Improving Deep Neural Networks – week 1)

Initialization

Experimenting different initialization methods for a 3-layer neural network (already implemented).

The initialization methods experimented:

- Zeros initialization - setting *initialization = "zeros"* in the input argument.
- Random initialization - setting *initialization = "random"* in the input argument. This initializes the weights to scaled random values and the biases to zeros.
- He initialization - setting *initialization = "he"* in the input argument. This initializes the weights to random values scaled according to a paper by He et al., 2015.

There are two types of parameters to initialize in a neural network:

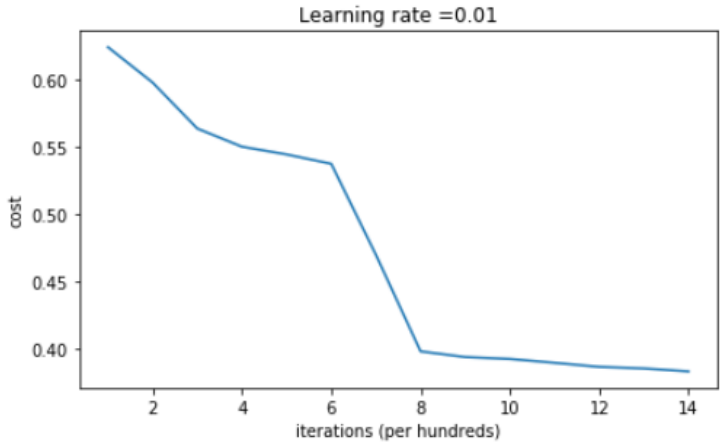
1) The weight matrices $W^{[1]}, W^{[2]}, \dots, W^{[L]}$
2) The bias vectors $b^{[1]}, b^{[2]}, \dots, b^{[L]}$

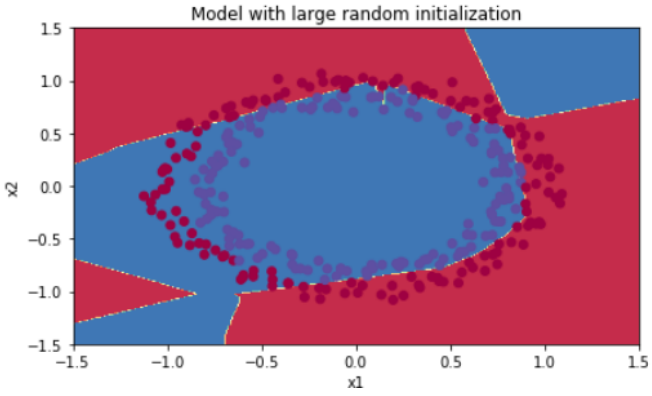
Zeros initialization

In general, initializing all the weights to zero results in the network failing to break symmetry.

The weights $W^{[l]}$ should be initialized randomly to break symmetry. It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[1]}$ is initialized randomly.

Random initialization (scaled by · 10)





On the training set: Accuracy = 0.83

On test set: Accuracy = 0.86

The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 for some examples, and when it gets that example wrong it incurs a very high loss for that example.

Poor initialization can lead to vanishing / exploding gradients, which also slows down the optimization algorithm. If you train this network longer you will see better results but initializing with overly large random numbers slows down the optimization.

Initializing weights to very large random values does not work well.

He initialization

Similar to "Xavier initialization": Xavier initialization uses a scaling factor for the weights $W^{[l]}$ of:

