify dimensions:
    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache


def L_model_forward(X, parameters):
    """
    Implement forward propagation for the
       [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of
       examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there
                   are L-1 of them, indexed from 0 to L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2   # number of layers in the neural
       network
```

```
    # Implement [LINEAR -> TANH]*(L). Add "cache" to the "caches"
       list.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev,
           parameters["W"+ str(l)],parameters["b"+ str(l)],
           activation = "tanh")
        caches.append(cache)

    # Implement LINEAR -> TANH. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters["W"+
       str(L)],parameters["b"+ str(L)], activation =  "tanh")
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches


def compute_cost(AL, Y):
    """
    Implement the cost function as MSE

    Arguments:
    AL -- probability vector corresponding to your label
       predictions, shape (1, number of examples)
    Y -- true "label" vector (for example: containing 0 if NO, 1
       if YES), shape (1, number of examples)

    Returns:
    cost -- cross-entropy cost
    """

    # Compute loss from aL and y.
    cost = np.mean( (Y-AL)**2 ) #MSE cost

    cost = np.squeeze(cost)      # To make sure your cost's shape
       is what we expect (e.g. this turns [[17]] into 17).
    assert(cost.shape == ())

    return cost


def linear_backward(dZ, cache):
```

```
    """
    Implement the linear portion of backward propagation for a
       single layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output
       (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the
       forward propagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation
       (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer
       l), same shape as W
    db -- Gradient of the cost with respect to b (current layer
       l), same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1/m*np.dot(dZ,A_prev.T)
    db = 1/m*np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(W.T,dZ)
    #Verify dimensions:
    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db


def tanh_backward(dA, cache):
    """
    Implement the backward propagation for a single TANH unit.
    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation
       efficiently
    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
```

```
    s, _ = my_tanh(Z)
    dZ = dA * (1-s**2)
    #Verify dimensions:
    assert (dZ.shape == Z.shape)

    return dZ


def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION
      layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we
      store for computing backward propagation efficiently
    activation -- the activation to be used in this layer, stored
      as a text string: "tanh"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation
      (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer
      l), same shape as W
    db -- Gradient of the cost with respect to b (current layer
      l), same shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "tanh":
        dZ = tanh_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    else:
        print("error with backward activation function label!")

    return dA_prev, dW, db


def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->TANH] *
      (L) group
```

```
    Arguments:
    AL -- probability vector, output of the forward propagation
       (L_model_forward())
    Y -- true "label" vector (containing 0 if NO, 1 if YES)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with
                   "tanh"

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape
       as AL

    # Initializing the backpropagation
    ### START CODE HERE ### (1 line of code)
    #dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) #
       derivative of cost with respect to AL
    dAL = Y-AL
    ### END CODE HERE ###

    # Lth layer (TANH -> LINEAR) gradients. Inputs: "dAL,
       current_cache". Outputs: "grads["dAL-1"], grads["dWL"],
       grads["dbL"]
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" +
       str(L)] = linear_activation_backward(dAL, current_cache,
       activation="tanh")

    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        # lth layer: (TANH -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 1)], current_cache".
           Outputs: "grads["dA" + str(l)] , grads["dW" + str(l +
           1)] , grads["db" + str(l + 1)]
        current_cache = caches[l]
```

```
        dA_prev_temp, dW_temp, db_temp =
           linear_activation_backward(grads["dA" + str(l + 1)],
           current_cache, activation="tanh")
        grads["dA" + str(l)] =  dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads


def update_parameters(parameters, grads, learning_rate,
   momentum_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output
       of L_model_backward

    Returns:
    parameters -- python dictionary containing your updated
       parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural
       network

    # Update rule for each parameter. Use a for loop.
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
           learning_rate * grads["dW" + str(l + 1)] -
           momentum_rate * (learning_rate * grads["dW" + str(l)])
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
           learning_rate * grads["db" + str(l + 1)] -
           momentum_rate * (learning_rate * grads["db" + str(l)])
    return parameters


def predict(X, y, parameters):
    """
```

```
    This function is used to predict the results of a  L-layer
       neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    n = len(parameters) // 2 # number of layers in the neural
       network
    p = np.zeros((1,m))

    # Forward propagation
    probas, caches = L_model_forward(X, parameters)


    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0:
            p[0,i] = 1
        else:
            p[0,i] = 0

