10. Gradient Checking

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

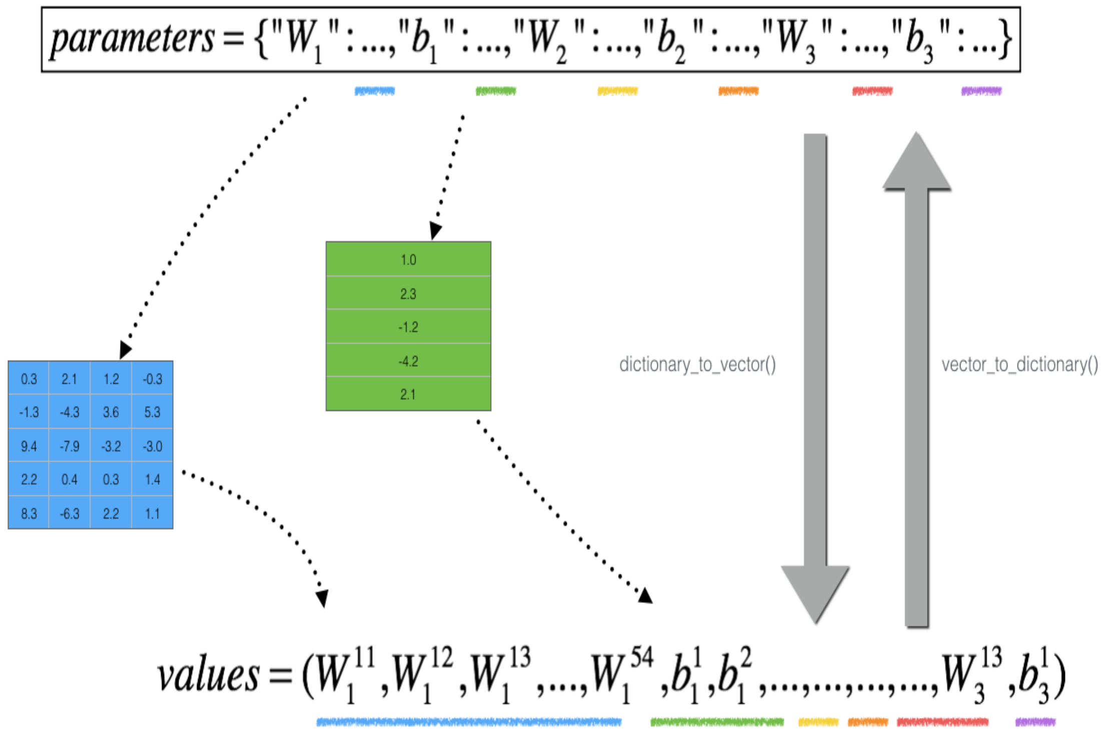
Because forward propagation is relatively easy to implement, we assume we're computing the cost J correctly. Thus, we can use our code for computing J to verify the code for computing $\frac{\partial J}{\partial \theta}$.

The definition of a derivative (or gradient):

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2 \varepsilon} \tag{26}$$

How does gradient checking work?.

We convert the "parameters" dictionary into a vector, obtained by reshaping all parameters (W1, b1, W2, b2, W3, b3, ...) into vectors and concatenating them.


parameters = {"W1":...,"b1":...,"W2":...,"b2":...,"W3":...,"b3":...}
0.3 2.1 1.2 -0.3
-1.3 -4.3 3.6 5.3
9.4 -7.9 -3.2 -3.0
2.2 0.4 0.3 1.4
8.3 -6.3 2.2 1.1
1.0
2.3
-1.2
-4.2
2.1
dictionary_to_vector()
vector_to_dictionary()
values = (W1^11, W1^12, W1^13, ..., W1^54, b1^1, b1^2, ..., ..., ..., ..., W3^13, b3^1)

For each parameter:

- First compute "gradapprox" using the formula (26) above 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"

Thus, we get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `params_values[i]`. Finally, compute the relative difference between "gradapprox" and the "grad" using the following formula:

$$difference = \frac{\|grad - gradapprox\|_2}{\|grad\|_2 + \|gradapprox\|_2} \tag{27}$$

Note

- 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. We would usually run the gradient check algorithm without dropout to make sure our backprop is correct, then add dropout.

In [26]:

```python
# convert params to a vector

def params_to_vector(params):
    """
    Arguments:
    params: python dictionary containing weight matrix wl and bias vector bl for the lth layer,
            params['W1'], params['W2'], ..., params['WL'], ..., and params['b1'], params['b2'], ..., params['bl'], ...

    Returns:
    theta: one-column vector by flattening and concatenating params['W1'], params['b1'], params['W2'], ...,
            and params['b2'], ..., params['WL'], params['bl'], ...
    keys: list of keys for each row of theta, ['W1', 'W1'...,'b1','b1',...]
    """
    keys = []
    L = len(params) // 2 + 1
    first_flag = True
    for l in range(1, L):
        cur_vector = np.reshape(params['W' + str(l)], (-1, 1))
        keys = keys + ['W' + str(l)] * cur_vector.shape[0]
        if first_flag:
            theta = cur_vector
            first_flag = False
        else:
            theta = np.concatenate((theta, cur_vector), axis = 0)

        cur_vector = np.reshape(params['b' + str(l)], (-1, 1))
        keys = keys + ['b' + str(l)] * cur_vector.shape[0]
        theta = np.concatenate((theta, cur_vector), axis = 0)

    return theta, keys
```