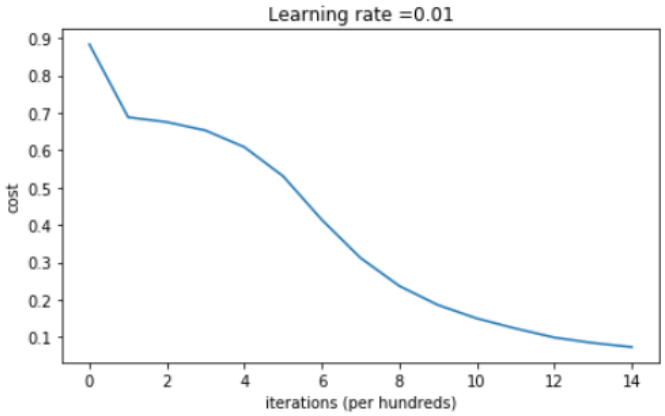
$$sqrt(1./layers\_dims[l-1])$$

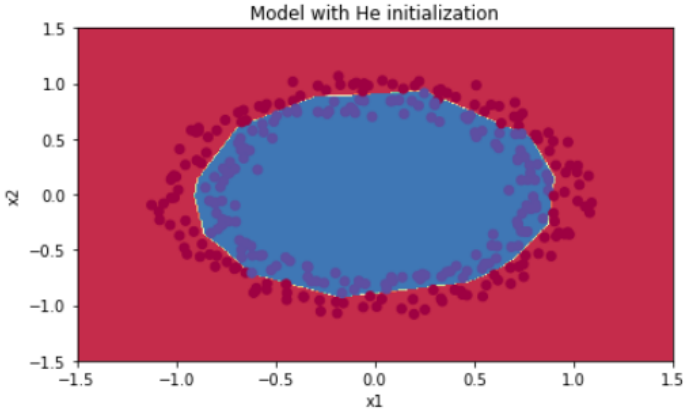
He initialization uses a scaling factor for the weights $W^{[l]}$ of:

$$sqrt(1./layers\_dims[l-1])$$

! ***layers_dims*** **– list containing the size of each layer.**

This initialization is similar to the random initialization. The only difference is that instead of multiplying the random $W^{[l]}$ values by 10 (scaling by · 10), we multiply by $\sqrt{\frac{2}{layers\_dims[l-1]}}$, which is what He initialization recommends for layers with a ReLU activation function.





On the training set: Accuracy = 0.9933333333

On test set: Accuracy = 0.96

The model with He initialization separates the blue and the red dots very well in a small number of iterations.

Conclusions

Model	Train accuracy	Problem/Comment
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

Regularization

Experimenting different regularization methods for a 3-layer neural network (already implemented).

This model can be used:

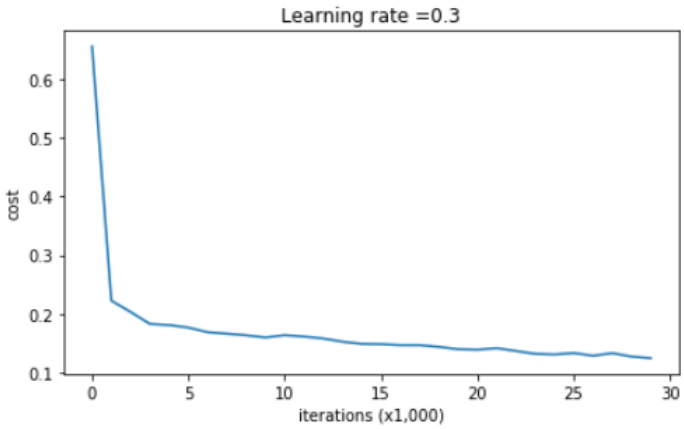
- regularization mode -- by setting the *lambd* input to a non-zero value (We use "lambd" instead of "lambda" because "lambda" is a reserved keyword in Python).
- in dropout mode -- by setting the *keep_prob* to a value less than one.