    #print results
    print ("predictions: " + str(p))
    print ("true labels: " + str(y))
    print("Accuracy: "  + str(np.sum((p == y)/m)))

    return p


def L_layer_model(X, Y, layers_dims, learning_rate = 0.1,
   momentum_rate = 0.1, num_iterations = 3000, print_cost=False,
   tol=0.01):#lr was 0.009
    """
    Implements a L-layer neural network:
       [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Arguments:
```

```
    X -- data, numpy array of shape (number of examples, entry
       size)
    Y -- true "label" vector (containing 0 if NO, 1 if YES), of
       shape (1, number of examples)
    layers_dims -- list containing the input size and each layer
       size, of length (number of layers + 1).
    learning_rate -- learning rate of the gradient descent update
       rule
    momentum_rate -- momentum rate of the gradient descent
       momentum update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every step
    tol -- accepted tolerance for error

    Returns:
    parameters -- parameters learnt by the model. They can then be
       used to predict.
    """

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization.
    parameters = initialize_parameters_deep(layers_dims)

    # Loop (gradient descent)
    epoch = 0 #number of epochs
    go_on = True #boolean to stop learning
    while epoch < num_iterations and go_on:

        # Forward propagation: [LINEAR -> TANH]*(L).
        AL, caches = L_model_forward(X, parameters)

        # Compute cost.
        cost = compute_cost(AL, Y)

        # Backward propagation.
        grads = L_model_backward(AL, Y, caches)

        # Update parameters.
        parameters = update_parameters(parameters, grads,
           learning_rate, momentum_rate)

        # Print the cost every training example
```

```
        if print_cost:
            print ("Cost after iteration %i: %f" %(epoch, cost))
            costs.append(cost)

        if cost < tol: #the algorithm converged
            go_on = False #end learning loop
        epoch += 1
    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.xlabel("Epoch")
    plt.ylabel("Mean Square Error")
    plt.title("Learning curve of NN")
    plt.show()

    return parameters


def plotW(layer, pos, cmap='YlOrRd'):
        """
        Plots evolution of weights and bias.

        Arguments:
        layer : int - indicates neuron layer
        pos : int - indicates neuron position
        cmap : string (default = 'YlOrRd') - Colormap for the
           graphics.
        """

        w_array = np.array(self.w_list)
        n_iter, d = w_array.shape
        x = np.arange(n_iter)

        colors = eval(f"plt.cm.{cmap}(np.linspace(0,1,n))")

        names = ["Bias"]
        names += list(map(lambda i: f"Neuron {i}", np.arange(1,d)))

        for i in range(d):
            plt.plot(x, w_array[:,i], c=colors[i], label=names[i])
        plt.title(f"Evolution of weights and bias of Neuron
           ({layer, pos})")
        plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
        plt.xlabel("Epoch")
        plt.show()
```

```python
#                     ------  Exercise 2 - XOR  ------


train_x = np.array([[0,0],
                    [0,1],
                    [1,0],
                    [1,1]]).T
train_y = np.array([[0,1,1,0]])

alpha = 0.1   # learning rate
mom = 0.1   # momentum rate
layers_dims = [2, 2, 1]   # 2 - layer model

parameters = L_layer_model(train_x, train_y, layers_dims,
   learning_rate = alpha, momentum_rate = mom, num_iterations =
   2500, print_cost = True)

test_data = np.random.random((30000,2))
pred_train = predict(train_x, train_y, parameters)

test_data0 = test_data[pred_train == 0]
test_data1 = test_data[pred_train == 1]

plt.figure(figsize=(6,6))
plt.scatter(test_data1[:,0], test_data1[:,1], s=3, c='red',
   label='1')
plt.scatter(test_data0[:,0], test_data0[:,1], s=3, c='blue',
   label='0')
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.title("Classification Regions of the NN")
plt.show()

df = pd.DataFrame(np.concatenate([test_data, predict(train_x,
   train_y, parameters)], axis=1), columns=['X1', 'X2', 'Output'])

plt.figure(figsize=(6,6))
sns.scatterplot(x='X1', y='X2', hue='Output', pallete = 'bright',
   data=df)
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.title("Regression Region of the NN")
plt.show()
```

```
# Plotting wheigts and bias evolutions for each neuron:

for i in range(len(layers_dims)):
    for j in range(layers_dims[i]):
        plotW(i, j, 'autumn')
```

Algoritmo 1: Código em Python que implementa o algoritmo de foward e back propagation para uma rede de L camadas, com função de ativação *tanh* e função de custo MSE.