Regularization

L2 regularization:

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

$$\Big\downarrow \text{Regularization}$$

cross-entropy cost

$$J = \boxed{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(a^{[L](i)}) + (1-y^{(i)})\log(1-a^{[L](i)})\right)} + \boxed{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\left(\sum_{k}\sum_{j} W_{kj}^{[l]2}\right)}$$

number of examples

L2 regularization Cost

$$\sum_{k}\sum_{j} W_{kj}^{[l]2}$$

→ np.sum(np.square(W1))

Return the element-wise square of input

backward propagation: $\frac{d}{dw}\left(\frac{1}{m}\frac{\lambda}{2}W^2\right) = \frac{\lambda}{m}W$

```python
# GRADED FUNCTION: backward_propagation_with_regularization

def backward_propagation_with_regularization(X, Y, cache, lambd):
    """
    Implements the backward propagation of our baseline model to which we added an L2 regularization.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    cache -- cache output from forward_propagation()
    lambd -- regularization hyperparameter, scalar

    Returns:
    gradients -- A dictionary with the gradients with respect to each parameter, activation and pre-activation variables
    """

    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

    dZ3 = A3 - Y

    ### START CODE HERE ### (approx. 1 line)
    dW3 = 1./m * np.dot(dZ3, A2.T) + lambd / m * W3
    ### END CODE HERE ###
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    ### START CODE HERE ### (approx. 1 line)
    dW2 = 1./m * np.dot(dZ2, A1.T) + lambd / m * W2
    ### END CODE HERE ###
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    ### START CODE HERE ### (approx. 1 line)
    dW1 = 1./m * np.dot(dZ1, X.T) + lambd / m * W1
    ### END CODE HERE ###
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
                 "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients
```

Dropout

1. $D^{[l]} = [\, d^{[l](1)} \quad d^{[l](2)} \; \cdots\cdots \; d^{[l](m)} \,]$

np. random. rand( )

$\rightarrow$ $d^{[l]}$ randomly get numbers between 0 and 1

D1 = np.random.rand(A1.shape[0] , A1 . shape[1])

2. D1 $\begin{cases} 1-\text{keep\_prob} \rightarrow 0 \\ \text{keep\_prob} \longrightarrow 1 \end{cases}$

D1 = D1 < (keep_prob)

3. Set $A^{[l]} = A^{[l]} * D^{[l]}$

4. Divide $A^{[l]}$ by keep_prob

(inverted dropout)

A1 = A1 / keep_prob

Backward propagation:

1. $D^{[l]} \rightarrow dA1$ $\quad dA1 = dA1 * D1$

2. $dA1 = \dfrac{dA1}{\text{keep\_prob}}$

Dropout : Randomly omit some perceptrons from the hidden layer on each training instance