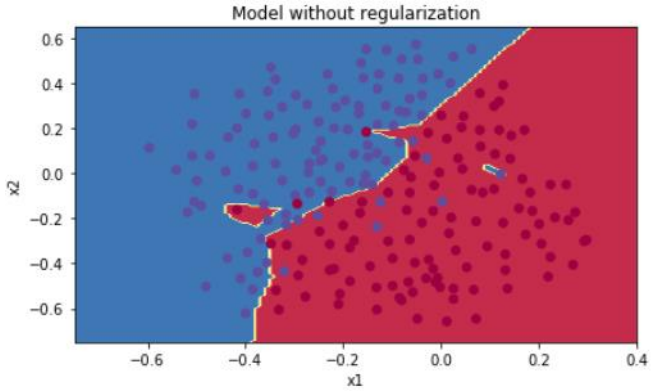
The modes experimented:

- Non regularized model
- L2 regularization – functions: "compute_cost_with_regularization" and "backward_propagation_with_regularization".
- Dropout – functions:" "forward_propagation_with_dropout()" and "backward_propagation_with_dropout()"

Non regularized model

This is the baseline model.





On the training set: Accuracy = 0.947867298578

On test set: Accuracy = 0.915

The non-regularized model is overfitting the training set.

L2 regularization model

L2 regularization is a standard way to avoid overfitting. It consists of appropriately modifying the cost function:

From – *compute_cost*:

$$J = \underbrace{\frac{-1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + \left(1-y^{(i)}\right)\log\left(1-a^{[L](i)}\right)\right)}_{cross-entropy\ cost}$$

To – *compute_cost_with_regularization*:

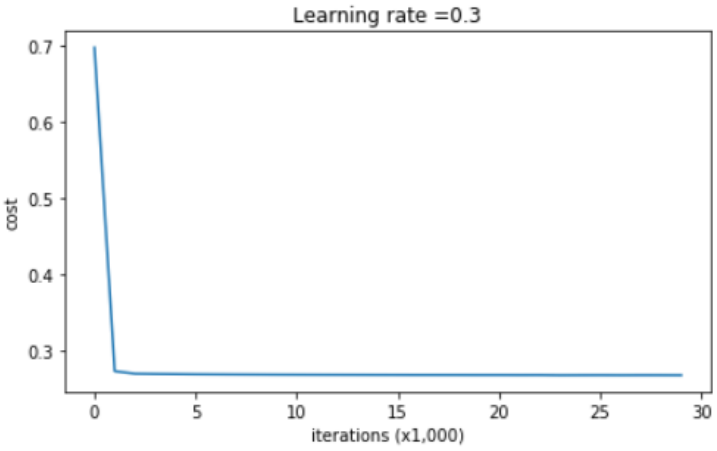
$$J_{Regularized} = \frac{-1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + \left(1-y^{(i)}\right)\log\left(1-a^{[L](i)}\right)\right) + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{kj}^{[l]2}}_{L2\ regularization\ cost}$$

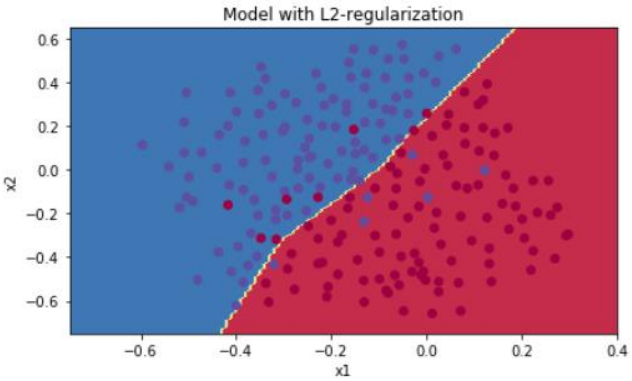
Since the cost changed, we changed the backward propagation as well. All the gradients must be computed with respect to the new cost.

The changes are implemented in *backward_propagation_with_regularization* function. The changes only concern dW1, dW2 and dW3. For each, the regularization term's gradient is added:

$$\frac{d}{dW}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right) = \frac{\lambda}{m}W$$

Running the model with $\lambda = 0.7$:





On the training set: Accuracy = 0.938388625592

On test set: Accuracy = 0.93

L2 regularization makes the decision boundary smoother.

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights. This leads to a smoother model in which the output changes more slowly as the input changes.

