Gradient Checking

Welcome to the final assignment for this week! In this assignment you'll be implementing gradient checking.

By the end of this notebook, you'll be able to:

Implement gradient checking to verify the accuracy of your backprop implementation

Table of Contents

1 - Packages

In [22]:

```
import numpy as np
from testCases import *
from public_tests import *
from gc_utils import sigmoid, relu, dictionary_to_vector, vector_to_dictionary,
gradients_to_vector

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2 - Problem Statement

You are part of a team working to make mobile payments available globally, and are asked to build a deep learning model to detect fraud--whenever someone makes a payment, you want to see if the payment might be fraudulent, such as if the user's account has been taken over by a hacker.

You already know that backpropagation is quite challenging to implement, and sometimes has bugs. Because this is a mission-critical application, your company's CEO wants to be really certain that your implementation of backpropagation is correct. Your CEO says, "Give me proof that your backpropagation is actually working!" To give this reassurance, you are going to use "gradient checking."

Let's do it!

3 - How does Gradient Checking work?

Backpropagation computes the gradients $\frac{\partial J}{\partial \theta}$, where θ denotes the parameters of the model. J is computed using forward propagation and your loss function.

Because forward propagation is relatively easy to implement, you're confident you got that right, and so you're almost 100% sure that you're computing the cost J correctly. Thus, you can use your code for computing J to verify the code for computing $\frac{\partial J}{\partial \theta}$.

Let's look back at the definition of a derivative (or gradient):

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2 \varepsilon}$$

If you're not familiar with the "$\lim_{\varepsilon \to 0}$" notation, it's just a way of saying "when ε is really, really small."

You know the following:

$\frac{\partial J}{\partial \theta}$ is what you want to make sure you're computing correctly. You can compute $J(\theta + \varepsilon)$ and $J(\theta - \varepsilon)$ (in the case that θ is a real number), since you're confident your implementation for J is correct. Let's use equation (1) and a small value for ε to convince your CEO that your code for computing $\frac{\partial J}{\partial \theta}$ is correct!

4 - 1-Dimensional Gradient Checking

Consider a 1D linear function $J(\theta) = \theta x$. The model contains only a single real-valued parameter θ, and takes x as input.

You will implement code to compute $J(.)$ and its derivative $\frac{\partial J}{\partial \theta}$. You will then use gradient checking to make sure your derivative computation for J is correct.

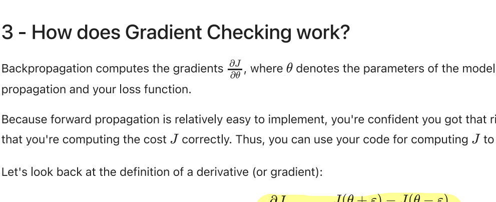

Figure 1:1D linear model

The diagram above shows the key computation steps: First start with x, then evaluate the function $J(x)$ ("forward propagation"). Then compute the derivative $\frac{\partial J}{\partial \theta}$ ("backward propagation").

Exercise 1 - forward_propagation

Implement `forward propagation`. For this simple function compute $J(.)$

In [23]:

```
# 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
    """

    # (approx. 1 line)
    # J =
    # YOUR CODE STARTS HERE
    J = np.dot(theta, x)
    # YOUR CODE ENDS HERE

    return J
```

In [24]:

```
x, theta = 2, 4
J = forward_propagation(x, theta)
print ("J = " + str(J))
forward_propagation_test(forward_propagation)
```

```
J = 8
All tests passed.
```

Exercise 2 - backward_propagation

Now, implement the `backward propagation` step (derivative computation) of Figure 1. That is, compute the derivative of $J(\theta) = \theta x$ with respect to θ. To save you from doing the calculus, you should get $dtheta = \frac{\partial J}{\partial \theta} = x$.

In [25]:

```
# 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
    """

    # (approx. 1 line)
    # dtheta =
    # YOUR CODE STARTS HERE
    dtheta = x
    # YOUR CODE ENDS HERE

    return dtheta
```

In [26]:

```
x, theta = 2, 4
dtheta = backward_propagation(x, theta)
print ("dtheta = " + str(dtheta))
backward_propagation_test(backward_propagation)
```

```
dtheta = 2
All tests passed.
```

Exercise 3 - gradient_check

To show that the `backward_propagation()` function is correctly computing the gradient $\frac{\partial J}{\partial \theta}$, let's implement gradient checking.

Instructions:

- First compute "gradapprox" using the formula above (1) and a small value of ε. Here are the Steps to follow:
  1. $\theta^{+} = \theta + \varepsilon$
  2. $\theta^{-} = \theta - \varepsilon$
  3. $J^{+} = J(\theta^{+})$
  4. $J^{-} = J(\theta^{-})$
  5. $gradapprox = \frac{J^{+} - J^{-}}{2 \varepsilon}$
- Then compute the gradient using backward propagation, and store the result in a variable "grad"
- Finally, compute the relative difference between "gradapprox" and the "grad" using the following formula:

$$difference = \frac {\mid\mid grad - gradapprox \mid\mid_2}{\mid\mid grad \mid\mid_2 + \mid\mid gradapprox \mid\mid_2} \tag{2}$$

  You will need 3 Steps to compute this formula:
  - 1'. compute the numerator using np.linalg.norm(...)
  - 2'. compute the denominator. You will need to call np.linalg.norm(...) twice.
  - 3'. divide them.
- If this difference is small (say less than 10^{-7}), you can be quite confident that you have computed your gradient correctly. Otherwise, there may be a mistake in the gradient computation.

In [27]:

```
# GRADED FUNCTION: gradient_check

def gradient_check(x, theta, epsilon=1e-7, print_msg=False):
    """
    Implement the backward propagation presented in Figure 1.

