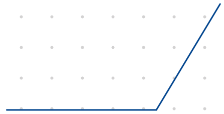


ReLU → Rectified

BACK PROPAGATION

$dW^{[1]} = dz^{[1]} x$

$db^{[1]} = dz^{[1]}$

$W^{[2]}$

$b^{[2]}$

x, $W^{[1]}$, $b^{[1]}$ → $z^{[1]} = W^{[1]}x + b^{[1]}$ → $a^{[1]} = \sigma(z^{[1]})$ → $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ → $a^{[2]} = \sigma(z^{[2]})$ → $\mathcal{L}(a^{[2]}, y)$

$dW^{[2]} = dz^{[2]} \cdot \dfrac{dz^{[2]}}{dW^{[2]}} = dz^{[2]}(a^{[1]})^T$

$db^{[2]} = dz^{[2]}$

$dz^{[1]} = dz^{[2]} W^{[2]} * g^{[1]\prime}(z^{[1]})$

element wise product

$da^{[1]} = dz^{[2]} \dfrac{dz^{[2]}}{da^{[1]}}$

$= dz^{[2]} W^{[2]}$

$dz^{[2]} = \dfrac{d\mathcal{L}}{dz^{[2]}}$

$= \dfrac{d\mathcal{L}}{da^{[2]}} \dfrac{da^{[2]}}{dz^{[2]}}$

$= a^{[2]} - y$

$da^{[2]} = \dfrac{d\mathcal{L}}{da^{[2]}}$

L- Layer Neural Network

size of input $X = (12288, 209)$ → $m = 209$ $\quad n^{[0]} = 12288$

	Shape of W	Shape of b	Activation	Shape of Function
Layer 1	$n^{[1]} \times 12288$	$n^{[1]} \times 1$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$n^{[1]} \times 209$
Layer 2	$n^{[2]} \times n^{[1]}$	$n^{[2]} \times 1$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$n^{[2]} \times 209$
⋮	⋮	⋮	⋮	⋮
Layer L-1	$n^{[L-1]} \times n^{[L-2]}$	$n^{[L-1]} \times 1$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$n^{[L-1]} \times 209$
Layer L	$n^{[L]} \times n^{[L-1]}$	$n^{[L]} \times 1$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$n^{[L]} \times 209$

initialize parameters :

layer_dims: contains the dimensions of each layer in the network

```
def init_param (layer_dims) :
    parameters = { }
    for l in range (1, L):
        parameters ['W' + str(l)] = np.random.randn (layer_dims [l] , layer_dims [l-1]) * 0.01
        parameters ['b' + str(l)] = np.zeros ((layer_dims [l], 1))

    return parameters
```

forward propagation

1. Linear forward $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ where $A^{[0]} = X$
2. Activation forward $A^{[l]} = g(Z^{[l]})$

A: activations from prev layer or input
W: numpy array of shape ($n_{current}$, n_{prev})
b: numpy array of shape ($n_{current}$, 1)
Z: input of the activation function
cache: for backward pass store A, W, b

all together

```
def linear_forward (A, W, b):

    Z = np.dot(W, A) + b
    cache = (A, W, b)
    return Z, cache
```

```
def linear_activation_forward (A_prev, W, b, activation):

    if activation == 'sigmoid':

        Z, linear_cache = linear_forward (A_prev, W, b)
        A, activation_cache = sigmoid (Z)

    elif activation == 'relu':

        Z, linear_cache = linear_forward (A_prev, W, b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache
```

```
def L_model_forward (X, parameters):

    L = len (parameters) // 2
    caches = [ ]
    A = X

    for l in range (1, L):
        A_prev = A
        A, cache = linear_activation_forward (A_prev,
                                              parameters['W' + str(l)],
                                              parameters ['b' + str (l)],
                                              'relu')

        caches.append (cache)

    AL, cache = linear_activation_forward (A, parameters['W' + str(L)],
                                           parameters['b' + str(L)], 'sigmoid')

    caches.append (cache)

    return AL, caches
```