Dropout model

Dropout model randomly shuts down some neurons in each iteration. At each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.

Forward propagation with dropout

forward_propagation_with_dropout function adds dropout to the first and second hidden layers (out of 3 layer neural network), using 4 steps:

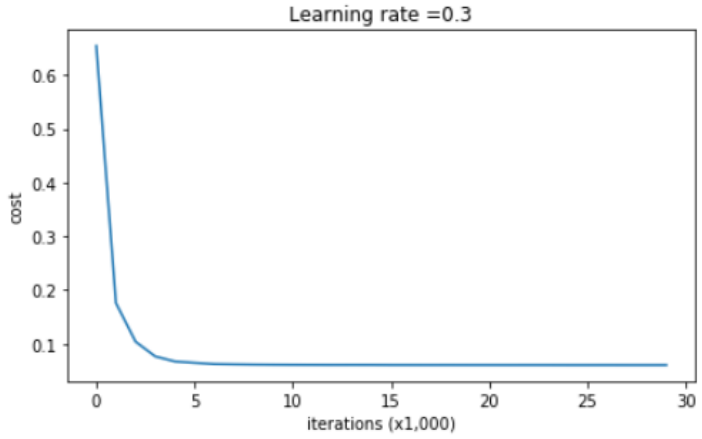
1) Create a random matrix $D^{[l]} = \left[d^{[l](1)}, d^{[l](2)}, \ldots, d^{[l](m)}\right]$ of the same dimensions as $A^{[l]}$
2) Set each entry of $D^{[l]}$ with probability above *keep_prob* to 1, otherwise 0.
3) Set $A^{[l]}$ to $A^{[l]} * D^{[l]}$
4) Divide $A^{[l]}$ by *keep_prob* to assure that the result of the cost will still have the same expected value as without dropout ("inverted dropout").

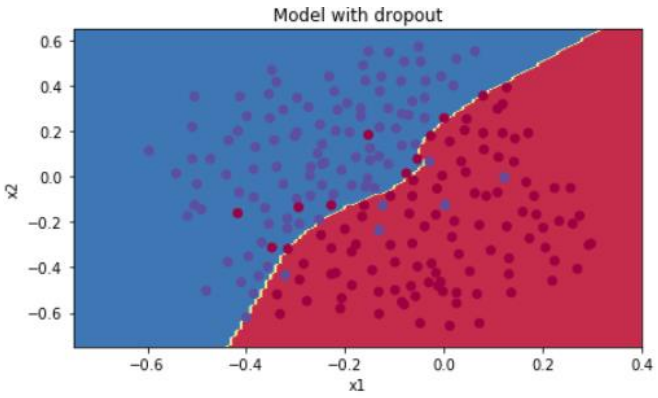
Backward propagation with dropout

backward_propagation_with_dropout function adds dropout to the first and second hidden layers (out of 3 layer neural network), using the masks $D^{[1]}$ and $D^{[2]}$ stored in the cache, in 2 steps:

1) Shut down the same neurons, by reapplying the same mask $D^{[l]}$ to $dA^{[l]}$ $\left(dA^{[l]} * D^{[l]}\right)$.
2) Divide $dA^{[l]}$ by *keep_prob* (the calculus interpretation is that if $A^{[l]}$ is scaled by *keep_prob*, then its derivative $dA^{[l]}$ is also scaled by the same *keep_prob*).

Runing the model with $keep\ prob\ =\ 0.86$:





On the training set: Accuracy = 0.928909952607

On test set: Accuracy = 0.95

The model is not overfitting.

Note:

A **common mistake** when using dropout is to use it both in training and testing. You should use dropout (randomly eliminate nodes) only in training.

Apply dropout both during forward and backward propagation.

Conclusions

model	train accuracy	test accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

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.

