Deep Neural Networks from scratch in Python

Piotr Babel Follow
Jun 11 · 7 min read ★

In this guide we will build a deep neural network, with as many layers as you want! The network can be applied to supervised learning problem with binary classification.

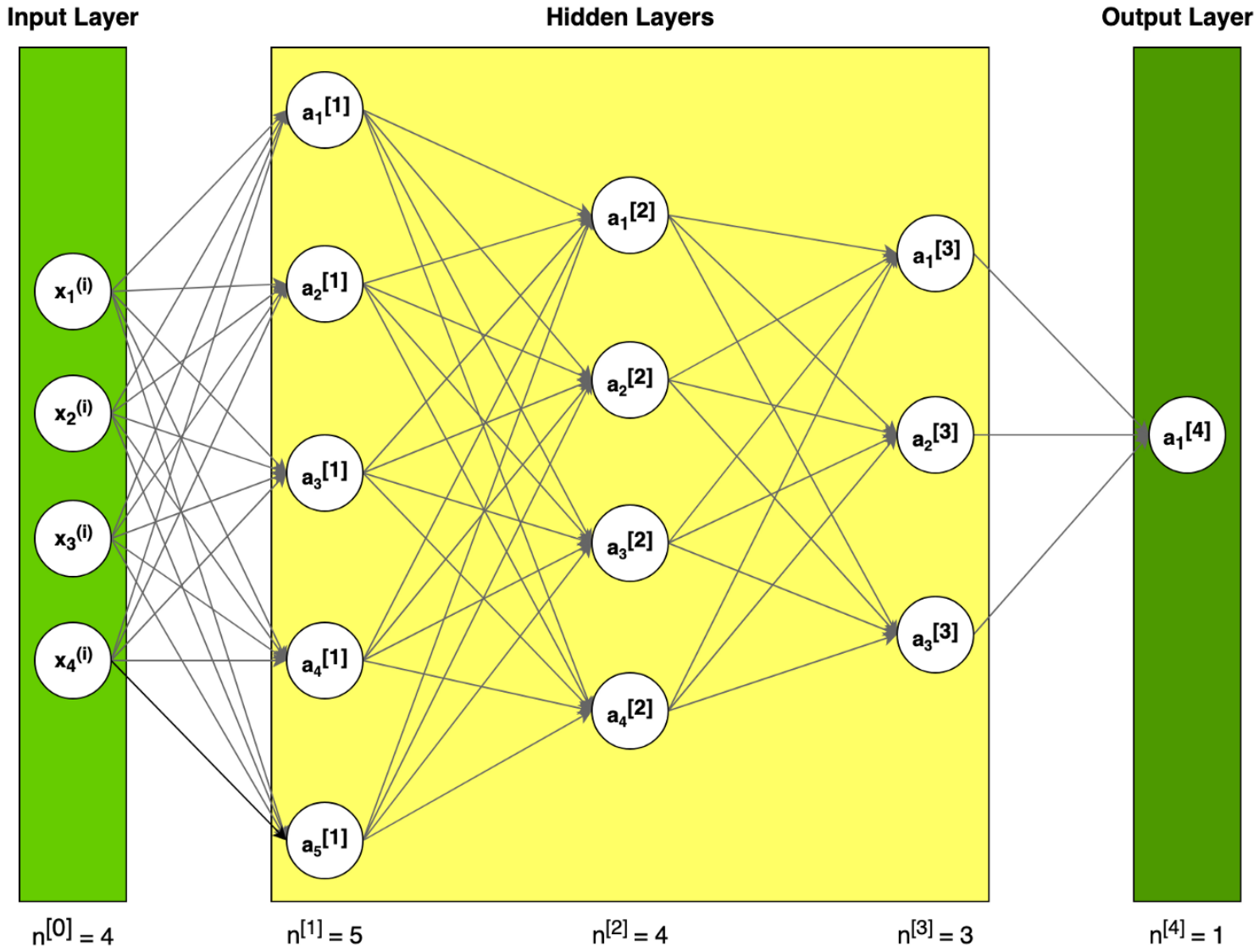

Figure 1. Example of neural network architecture

Notation

Superscript $[l]$ denotes a quantity associated with the l^{th} layer.

