§ Week 1 / Assignment — Gradient Checking

1. Gradient Descent checking

1) 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."

We know the following:

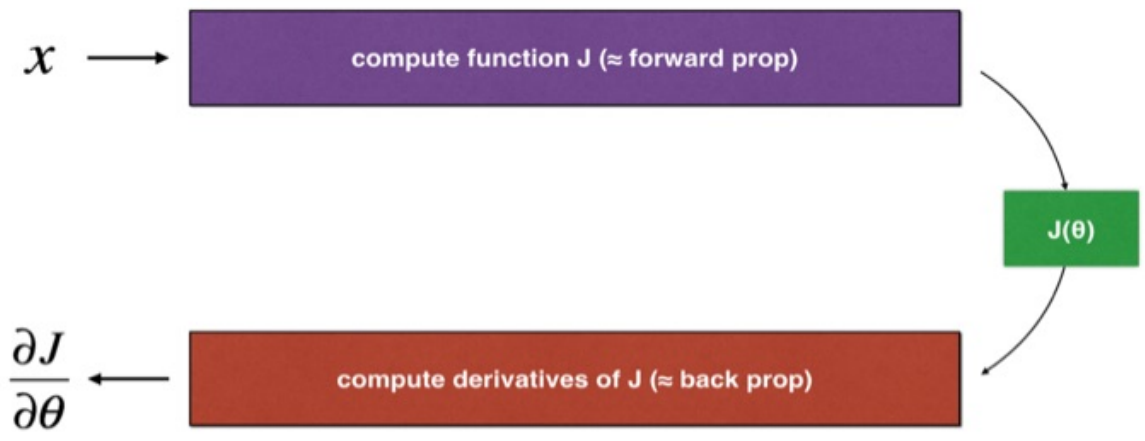
- $\frac{\partial J}{\partial \theta}$ is what you want to make sure you're computing correctly.
- You can compute $J(\theta + \varepsilon)$ and $J(\theta - \varepsilon)$ (in the case that θ is a real number), since you're confident your implementation for J is correct.

Lets 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!

2. 1-D 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.



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").

- 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:

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

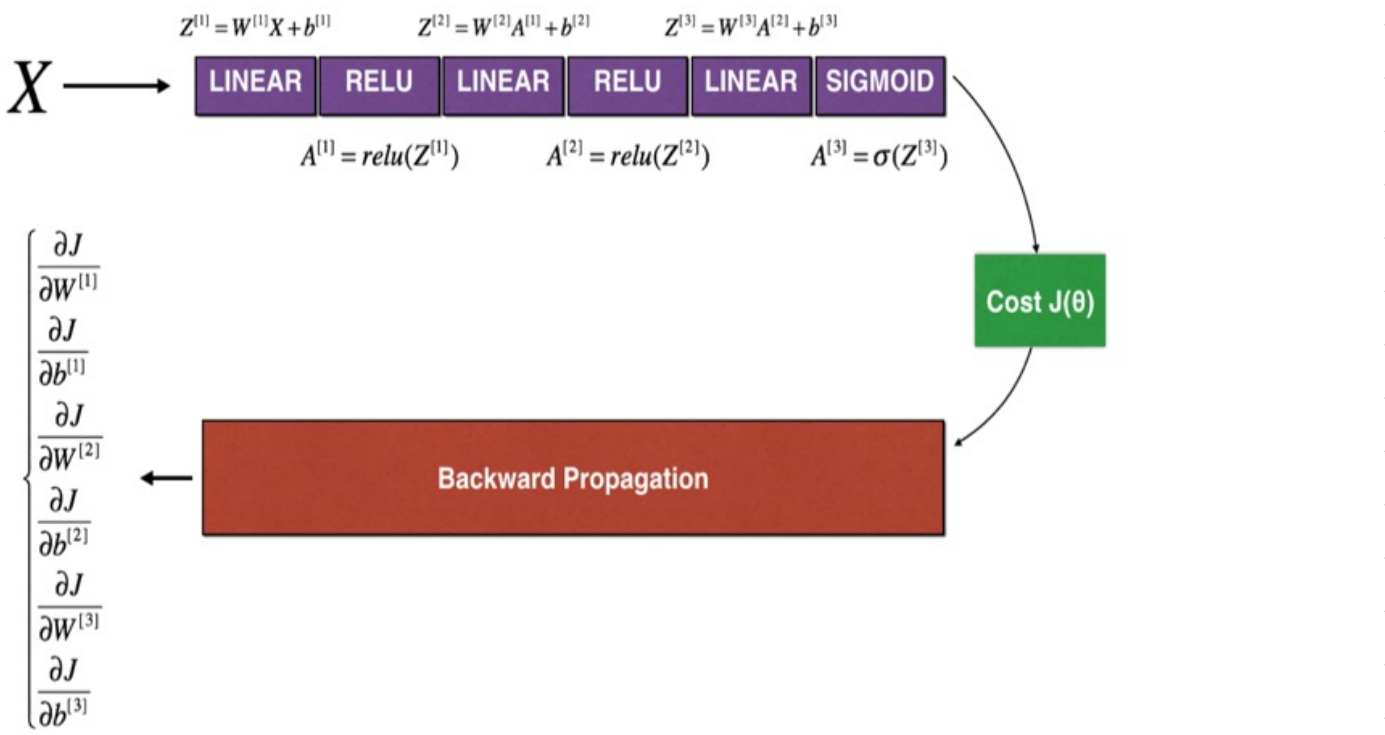
  You will need 3 Steps to compute this formula:
  - 1'. compute the numerator using np.linalg.norm(...)
  - 2'. compute the denominator. You will need to call np.linalg.norm(...) twice.
  - 3'. divide them.
- If this difference is small (say less than 10^{-7}), you can be quite confident that you have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

```
    ### 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 ###
```