If you are confident your implementation for J is correct, you can use your code for computing J to **verify the code for computing $\frac{\partial J}{\partial \theta}$.**

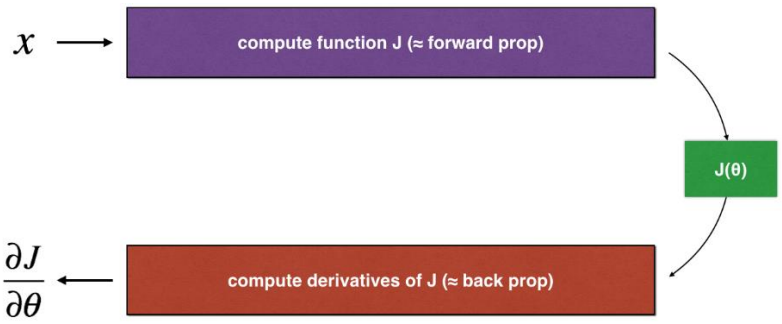
The definition of a derivative (or gradient):

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

1-dimensional gradient checking

1D linear function – The model contains only a single real-valued parameter θ, and takes x as input:

$$J(\theta) = \theta x$$



The derivative of J:

$$\frac{\partial J}{\partial \theta} = x$$

Gradient check implementation

1) Compute "*gradapprox*" using the steps:
   a) $\theta^+ = \theta + \varepsilon$
   b) $\theta^- = \theta - \varepsilon$
   c) $J^+ = J(\theta^+)$
   d) $J^- = J(\theta^-)$
   e) $gradapprox = \frac{J^+ - J^-}{2\varepsilon}$
2) Compute the gradient using backward propagation and store the result in a variable "*grad*".
3) Compute the relative difference between "*gradapprox*" and "*grad*" using the following formula:

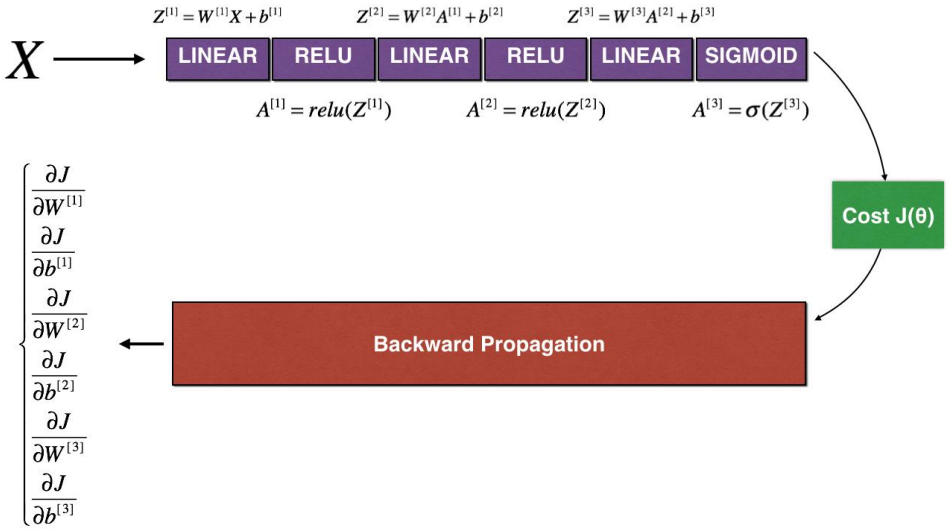
$$difference = \frac{||grad - gradapprox||_2}{||grad||_2 + ||gradapprox||_2}$$

   3 steps to compute this formula:
   a) Compute the numerator using np.linalg.norm(...)
   b) Compute the denominator. You will need to call np.linalg.norm(...) twice.
   c) 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.

