Project:

This project was to implement gradient checking to ensure that the implementation of backpropagation is correct.

1-dimensional gradient checking

```
def forward_propagation(x, theta):
    """
    Implement the linear forward propagation (compute J)
(J(theta) = theta * x)

    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well

    Returns:
    J -- the value of function J, computed using the formula J(theta) = theta * x
    """

   J = theta*x
   return J
--
```

```
def gradient_check(x, theta, epsilon = 1e-7):
    """
    Implement the backward propagation
    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well
    epsilon -- tiny shift to the input to compute approximated gradient with
formula(1)


    Returns:
    difference -- difference (2) between the approximated gradient and the
backward propagation gradient
    """


    # Compute gradapprox
   thetaplus = theta+epsilon                          # Step 1
    thetaminus = theta-epsilon                          # Step 2
    J_plus = forward_propagation(x,thetaplus)                           # Step 3
    J_minus = forward_propagation(x,thetaminus)                          # Step 4
    gradapprox = (J_plus-J_minus)/(2*epsilon)                       # Step 5


    # Check if gradapprox is close enough to the output of backward_propagation()
```

```
    grad = backward_propagation(x,theta)

    numerator = np.linalg.norm(grad-gradapprox)                    # Step 1'
    denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
# Step 2'
    difference = numerator/denominator                             # Step 3'

    if difference < 1e-7:
        print ("The gradient is correct!")
    else:
        print ("The gradient is wrong!")

    return difference
```

N-dimensional gradient checking

```
def forward_propagation_n(X, Y, parameters):
    """
    Implements the forward propagation (and computes the cost)
    Arguments:
    X -- training set for m examples
    Y -- labels for m examples
```

```
    parameters -- python dictionary containing your parameters "W1", "b1", "W2",
"b2", "W3", "b3":
                W1 -- weight matrix of shape (5, 4)
                b1 -- bias vector of shape (5, 1)
                W2 -- weight matrix of shape (3, 5)
                b2 -- bias vector of shape (3, 1)
                W3 -- weight matrix of shape (1, 3)
                b3 -- bias vector of shape (1, 1)

    Returns:
    cost -- the cost function (logistic cost for one example)
    """

    # retrieve parameters
    m = X.shape[1]
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
```

```
    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = sigmoid(Z3)

    # Cost
    logprobs = np.multiply(-np.log(A3),Y) + np.multiply(-np.log(1 - A3), 1 - Y)
    cost = 1./m * np.sum(logprobs)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3)

    return cost, cache

--
def backward_propagation_n(X, Y, cache):
    """
    Implement the backward propagation
    Arguments:
    X -- input datapoint, of shape (input size, 1)
```

```
    Y -- true "label"
    cache -- cache output from forward_propagation_n()

    Returns:
    gradients -- A dictionary with the gradients of the cost with respect to each parameter, activation and pre-activation variables.
    """

    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

    dZ3 = A3 - Y
    dW3 = 1./m * np.dot(dZ3, A2.T)
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1./m * np.dot(dZ2, A1.T)
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
```

```
    dW1 = 1./m * np.dot(dZ1, X.T)
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
                 "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients
```

--

```
def gradient_check_n(parameters, gradients, X, Y, epsilon = 1e-7):
    """
    Checks if backward_propagation_n computes correctly the gradient of the cost output by forward_propagation_n

    Arguments:
    parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3":
    grad -- output of backward_propagation_n, contains gradients of the cost with respect to the parameters.
    x -- input datapoint, of shape (input size, 1)
    y -- true "label"
```

    epsilon -- tiny shift to the input to compute approximated gradient with formula(1)

    Returns:

    difference -- difference (2) between the approximated gradient and the backward propagation gradient

    """

    # Set-up variables

    parameters_values, _ = dictionary_to_vector(parameters)

    grad = gradients_to_vector(gradients)

    num_parameters = parameters_values.shape[0]

    J_plus = np.zeros((num_parameters, 1))

    J_minus = np.zeros((num_parameters, 1))

    gradapprox = np.zeros((num_parameters, 1))

    # Compute gradapprox

    for i in range(num_parameters):

        # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output = "J_plus[i]".

        # "_" is used because the function you have to outputs two parameters but we only care about the first one

```
        thetaplus = np.copy(parameters_values)                              # Step 1
        thetaplus[i][0] = thetaplus[i][0]+epsilon                         # Step 2
        J_plus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(thetaplus))
# Step 3

        # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output =
"J_minus[i]".
        thetaminus = np.copy(parameters_values)                             # Step 1
        thetaminus[i][0] = thetaplus[i][0]-epsilon                          # Step 2
        J_minus[i], _ = forward_propagation_n(X, Y,
vector_to_dictionary(thetaminus))                            # Step 3

        # Compute gradapprox[i]
        gradapprox[i] = (J_plus[i]-J_minus[i])/(2*epsilon)

    # Compare gradapprox to backward propagation gradients by computing
difference.
     numerator = np.linalg.norm(grad-gradapprox)                                # Step 1'
    denominator = np.linalg.norm(grad)+np.linalg.norm(gradapprox)
# Step 2'
    difference = numerator/denominator                                      # Step 3'
    if difference > 2e-7:
        print ("\033[93m" + "There is a mistake in the backward propagation!
difference = " + str(difference) + "\033[0m")
```

```
    else:
        print ("\033[92m" + "Your backward propagation works perfectly fine! difference = " + str(difference) + "\033[0m")

    return difference
```