In [27]:
```
# convert grads to a vector

def grads_to_vector(grads):
    """
    Arguments:
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWL'], ..., and grads['db1'], grads['db2'], ..., grads['dbL'], ...

    Returns:
    theta: one-column vector by flattening and concatenating grads['dW1'], grads['db1'], grads['dW2'], ...,
            and grads['db2'], ..., grads['dWL'], grads['dbL'], ...
    keys: list of keys for each row of theta, ['dW1', 'dW1'...,'db1','db1',...]
    """
    keys = []
    L = len(grads) // 2 + 1
    first_flag = True
    for l in range(1, L):
        cur_vector = np.reshape(grads['dW' + str(l)], (-1, 1))
        keys = keys + ['dW' + str(l)] * cur_vector.shape[0]
        if first_flag:
            theta = cur_vector
            first_flag = False
        else:
            theta = np.concatenate((theta, cur_vector), axis = 0)

        cur_vector = np.reshape(grads['db' + str(l)], (-1, 1))
        keys = keys + ['db' + str(l)] * cur_vector.shape[0]
        theta = np.concatenate((theta, cur_vector), axis = 0)

    return theta, keys
```

In [28]:
```python
# convert vector back to params

def vector_to_params(theta, params):
    """
    Arguments:
    theta: one-column vector by flattening and concatenating grads['dW1'], grads['db1'], grads['dW2'], ...,
           and grads['db2'], ..., grads['dWL'], grads['dbL'], ...
    params: original python dictionary containing weight matrix wL and bias vector bL for the Lth layer,
            params['W1'], params['W2'], ..., params['WL'], ..., and params['b1'], params['b2'], ..., params['bL'], ...
            it has the information of the exact dimension of each matrix

    Returns:
    params: converted weight matrices and bias vectors from theta
    """
    L = len(params) // 2 + 1
    index = 0;

    for l in range(1, L):
        params['W' + str(l)] = theta[index : index + params['W' + str(l)].size, 0].reshape(params['W' + str(l)].shape)
        index += params['W' + str(l)].size
        params['b' + str(l)] = theta[index : index + params['b' + str(l)].size, 0].reshape(params['b' + str(l)].shape)
        index += params['b' + str(l)].size

    return params
```

In [29]:
```
# gradient checking

def gradient_checking(params, grads, X, Y, epsilon = 1e-7):
    """
    Arguments:
    params: python dictionary containing weight matrix wl and bias vector bl for the lth layer,
            params['W1'], params['W2'], ..., params['WL'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
    grads: python dictionary containing gradients of the cost with respect to the weight matrices and bias vectors in each lay
er,
            grads['dW1'], grads['dW2'], ..., grads['dWL'], ..., and grads['db1'], grads['db2'], ..., grads['dbl'], ...,
            to be compared to "gradapprox"
    X: data set features, with the dimension of (number of features, number of examples)
    Y: data set labels, with the dimension of (1, number of examples)

    Returns:
    difference: difference between the approximated gradient and the backward propagation gradient, defined above
    """
    params_values, _ = params_to_vector(params)
    grads_values, _ = grads_to_vector(grads)
    num_params = params_values.shape[0]
    J_plus = np.zeros(params_values.shape)
    J_minus = np.zeros(params_values.shape)

    for i in range(num_params):
        # get J_plus
        params_values_copy = np.copy(params_values)
        params_values_copy[i, 0] += epsilon
        Aout, _ = L_layer_forward(X, vector_to_params(params_values_copy, params.copy()))
        J_plus[i, 0] = cost_func(Aout, Y)
        # get J_minus
        params_values_copy = np.copy(params_values)
        params_values_copy[i, 0] -= epsilon
        Aout, _ = L_layer_forward(X, vector_to_params(params_values_copy, params.copy()))
        J_minus[i, 0] = cost_func(Aout, Y)

    # get approximated gradients
    gradapprox = np.subtract(J_plus, J_minus) / (2 * epsilon)

    # calculate difference
    difference = np.linalg.norm(grads_values - gradapprox) / (np.linalg.norm(grads_values) + np.linalg.norm(gradapprox))

    if difference > 2e-7:
        # font color:
        # Red = '\033[91m', Green = '\033[92m', Blue = '\033[94m', Cyan = '\033[96m', White = '\033[97m',
```

```python
        # Yellow = '\033[93m', Magenta = '\033[95m', Grey = '\033[90m', Black = '\033[90m', Default = '\033[99m'
        # end: '\033[0m'
        print('\033[91m' + 'There is a mistake, the difference is ' + str(difference) + '\033[0m')
    else:
        print('\033[92m' + 'Good, the difference is ' + str(difference) + '\033[0m')

    return difference
```

In [30]:

```python
# create random values of X, Y and params as test case

def gradient_checking_test():
    np.random.seed(10)
    X = np.random.randn(5, 4)
    Y = np.array([[1, 1, 0, 0]])
    W1 = np.random.randn(6, 5)
    b1 = np.random.randn(6, 1)
    W2 = np.random.randn(3, 6)
    b2 = np.random.randn(3, 1)
    W3 = np.random.randn(1, 3)
    b3 = np.random.randn(1, 1)
    params = {'W1':W1, 'b1':b1, 'W2':W2, 'b2':b2, 'W3':W3, 'b3':b3}

    return X, Y, params
```

In [31]:

```python
# run gradient checking
def run_gradient_checking():
    X, Y, params = gradient_checking_test()
    Aout, caches = L_layer_forward(X, params)
    grads = L_layer_backward(Aout, Y, caches)
    gradient_checking(params, grads, X, Y)
```

In [32]:

```python
run_gradient_checking()
```

```
Good, the difference is 1.959878475377247e-09
```