N-dimensional gradient checking

Forward and backward propagation (for deep neural network – *LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID*):



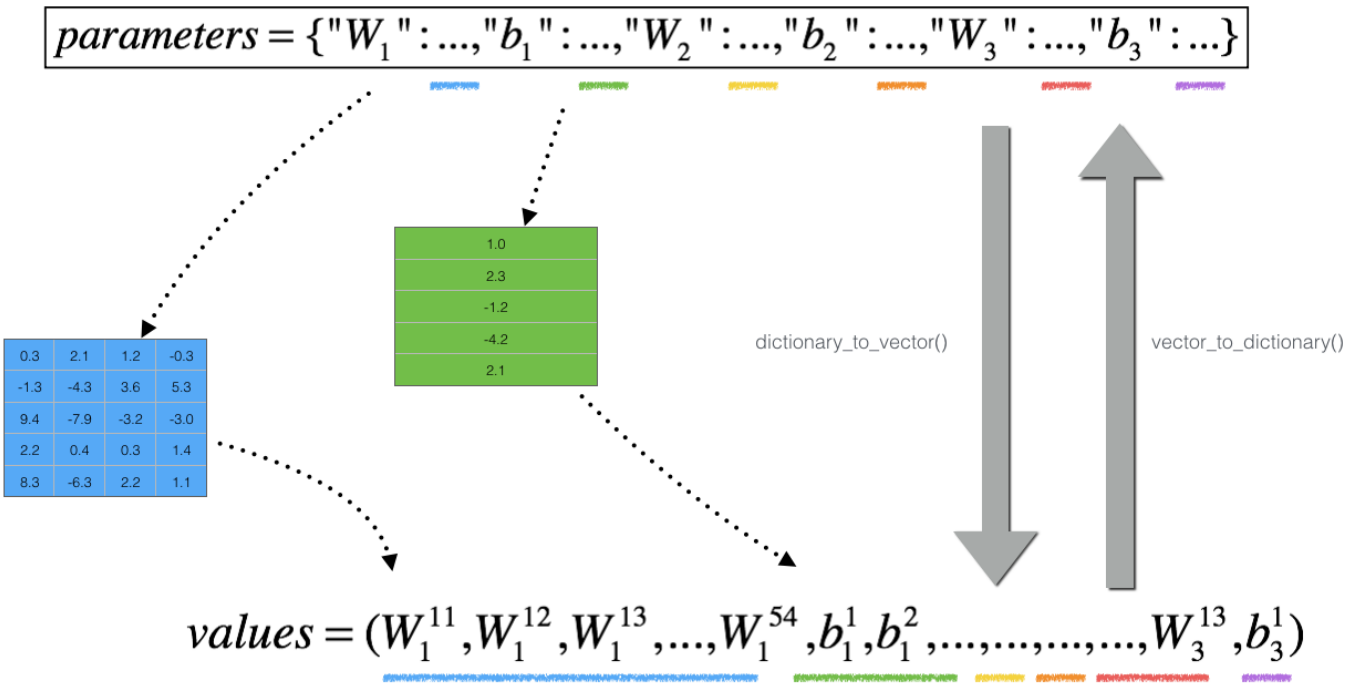
The definition of a derivative (or gradient):

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

! ***θ* is a dictionary called 'parameters', not a scalar.**

dictionary_to_vector() – Converts the "parameters" (θ) dictionary into a vector called "values", obtained by reshaping all parameters (W1, b1, W2, b2, W3, b3) into vectors and concatenating them.

vector_to_dictionary() – The inverse function, outputs back the "parameters" dictionary.



gradients_to_vector() – Converts the "gradients" dictionary into a vector "grad"

Gradient check implementation

For each i in num_parameters:

1) To compute $J\_plus[i]$:
    a) Set θ^+ to *np.copy(parameters_values*)
    b) Set θ_i^+ to $\theta_i^+ + \varepsilon$
    c) Calculate J_i^+ using *forward_propagation_n(x, y, vector_to_dictionary(θ^+))*
2) Compute $J\_minus[i]$ similarly using θ^-
3) Compute $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$
4) Compute the gradients vector *grad* using backward propagation.
5) Compare *gradapprox* vector to *grad* vector. Compute:

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

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 do not run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.

Gradient Checking 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.