Superscript (i) denotes a quantity associated with the i^{th} example.

Lowerscript i denotes the i^{th} entry of a vector.

. . .

This article was written assuming that the reader is already familiar with the concept of a neural network. Otherwise, I recommend to read this nice introduction https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6

. . .

Single neuron

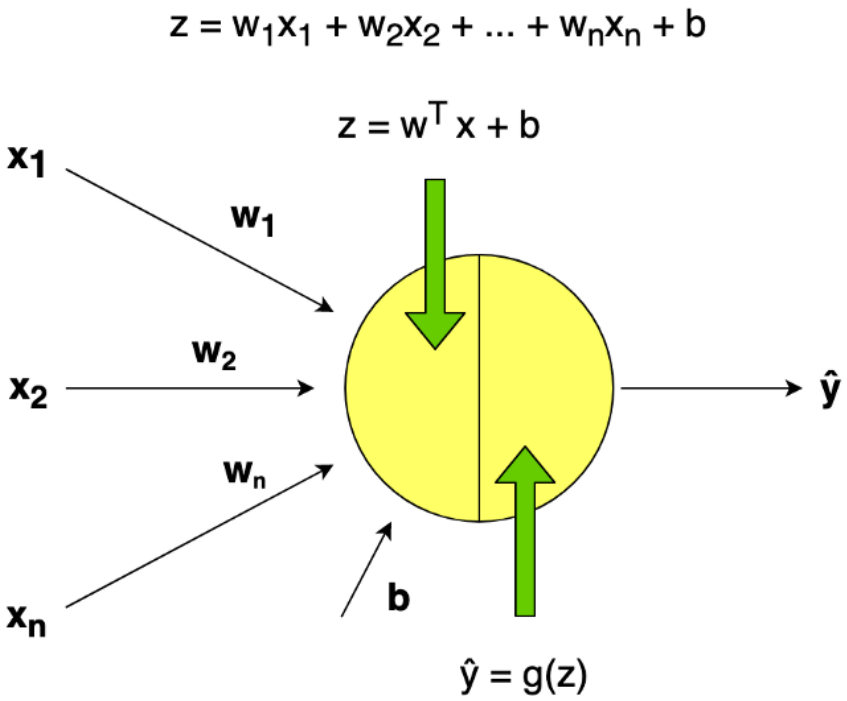

Figure 2. Example of single neuron representation

A neuron computes a linear function (z = Wx + b) followed by an activation function. We generally say that the output of a neuron is a =

g(Wx + b) where g is the activation function (sigmoid, tanh, ReLU, …).

Dataset

Let's assume that we have a very big dataset with weather data such as temperature, humidity, atmospheric pressure and the probability of rain.

Problem statement:

- a training set of m_train weather data labeled as rain (1) or not (0)
- a test set of m_test weather data labeled as rain or not
- each weather data consists of x1 = temperature, x2 = humidity, x3 = atmospheric pressure

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you subtract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array.

General methodology (building the parts of our algorithm)

We will follow the Deep Learning methodology to build the model:

1. Define the model structure (such as number of input features)
2. Initialize parameters and define hyperparameters:

- number of iterations
- number of layers L in the neural network

- size of the hidden layers
- learning rate α

3. Loop for num_iterations:

- Forward propagation (calculate current loss)
- Compute cost function
- Backward propagation (calculate current gradient)
- Update parameters (using parameters, and grads from backprop)

4. Use trained parameters to predict labels

Initialization

The initialization for a deeper L-layered neural network is more complicated because there are many more weight matrices and bias vectors. I provide the tables below in order to help you keep the right dimensions of the structures.