    Arguments:
    x -- a float input
    theta -- our parameter, a float 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. Float output
    """

    # Compute gradapprox using left side of formula (1). epsilon is small enough,
you don't need to worry about the limit.
    # (approx. 5 lines)
    # theta_plus =                                 # Step 1
    # theta_minus =                                # Step 2
    # J_plus =                                    # Step 3
    # J_minus =                                   # Step 4
    # gradapprox =                                # Step 5
    # YOUR CODE STARTS HERE
    theta_plus = theta + epsilon
    theta_minus = theta - epsilon
    J_plus = forward_propagation(x, theta_plus)
    J_minus = forward_propagation(x, theta_minus)
    gradapprox = (J_plus - J_minus) / (2*epsilon)
    # YOUR CODE ENDS HERE

    # Check if gradapprox is close enough to the output of backward_propagation()
    #(approx. 1 line) DO NOT USE "grad = gradapprox"
    # grad =
    # YOUR CODE STARTS HERE
    grad = backward_propagation(x, theta)
    # YOUR CODE ENDS HERE

    #(approx. 1 line)
    # numerator =                                 # Step 1'
    # denominator =                               # Step 2'
    # difference =                                # Step 3'
    # YOUR CODE STARTS HERE
    numerator = np.linalg.norm(grad-gradapprox)
    denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
    difference = numerator / denominator
    # YOUR CODE ENDS HERE
    if print_msg:
        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
```

In [28]:

```
x, theta = 2, 4
difference = gradient_check(2,4, print_msg=True)

#gradient_check_test(gradient_check)
```

```
Your backward propagation works perfectly fine! difference = 2.919335883291695e-10
```

Congrats, the difference is smaller than the 10^{-7} threshold. So you can have high confidence that you've correctly computed the gradient in `backward_propagation()`.

Now, in the more general case, your cost function J has more than a single 1D input. When you are training a neural network, θ actually consists of multiple matrices $W^{[l]}$ and biases $b^{[l]}$! It is important to know how to do a gradient check with higher-dimensional inputs. Let's do it!

5 - N-Dimensional Gradient Checking

The following figure describes the forward and backward propagation of your fraud detection model.

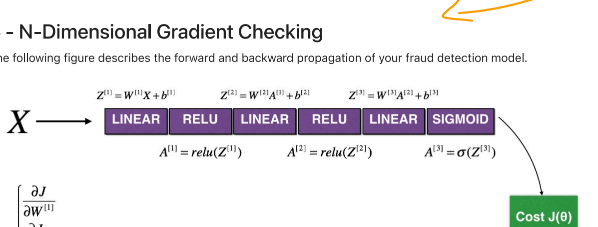

Figure 2: Deep neural network. LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

Let's look at your implementations for forward propagation and backward propagation.

In [29]:

```
def forward_propagation_n(X, Y, parameters):
    """
    Implements the forward propagation (and computes the cost) presented in
Figure 3.

    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)
    cache -- a tuple with the intermediate values (Z1, A1, W1, b1, Z2, A2, W2,
b2, Z3, A3, W3, b3)

    """

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

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

    return cost, cache
```

Now, run backward propagation.

In [30]:

```
def backward_propagation_n(X, Y, cache):
    """
    Implement the backward propagation presented in figure 2.

