# GRADED FUNCTION: forward\_propagation

def forward\_propagation(x, theta):

"""

Implement the linear forward propagation (compute J) presented in Figure 1 (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

"""

### START CODE HERE ### (approx. 1 line)

**J = np.dot(theta, x)**

### END CODE HERE ###

return J

# GRADED FUNCTION: backward\_propagation

def backward\_propagation(x, theta):

"""

Computes the derivative of J with respect to theta (see Figure 1).

Arguments:

x -- a real-valued input

theta -- our parameter, a real number as well

Returns:

dtheta -- the gradient of the cost with respect to theta

"""

### START CODE HERE ### (approx. 1 line)

**dtheta = x**

### END CODE HERE ###

return dtheta

# GRADED FUNCTION: gradient\_check

def gradient\_check(x, theta, epsilon=1e-7):

"""

Implement the backward propagation presented in Figure 1.

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 using left side of formula (1). epsilon is small enough, you don't need to worry about the limit.

### START CODE HERE ### (approx. 5 lines)

**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**

### END CODE HERE ###

# Check if gradapprox is close enough to the output of backward\_propagation()

### START CODE HERE ### (approx. 1 line)

**grad = backward\_propagation(x, theta)**

### END CODE HERE ###

### START CODE HERE ### (approx. 1 line)

**numerator = np.linalg.norm(grad – gradapprox) # Step 1'**

**denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox) # Step 2'**

**difference = numerator/denominator # Step 3'**

### END CODE HERE ###

if difference < 1e-7:

print("The gradient is correct!")

else:

print("The gradient is wrong!")

return difference

# GRADED FUNCTION: gradient\_check\_n

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

### START CODE HERE ### (approx. 3 lines)

**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**

### END CODE HERE ###

# Compute J\_minus[i]. Inputs: "parameters\_values, epsilon". Output = "J\_minus[i]".

### START CODE HERE ### (approx. 3 lines)

**thetaminus = np.copy(parameters\_values) # Step 1**

**thetaminus[i][0] = thetaminus[i][0] + epsilon # Step 2**

**J\_minus[i], \_ = forward\_propagation\_n(X, Y, vector\_to\_dictionary(**

**thetaminus)) # Step 3**

### END CODE HERE ###

# Compute gradapprox[i]

### START CODE HERE ### (approx. 1 line)

**gradapprox[i] = (J\_plus - J\_minus)/(2\*epsilon)**

### END CODE HERE ###

# Compare gradapprox to backward propagation gradients by computing difference.

### START CODE HERE ### (approx. 1 line)

**numerator = np.linalg.norm(grad – gradapprox) # Step 1'**

**denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox) # Step 2'**

**difference = numerator/denominator # Step 3'**

### END CODE HERE ###

if difference > 1e-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