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, n^{[0]})$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]} X + b^{[1]}$	$(n^{[1]}, m)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$	$(n^{[2]}, m)$
...	...	...	...	...
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]} A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, m)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$	$(n^{[L]}, m)$

Table 1. Dimensions of weight matrix **W,** bias vector **b** and activation **Z** for layer l

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	(5, 4)	(5, 1)	$Z^{[1]} = W^{[1]} X + b^{[1]}$	(5, m)
Layer 2	(4, 5)	(4, 1)	$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$	(4, m)
Layer 3	(3, 4)	(3, 1)	$Z^{[3]} = W^{[3]} A^{[2]} + b^{[3]}$	(3, m)
Layer 4	(1, 3)	(1, 1)	$Z^{[4]} = W^{[4]} A^{[3]} + b^{[4]}$	(1, m)

Table 2. Dimensions of weight matrix **W,** bias vector **b** and activation **Z** for the neural network for our example architecture

Table 2 helps us prepare correct dimensions for the matrices of our example neural network architecture from Figure 1.

```
import numpy as np
import matplotlib.pyplot as plt

nn_architecture = [
    {"layer_size": 4, "activation": "none"}, # input layer
    {"layer_size": 5, "activation": "relu"},
    {"layer_size": 4, "activation": "relu"},
    {"layer_size": 3, "activation": "relu"},
    {"layer_size": 1, "activation": "sigmoid"}
]

def initialize_parameters(nn_architecture, seed = 3):
    np.random.seed(seed)
    # python dictionary containing our parameters "W1", "b1
    parameters = {}
    number_of_layers = len(nn_architecture)
```

Snippet 1. Initialization of the parameters

Parameters initialization using small random numbers is simple approach, but it guarantees good enough starting point for our algorithm.

Remember:

- Different initialization techniques such as Zero, Random, He or Xavier lead to different result
- Random initialization makes sure different hidden units can learn different things (initializing all the weights to zero causes, that every neuron in each layer will learn the same thing)
- Don't initialize to values that are too large

Activation functions

Activation functions give the neural networks non-linearity. In our example, we will use sigmoid and ReLU.

Sigmoid outputs a value between 0 and 1 which makes it a very good choice for binary classification. You can classify the output as 0 if it is less than 0.5 and classify it as 1 if the output is more than 0.5.

```
def sigmoid(Z):
    S = 1 / (1 + np.exp(-Z))
    return S

def relu(Z):
    R = np.maximum(0, Z)
    return R

def sigmoid_backward(dA, Z):
    S = sigmoid(Z)
    dS = S * (1 - S)
    return dA * dS
```

Snippet 2. Sigmoid and ReLU activation functions and their derivatives

In Snippet 2 you can see the vectorized implementation of activation functions and their derivatives (https://en.wikipedia.org/wiki/Derivative). The code will be used in the further calculation.

Forward propagation

During forward propagation, in the forward function for a layer *l* you need to know what the activation function in a layer is (Sigmoid, tanh, ReLU, etc.). Given input signal from the previous layer, we compute Z and then apply selected activation function.

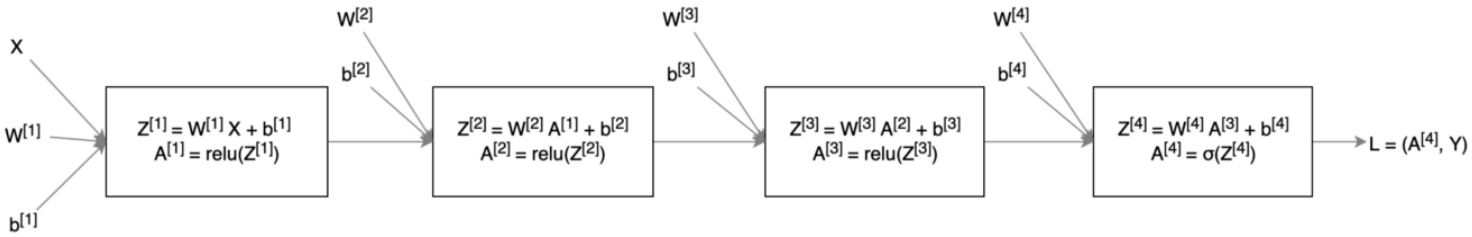

Figure 3. Forward propagation for our example neural network

The linear forward module (vectorized over all the examples) computes the following equations:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

Equation 1. Linear forward function

```
def L_model_forward(X, parameters, nn_architecture):
    forward_cache = {}
    A = X
    number_of_layers = len(nn_architecture)

    for l in range(1, number_of_layers):
        A_prev = A
        W = parameters['W' + str(l)]
        b = parameters['b' + str(l)]
        activation = nn_architecture[l]["activation"]
        Z, A = linear_activation_forward(A_prev, W, b, acti
        forward_cache['Z' + str(l)] = Z
        forward_cache['A' + str(l)] = A

    AL = A

    return AL, forward_cache

def linear_activation_forward(A_prev, W, b, activation):
    if activation == "sigmoid":
```

Snippet 3. Forward propagation module

We use “cache” (Python dictionary, which contains **A** and **Z** values computed for particular layers) to pass variables computed during forward propagation to the corresponding backward propagation step. It contains useful values for backward propagation to compute derivatives.

Loss function

In order to monitor the learning process, we need to calculate the value of the cost function. We will use the below formula to calculate the

cost.

$$-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(a^{[L](i)})+(1-y^{(i)})log(1-a^{[L](i)}))$$

Equation 2. Cross-entropy cost

```
def compute_cost(AL, Y):
    m = Y.shape[1]

    # Compute loss from AL and y
    logprobs = np.multiply(np.log(AL),Y) + np.multiply(1 -
    # cross-entropy cost
    cost = - np.sum(logprobs) / m

```

Snippet 4. Computation of the cost function

Backward propagation

Backpropagation is used to calculate the gradient of the loss function with respect to the parameters. This algorithm is the recursive use of a "chain rule" known from differential calculus.

Equations used in backpropagation calculation:

$$dZ^{[l]} = \frac{\partial L}{\partial Z^{[l]}}$$
$$dW^{[l]} = \frac{\partial L}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$
$$db^{[l]} = \frac{\partial L}{\partial b^{[l]}} = \frac{1}{m}\sum_{i=1}^{m} dZ^{[l](i)}$$
$$dA^{[l-1]} = \frac{\partial L}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l](i)}$$

Equation 3. Formulas for backward propagation calculation

The chain rule is a formula for calculating the derivatives of composite functions. Composite functions are functions composed of functions inside other function.

$$f(x) = A(B(C(D(x))))$$
$$\frac{df}{dx} = \frac{dA}{dB} \cdot \frac{dB}{dC} \cdot \frac{dC}{dD} \cdot \frac{dD}{dx}$$
$$\frac{df}{dD} = \frac{dA}{dB} \cdot \frac{dB}{dC} \cdot \frac{dC}{dD}$$
$$\frac{df}{dC} = \frac{dA}{dB} \cdot \frac{dB}{dC}$$

Equation 4. Chain rule examples

It is difficult to calculate the loss without "chain rule" (equation 5 as an example).

$$L(W^{[1]}) = L(A^{[4]}(Z^{[4]}(A^{[3]}(Z^{[3]}(A^{[2]}(Z^{[2]}(A^{[1]}(Z^{[1]}(W^{[1]}))))))))$$
$$\frac{dL}{dW^{[1]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}} \cdot \frac{dZ^{[3]}}{dA^{[2]}} \cdot \frac{dA^{[2]}}{dZ^{[2]}} \cdot \frac{dZ^{[2]}}{dA^{[1]}} \cdot \frac{dA^{[1]}}{dZ^{[1]}} \cdot \frac{dZ^{[1]}}{dW^{[1]}}$$

Equation 5. Loss function (with substituted data) and its derivative with respect to the first weight.

