A screenshot of a cell phone

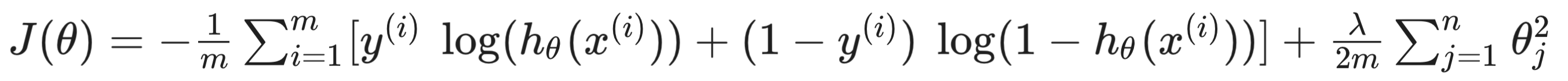
Description automatically generated

Cost Function

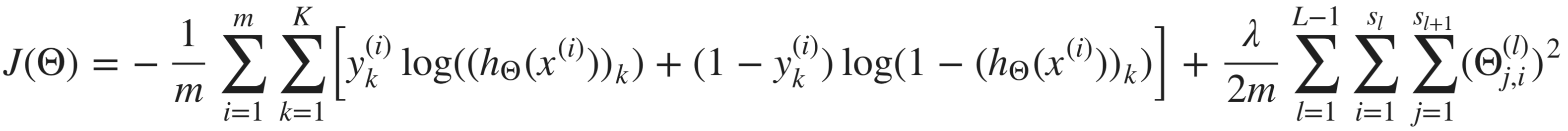
Let's first define a few variables that we will need to use:

* L = total number of layers in the network
* *sl*​ = number of units (not counting bias unit) in layer l
* K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote hΘ(x)k as being a hypothesis that results in the k^{th}*kth* output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was:



For neural networks, it is going to be slightly more complicated:

****

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

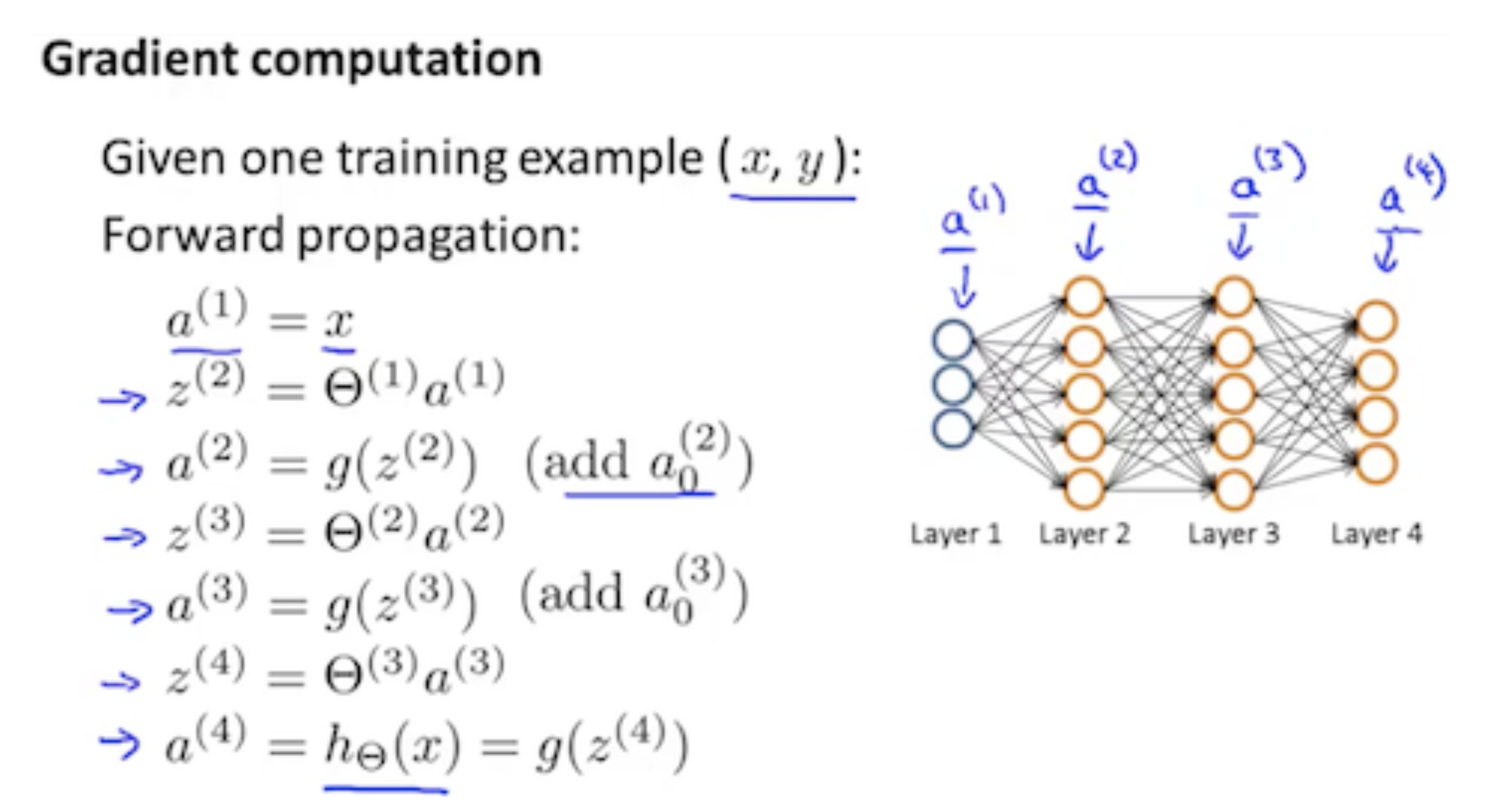
**Back propagation Algorithm**

A close up of text on a white background

Description automatically generated

A picture containing bird, tree, flower

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A close up of a tree

Description automatically generated

Gradient Checking

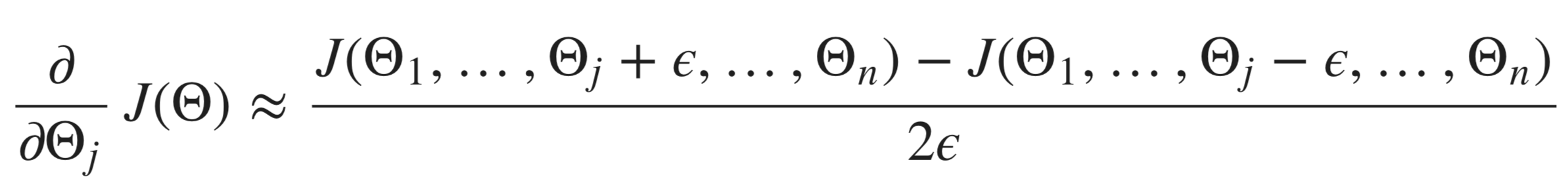
Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

A close up of a logo

Description automatically generated

With multiple theta matrices, we can approximate the derivative **with respect to** Θj as follows:

A small value for ϵ (epsilon) such as ϵ=10−4, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.



We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

Once you have verified **once** that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the following method:

A picture containing monitor

Description automatically generated

Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

* Number of input units = dimension of features *x*(*i*)
* Number of output units = number of classes
* Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
* Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

**Training a Neural Network**

1. Randomly initialize the weights
2. Implement forward propagation to get hΘ(x(i)) for any *x*(*i*)
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

When we perform forward and back propagation, we loop on every training example: