```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)


# import libraries for plotting
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline


# ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

```
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(1)

    W1 = np.random.randn(n_h,n_x)*0.01
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,n_h)*0.01
    b2 = np.zeros((n_y,1))

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation parameter
    cache -- a python tuple containing "A", "W" and "b" ; stored for computing the backward pass efficiently
    """


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


    #assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

```

def sigmoid(Z):

    s = 1/(1+np.exp(-Z))

    return s,Z
def sigmoid_backward(dA,cache):
    Z = cache
    dZ = np.array(dA, copy = True)
    dZ[Z <=0] = 0
    return dZ
```

```
def relu(x):
    f = np.maximum(x, 0)
    return f,x
def relu_backward(dA,cache):
    Z = cache
    s = 1/(1+np.exp(-Z))
    dZ = dA*s*(1-s)
    return dZ
```

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the post-activation value
    cache -- a python tuple containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".

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


    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".

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


    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

```
def compute_cost(AL, Y):

    m = Y.shape[1]
    # Compute loss from aL and y.

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

```
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (layer l)
    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the current layer
    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l),same shape as W
    db -- Gradient of the cost with respect to b (current layer l),same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = np.dot(dZ,A_prev.T)/m
    db = np.sum(dZ,axis=1,keepdims=True)/m
    dA_prev = np.dot(W.T,dZ)
    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    #assert (db.shape == b.shape)
    return dA_prev, dW, db
```

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

    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

```
def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent
    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of
    ,→ L_model_backward
    1. NEURAL NETWORKS AND DEEP LEARNING 77
    Returns:
    parameters -- python dictionary containing your updated parameters
    parameters["W" + str(l)] = ...
    parameters["b" + str(l)] = ...
    """
    L = len(parameters) // 2 # number of layers in the neural network
    # Update rule for each parameter. Use a for loop.

    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 parameter
```

```
def L_model_forward(X, parameters):
    caches = []
    A = X

    # number of layers in the neural network
    L = len(parameters) // 2

    # Using a for loop to replicate [LINEAR->RELU] (L-1) times
    for l in range(1, L):
        A_prev = A

        # Implementation of LINEAR -> RELU.
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], activation = "relu")

        # Adding "cache" to the "caches" list.
        caches.append(cache)


    # Implementation of LINEAR -> SIGMOID.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)], activation = "sigmoid")

    # Adding "cache" to the "caches" list.
    caches.append(cache)

    return AL, caches
```

```
def initialize_parameters_deep(layer_dimensions):
    parameters = {}

    # number of layers in the network
    L = len(layer_dimensions)

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dimensions[l], layer_dimensions[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dimensions[l], 1))

    return parameters
```

```
def L_model_backward(AL, Y, caches):
    grads = {}

    # the number of layers
    L = len(caches)
    m = AL.shape[1]

    # after this line, Y is the same shape as AL
    Y = Y.reshape(AL.shape)

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

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "dAL, current_cache". Outputs: "grads["dAL-1"], grads["dWL"], grads["dbL"]
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache, "sigmoid")

    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 1)], current_cache".
        # Outputs: "grads["dA" + str(l)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]

        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA"+str(l+1)], current_cache, "relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

```
def update_parameters(parameters, grads, learning_rate):
    # number of layers in the neural network
    L = len(parameters) // 2

    # Update rule for each parameter
    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
```

```
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):#lr was 0.009
    # keep track of cost
    costs = []

    # Parameters initialization.
    parameters = initialize_parameters_deep(layers_dims)

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

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

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

        # Backward propagation.
        grads = L_model_backward(AL, Y, caches)

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

```
def predict(X, parameters):
    m = X.shape[1]

    # number of layers in the neural network
    n = len(parameters) // 2
    p = np.zeros((1,m))

    # Forward propagation
    probas, caches = L_model_forward(X, parameters)

    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

    return p
```

```
#importation et visualisation de la dataset
data = pd.read_csv('C:/Users/yassine/Downloads/kag_risk_factors_cervical_cancer.csv')
data.head()
```

	Age	Number of sexual partners	First sexual intercourse	Num of pregnancies	Smokes	Smokes (years)	Smokes (packs/year)	Hormonal Contraceptives	Hormonal Contraceptives (years)	IUD	...	STDs: Time since first diagnosis	STDs: Time since last diagnosis	Dx:Cancer	Dx:CIN	Dx:HPV	Dx	Hinselmann	Schill
0	18	4.0	15.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	?	?	0	0	0	0	0	0
1	15	1.0	14.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	?	?	0	0	0	0	0	0
2	34	1.0	?	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	?	?	0	0	0	0	0	0
3	52	5.0	16.0	4.0	1.0	37.0	37.0	1.0	3.0	0.0	...	?	?	1	0	1	0	0	0
4	46	3.0	21.0	4.0	0.0	0.0	0.0	1.0	15.0	0.0	...	?	?	0	0	0	0	0	0

5 rows × 36 columns

```
#Gestion des données manquantes
data = data.replace('?', np.nan)
data.head()
```

	Age	Number of sexual partners	First sexual intercourse	Num of pregnancies	Smokes	Smokes (years)	Smokes (packs/year)	Hormonal Contraceptives	Hormonal Contraceptives (years)	IUD	...	STDs: Time since first diagnosis	STDs: Time since last diagnosis	Dx:Cancer	Dx:CIN	Dx:HPV	Dx	Hinselmann	Schill
0	18	4.0	15.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	NaN	NaN	0	0	0	0	0	0
1	15	1.0	14.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	NaN	NaN	0	0	0	0	0	0

	Age	Number of sexual partners	First sexual intercourse	Num of pregnancies	Smokes	Smokes (years)	Smokes (packs/year)	Hormonal Contraceptives	Hormonal Contraceptives (years)	IUD	...	STDs: Time since first diagnosis	STDs: Time since last diagnosis	Dx:Cancer	Dx:CIN	Dx:HPV	Dx	Hinselmann	Schill
2	34	1.0	NaN	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	NaN	NaN	0	0	0	0	0	0
3	52	5.0	16.0	4.0	1.0	37.0	37.0	1.0	3.0	0.0	...	NaN	NaN	1	0	1	0	0	0
4	46	3.0	21.0	4.0	0.0	0.0	0.0	1.0	15.0	0.0	...	NaN	NaN	0	0	0	0	0	0

5 rows × 36 columns

```
data.drop(['STDs: Time since first diagnosis','STDs: Time since last diagnosis'],inplace=True,axis=1)
```

```
data=data.dropna()
```

```
data.shape
```

```
(668, 34)
```

```
data.describe()
```

	Age	STDs: Number of diagnosis	Dx:Cancer	Dx:CIN	Dx:HPV	Dx	Hinselmann	Schiller	Citology	Biopsy
count	668.000000	668.000000	668.000000	668.000000	668.000000	668.000000	668.000000	668.000000	668.000000	668.000000
mean	27.264970	0.092814	0.025449	0.004491	0.023952	0.023952	0.044910	0.094311	0.058383	0.067365
std	8.727432	0.310355	0.157603	0.066915	0.153015	0.153015	0.207262	0.292480	0.234642	0.250841
min	13.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	21.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	26.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	33.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max	84.000000	3.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

```
category_data = ['Hinselmann', 'Schiller','Citology', 'Biopsy']
```

```
for feature in category_data:
   sns.factorplot(feature,data=data ,kind='count')
```

```
data['Age'].hist(bins=70)
plt.xlabel('Age')
plt.ylabel('Count')
print('Mean age of the Women facing the risk of Cervical cancer',data['Age'].mean())
```

```
Mean age of the Women facing the risk of Cervical cancer 27.26497005988024
```