The first step in backpropagation for our neural network model is to calculate the derivative of our loss function with respect to Z from the last layer. Equation 6 consists of two components, the derivative of the loss function from equation 2 (with respect to the activation function) and the derivative of the activation function "sigmoid" with respect to Z from the last layer.

$$\frac{dL}{dZ^{[4]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}}$$

Equation 6. The derivative of the loss function with respect to Z from 4^{th} layer

The result from equation 6 can be used to calculate the derivatives from equation 3:

$$dA^{[3]} = \frac{\partial L}{\partial A^{[3]}} = W^{[4]T} dZ^{[4](i)}$$

Equation 7. The derivative of the loss function with respect to A from 3th layer

The derivative of the loss function with respect to the activation function from the third layer (equation 7) is used in the further calculation.

$$\frac{dL}{dZ^{[3]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}}$$
$$\frac{dL}{dA^{[3]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}}$$
$$\frac{dL}{dZ^{[3]}} = \frac{dL}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}}$$

Equation 8. The derivatives for the third layer

The result from equation 7 and the derivative of the activation function "ReLU" from the third layer is used to calculate the derivatives from equation 8 (the derivative of the loss function with respect to Z). Following this, we make a calculation for equation 3.

We make similar calculations for equation 9 and 10.

$$\frac{dL}{dZ^{[2]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}} \cdot \frac{dZ^{[3]}}{dA^{[2]}} \cdot \frac{dA^{[2]}}{dZ^{[2]}}$$
$$\frac{dL}{dA^{[2]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}} \cdot \frac{dZ^{[3]}}{dA^{[2]}}$$
$$\frac{dL}{dZ^{[2]}} = \frac{dL}{dA^{[2]}} \cdot \frac{dA^{[2]}}{dZ^{[2]}}$$

Equation 9. The derivatives for the second layer

$$\frac{dL}{dZ^{[1]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}} \cdot \frac{dZ^{[3]}}{dA^{[2]}} \cdot \frac{dA^{[2]}}{dZ^{[2]}} \cdot \frac{dZ^{[2]}}{dA^{[1]}} \cdot \frac{dA^{[1]}}{dZ^{[1]}}$$
$$\frac{dL}{dA^{[1]}} = \frac{dL}{dA^{[4]}} \cdot \frac{dA^{[4]}}{dZ^{[4]}} \cdot \frac{dZ^{[4]}}{dA^{[3]}} \cdot \frac{dA^{[3]}}{dZ^{[3]}} \cdot \frac{dZ^{[3]}}{dA^{[2]}} \cdot \frac{dA^{[2]}}{dZ^{[2]}} \cdot \frac{dZ^{[2]}}{dA^{[1]}}$$
$$\frac{dL}{dZ^{[1]}} = \frac{dL}{dA^{[1]}} \cdot \frac{dA^{[1]}}{dZ^{[1]}}$$

Equation 10. The derivatives for the first layer

The general idea:

The derivative of the loss function with respect to Z from l^{th} layer helps to calculate the derivative of the loss function with respect to A from $(l-1)^{th}$ layer (the previous layer). Then the result is used with the derivative of the activation function.