    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) * 2
    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 = 4. / 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
```

You obtained some results on the fraud detection test set but you are not 100% sure of your model. Nobody's perfect! Let's implement gradient checking to verify if your gradients are correct.

How does gradient checking work?.

As in Section 3 and 4, you want to compare "gradapprox" to the gradient computed by backpropagation. The formula is still:

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2 \varepsilon} \tag{1}$$

However, θ is not a scalar anymore. It is a dictionary called "parameters". The function "`dictionary_to_vector()`" has been implemented for you. It converts the "parameters" dictionary into a vector called "values", obtained by reshaping all parameters (W1, b1, W2, b2, W3, b3) into vectors and concatenating them.

The inverse function is "`vector_to_dictionary`" which outputs back the "parameters" dictionary.

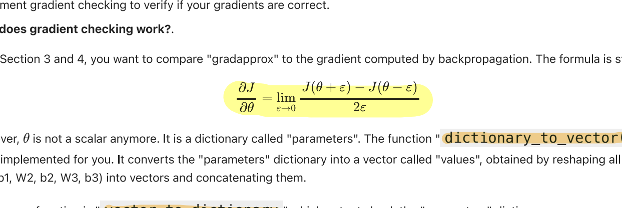

Figure 2: dictionary_to_vector() and vector_to_dictionary(). You will need these functions in gradient_check_n()

The "gradients" dictionary has also been converted into a vector "grad" using gradients_to_vector(), so you don't need to worry about that.

Now, for every single parameter in your vector, you will apply the same procedure as for the gradient_check exercise. You will store each gradient approximation in a vector `gradapprox`. If the check goes as expected, each value in this approximation must match the real gradient values stored in the `grad` vector.

Note that `grad` is calculated using the function `gradients_to_vector`, which uses the gradients outputs of the `backward_propagation_n` function.

Exercise 4 - gradient_check_n

Implement the function below.

Instructions: Here is pseudo-code that will help you implement the gradient check.

For each i in num_parameters:

- To compute `J_plus[i]`:
  1. Set θ^{+} to `np.copy(parameters_values)`
  2. Set θ^{+}_i to $\theta^{+}_i + \varepsilon$
  3. Calculate J^{+}_i using to `forward_propagation_n(x, y, vector_to_dictionary(` θ^{+} `))`.
- To compute `J_minus[i]`: do the same thing with θ^{-}
- Compute $gradapprox[i] = \frac{J^{+}_i - J^{-}_i}{2 \varepsilon}$

Thus, you get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `parameter_values[i]`. You can now compare this gradapprox vector to the gradients vector from backpropagation. Just like for the 1D case (Steps 1', 2', 3'), compute:

$$difference = \frac {\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2 } \tag{3}$$

Note: Use `np.linalg.norm` to get the norms

In [35]:

```
# GRADED FUNCTION: gradient_check_n

def gradient_check_n(parameters, gradients, X, Y, epsilon=1e-7, print_msg=False):
    """
    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
        #(approx. 3 lines)
        # theta_plus =                                        # Step 1
        # theta_plus[i] =                                     # Step 2
        # J_plus[i], _ =                                     # Step 3
        # YOUR CODE STARTS HERE
        theta_plus = np.copy(parameters_values)
        theta_plus[i][0] = theta_plus[i][0] + epsilon
        J_plus[i], _ = forward_propagation_n(X, Y,
vector_to_dictionary(theta_plus))
        # YOUR CODE ENDS HERE

        # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output =
"J_minus[i]".
        #(approx. 3 lines)
        # theta_minus =                                      # Step 1
        # theta_minus[i] =                                   # Step 2
        # J_minus[i], _ =                                   # Step 3
        # YOUR CODE STARTS HERE
        theta_minus = np.copy(parameters_values)
        theta_minus[i][0] = theta_minus[i][0] - epsilon
        J_minus[i], _ = forward_propagation_n(X, Y,
vector_to_dictionary(theta_minus))
        # YOUR CODE ENDS HERE

        # Compute gradapprox[i]
        # (approx. 1 line)
        # gradapprox[i] =
        # YOUR CODE STARTS HERE
        gradapprox[i] = (J_plus[i] - J_minus[i]) / (2*epsilon)
        # YOUR CODE ENDS HERE

    # Compare gradapprox to backward propagation gradients by computing
difference.
    # (approx. 1 line)
    # numerator =                                            # Step 1'
    # denominator =                                          # Step 2'
    # difference =                                           # Step 3'
    # YOUR CODE STARTS HERE
    numerator = np.linalg.norm(grad-gradapprox)
    denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
    difference = numerator / denominator
    # YOUR CODE ENDS HERE
    if print_msg:
        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
```

In [36]:

```
X, Y, parameters = gradient_check_n_test_case()

cost, cache = forward_propagation_n(X, Y, parameters)
gradients = backward_propagation_n(X, Y, cache)
difference = gradient_check_n(parameters, gradients, X, Y, 1e-7, True)
expected_values = [0.2850931567761623, 1.1890913024229996e-07]
assert not(type(difference) == np.ndarray), "You are not using np.linalg.norm for
numerator or denominator"
assert np.any(np.isclose(difference, expected_values)), "Wrong value. It is not
one of the expected values"
```

```
There is a mistake in the backward propagation! difference = 0.2850931567761623
```

Expected output:

There is a mistake in the backward propagation! difference = 0.2850931567761623

It seems that there were errors in the `backward_propagation_n` code! Good thing you've implemented the gradient check. Go back to `backward_propagation` and try to find/correct the errors *(Hint: check dW2 and db1)*. Rerun the gradient check when you think you've fixed it. Remember, you'll need to re-execute the cell defining `backward_propagation_n()` if you modify the code.

Can you get gradient check to declare your derivative computation correct? Even though this part of the assignment isn't graded, you should try to find the bug and re-run gradient check until you're convinced backprop is now correctly implemented.

Notes

- Gradient Checking is slow! Approximating the gradient with $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ is computationally costly. For this reason, we don't run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.
- Gradient Checking, at least as we've presented it, doesn't work with dropout. You would usually run the gradient check algorithm without dropout to make sure your backprop is correct, then add dropout.

Congrats! Now you can be confident that your deep learning model for fraud detection is working correctly! You can even use this to convince your CEO. :)

What you should remember from this notebook:

- Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
- Gradient checking is slow, so you don't want to run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.