```
X = data.drop('Biopsy',axis = 1).to_numpy()
Y = data['Biopsy'].to_numpy()
Y = np.reshape(Y,(668,1))
```

```
from sklearn.decomposition import PCA
# Instantiate and fit PCA to training set

pca = PCA()
pca.fit(X)
```

```
PCA()
```

```
plt.figure(figsize = (10, 6))
plt.clf()
plt.axes([.2, .2, .7, .7])
plt.plot(pca.explained_variance_ratio_, linewidth = 2)
plt.axis('tight')
plt.xlabel('Number of components')
plt.ylabel('Explained variance ratio')
```

```
Text(0, 0.5, 'Explained variance ratio')
```

```
cumsum = np.cumsum(pca.explained_variance_ratio_)
dim = np.argmax(cumsum >= 0.95) + 1
print('The number of dimensions required to preserve 95% of variance is',dim)
```

```
The number of dimensions required to preserve 95% of variance is 6
```

```
pca = PCA(n_components = 6)
pca.fit(X)
X = pca.transform(X)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.30)
X_train= X_train.T
X_test= X_test.T
Y_train=Y_train.T
Y_test=Y_test.T

Y_train.shape
```

```
(6, 467)
```

```
layers_dims = [6,3,3, 1] #  4-layer model
parameters = L_layer_model(X_train, Y_train, layers_dims, learning_rate = 0.05, num_iterations = 5000, print_cost = True)
```

```
Cost after iteration 0: 0.693142
Cost after iteration 100: 0.691020
Cost after iteration 200: 0.691020
Cost after iteration 300: 0.691020
Cost after iteration 400: 0.691020
Cost after iteration 500: 0.691020
Cost after iteration 600: 0.691020
Cost after iteration 700: 0.691020
Cost after iteration 800: 0.691020
Cost after iteration 900: 0.691020
Cost after iteration 1000: 0.691020
Cost after iteration 1100: 0.691020
Cost after iteration 1200: 0.691020
Cost after iteration 1300: 0.691020
Cost after iteration 1400: 0.691020
Cost after iteration 1500: 0.691020
Cost after iteration 1600: 0.691020
Cost after iteration 1700: 0.691020
Cost after iteration 1800: 0.691020
Cost after iteration 1900: 0.691020
Cost after iteration 2000: 0.691020
Cost after iteration 2100: 0.691020
Cost after iteration 2200: 0.691020
Cost after iteration 2300: 0.691020
Cost after iteration 2400: 0.691020
Cost after iteration 2500: 0.691020
Cost after iteration 2600: 0.691020
Cost after iteration 2700: 0.691020
Cost after iteration 2800: 0.691020
Cost after iteration 2900: 0.691020
Cost after iteration 3000: 0.691020
Cost after iteration 3100: 0.691020
Cost after iteration 3200: 0.691020
Cost after iteration 3300: 0.691020
Cost after iteration 3400: 0.691020
Cost after iteration 3500: 0.691020
Cost after iteration 3600: 0.691020
Cost after iteration 3700: 0.691020
Cost after iteration 3800: 0.691020
Cost after iteration 3900: 0.691020
Cost after iteration 4000: 0.691020
Cost after iteration 4100: 0.691020
Cost after iteration 4200: 0.691020
Cost after iteration 4300: 0.691020
Cost after iteration 4400: 0.691020
Cost after iteration 4500: 0.691020
Cost after iteration 4600: 0.691020
Cost after iteration 4700: 0.691020
Cost after iteration 4800: 0.691020
Cost after iteration 4900: 0.691020
```

png

```
print("train accuracy: {} %".format(np.mean(100 - (np.abs(predict(X_train, parameters) - Y_train)/(X_train.shape[1])) * 100)))
print("test accuracy: {} %".format(100 - np.mean(np.abs(predict(X_test, parameters) - Y_test)/(X_test.shape[1])) * 100))
```

```
train accuracy: 99.9853270912334 %
test accuracy: 99.96782257864905 %
```

```
```