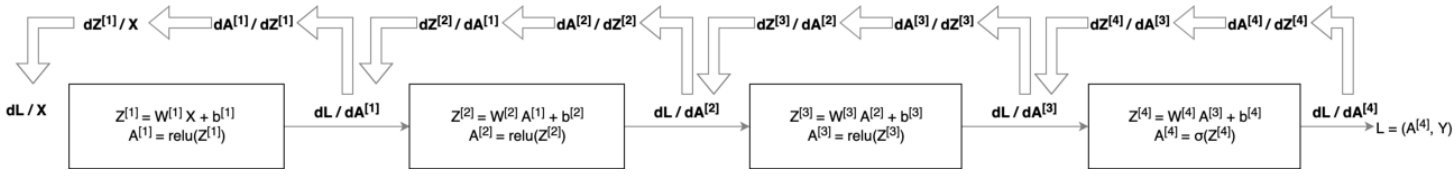

Figure 4. Backward propagation for our example neural network

```python
def L_model_backward(AL, Y, parameters, forward_cache, nn_a
    grads = {}
    number_of_layers = len(nn_architecture)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the sam

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
    dA_prev = dAL

    for l in reversed(range(1, number_of_layers)):
        dA_curr = dA_prev

        activation = nn_architecture[l]["activation"]
        W_curr = parameters['W' + str(l)]
        Z_curr = forward_cache['Z' + str(l)]
        A_prev = forward_cache['A' + str(l-1)]

        dA_prev, dW_curr, db_curr = linear_activation_backw

        grads["dW" + str(l)] = dW_curr
        grads["db" + str(l)] = db_curr

    return grads

def linear_activation_backward(dA, Z, A_prev, W, activation
    if activation == "relu":
        dZ = relu_backward(dA, Z)
```

Snippet 5. Backward propagation module

Update parameters

The goal of the function is to update the parameters of the model using gradient optimization.

```
def update_parameters(parameters, grads, learning_rate):
    L = len(parameters)

    for l in range(1, L):
        parameters["W" + str(l)] = parameters["W" + str(l)]
        parameters["b" + str(l)] = parameters["b" + str(l)]
```

Snippet 6. Updating parameters values using gradient descent

Full model

The full implementation of the neural network model consists of the methods provided in snippets.

```
def L_layer_model(X, Y, nn_architecture, learning_rate = 0.
    np.random.seed(1)
    # keep track of cost
    costs = []

    # Parameters initialization.
    parameters = initialize_parameters(nn_architecture)

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LI
        AL, forward_cache = L_model_forward(X, parameters,

        # Compute cost.
        cost = compute_cost(AL, Y)

        # Backward propagation.
        grads = L_model_backward(AL, Y, parameters, forward

        # Update parameters.
        parameters = update_parameters(parameters, grads, l

        # Print the cost every 100 training example
```

Snippet 7. The full model of the neural network

In order to make a prediction, you only need to run a full forward propagation using the received weight matrix and a set of test data.

You can modify *nn_architecture* in **Snippet 1** to build a neural network with a different number of layers and sizes of the hidden layers. Moreover, prepare the correct implementation of the activation functions and their derivatives (**Snippet 2**). The implemented functions can be used to modify *linear_activation_forward* method in **Snippet 3** and *linear_activation_backward* method in **Snippet 5**.

Further improvements

You can face the "overfitting" problem if the training dataset is not big enough. It means that the learned network doesn't generalize to new examples that it has never seen. You can use **regularization** methods such as **L2 regularization** (it consists of appropriately modifying your cost function) or **dropout** (it randomly shuts down some neurons in each iteration).

We used **Gradient Descent** to update the parameters and minimize the cost. You can learn more advanced optimization methods that can speed up learning and even get you to a better final value for the cost function for example:

- Mini-batch gradient descent
- Momentum
- Adam optimizer

. . .

References:

[1] https://www.coursera.org/learn/neural-networks-deep-learning

[2] https://www.coursera.org/learn/deep-neural-network

[3] https://ml-cheatsheet.readthedocs.io/en/latest/index.html

[4] https://medium.com/towards-artificial-intelligence/one-lego-at-a-time-explaining-the-math-of-how-neural-networks-learn-with-

implementation-from-scratch-39144a1cf80

[5] https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0

[6] https://medium.com/datadriveninvestor/math-neural-network-from-scratch-in-python-d6da9f29ce65

[7] https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6