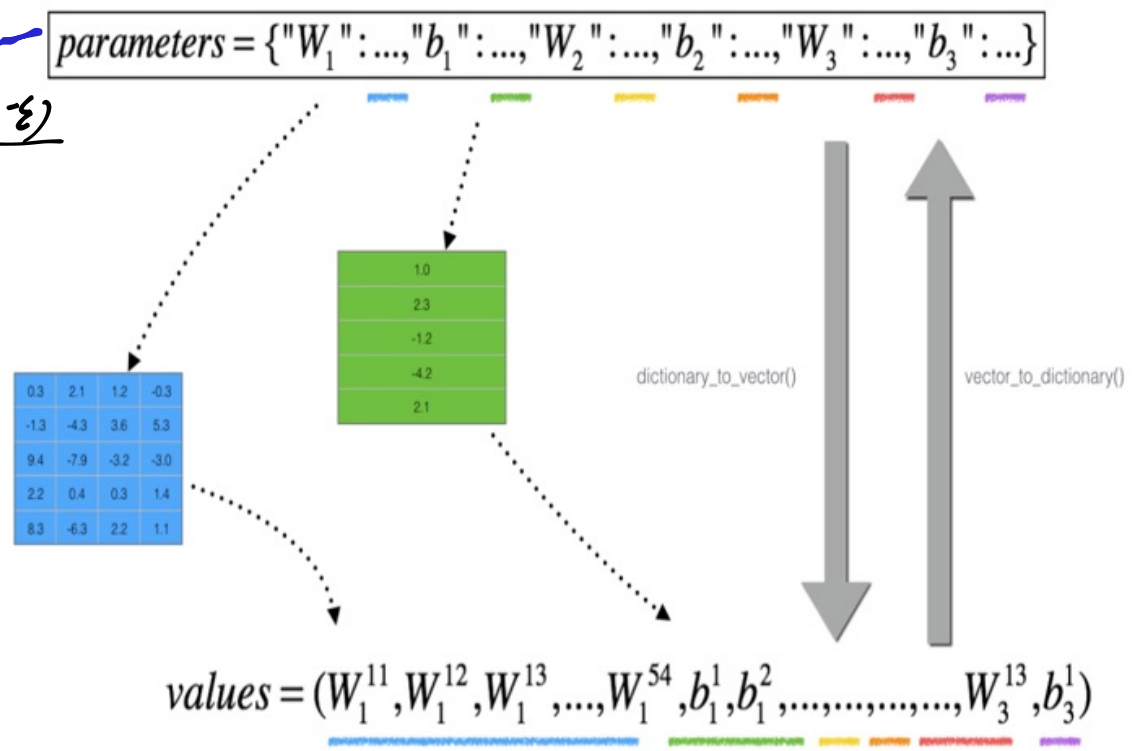
1~5

1'~3'

3. N-D Gradient Checking



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



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:

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

```
    # 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 c
ut the first one
        ### START CODE HERE ### (approx. 3 lines)
        thetaplus = np.copy(parameters_values)                                      # Step
        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
        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[i] - J_minus[i]) / (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
```

- 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 we don't 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.