Cost Function

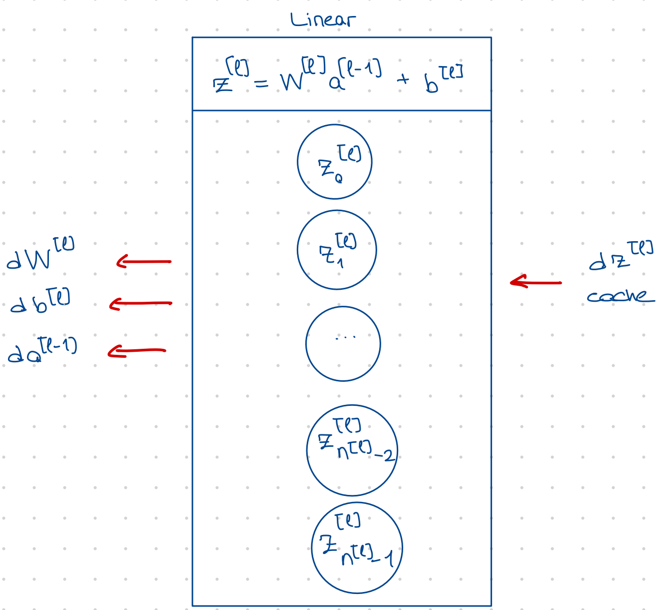
Cross entropy cost $J = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(a^{[L](i)}\right) + 1 - y^{(i)} \log\left(1 - a^{[L](i)}\right) \right]$

```
def compute_cost(AL, Y):

    m = Y.shape[1]

    cost = (-1/m) * np.dot(Y, np.log(AL).T) + np.dot((1-Y), np.log(1-AL).T))

    cost = np.squeeze(cost)
    assert(cost.shape == ())

    return cost
```

Backward Propagation Module

1. Linear Backward
2. Activation Backward

For layer ℓ, linear forward: $Z^{[\ell]} = W^{[\ell]} X + b^{[\ell]}$ followed by an activation



$$dW^{[\ell]} = \frac{\partial J}{\partial W^{[\ell]}} = \frac{1}{m} dZ^{[\ell]} A^{[\ell-1]T}$$

$$dA^{[\ell-1]} = \frac{\partial L}{\partial A^{[\ell-1]}} = W^{[\ell]T} dZ^{[\ell]}$$

$$db^{[\ell]} = \frac{\partial J}{\partial b^{[\ell]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[\ell](i)}$$

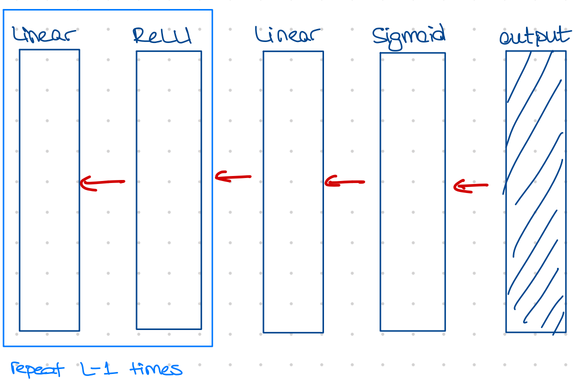
```
def linear_backward(dZ, cache):

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = (1/m) * np.dot(dZ, A_prev.T)

    db = (1/m) * np.sum(dZ, axis=1, keepdims=True)

    dA_prev = np.dot(W.T, dZ)

    return dW, db, dA_prev
```

```
def linear_activation_backward(dA, cache, activation):
```

$\rightarrow dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$

```
    linear_cache, activation_cache = cache

    if activation == 'relu':
        dZ = relu_backward(dA, activation_cache)

    elif activation == 'sigmoid':
        dZ = sigmoid_backward(dA, activation_cache)

    dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

L-Model Backward



dAL = -(np.divide(Y, AL) - np.divide(1-Y, 1-AL)) $\Rightarrow da = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right)$

```
def L_model_backward(AL, Y, caches):

    grads = {}
    L = len(caches) #num of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

    #Initialize backprop
    dAL = -(np.divide(Y, AL) - np.divide(1-Y, 1-AL))

    current_cache = caches[L-1]  # last layer

    grads['dA' + str(L-1)], grads['dW' + str(L)], grads['db' + str(L)]

            = linear_activation_backward(dAL, current_cache, 'sigmoid')

    for l in reversed(range(L-1)):

        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp
        = linear_activation_backward(grads['dA' + str(l+1)], current_cache, activation='relu')

        grads['dA' + str(l)] = dA_prev_temp
        grads['dW' + str(l+1)] = dW_temp
        grads['db' + str(l+1)] = db_temp

    return grads
```

Update parameters

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]}$$

where α : learning rate

$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$$

```
def update_parameters(parameters, grads, learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        parameters['W' + str(l+1)] = parameters['W' + str(l+1)]
                                     - learning_rate * grads['dW' + str(l+1)]

        parameters['b' + str(l+1)] = parameters['b' + str(l+1)]
                                     - learning_rate * grads['db' + str(l+1)]

    return parameters
```