Project:

This project was to build an **image recognition algorithm** and to prove that the accuracy of the algorithm increased as we moved from a logistic regression model to a 2 layer neural network and finally to a deep neural network.

Project was done in Python.

Packages used: numpy, h5py, matplotlib, PIL and scripy

Problem Statement: We have a dataset containing:

- a training set of m_train images labeled as cat (y=1) or non-cat (y=0)

- a test set of m_test images labeled as cat or non-cat

- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = num_px) and (width = num_px).

We need to build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat.

Logistic regression:

Program

```
# Import Numpy package

import numpy as np

# Load the data

train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()

# Find the values for: m_train (number of training examples, m_test (number of test examples) and num_px (= height = width of a training image)
m_train = train_set_x_orig.shape[0]

m_test = test_set_x_orig.shape[0]

num_px = train_set_x_orig.shape[1]

# Reshape the training and test data sets so that images of size (num_px, num_px, 3) are flattened into single vectors of shape (num_px * num_px *3, 1).

train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0],-1).T

test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0],-1).T
```

```
#Standardize the picture dataset by dividing every row by 255 (the maximum value of a pixel channel)

train_set_x = train_set_x_flatten/255

test_set_x = test_set_x_flatten/255

#        Sigmoid function

sigmoid(z) = 1/(1+np.exp (-z))

“””

Arguments:

   dim -- number of parameters

   w -- weights, a numpy array of size (num_px * num_px * 3, 1)

    b -- bias, a scalar

    X -- data of size (num_px * num_px * 3, number of examples)

   num_iterations – number of iterations of the optimization loop

    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, number of examples)

    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for the examples in X

“””

#Initialize weight matrix and bias vector

w = np.zeros(shape=(dim,1))

b = 0

#m is number of training examples

m = X.shape[1]

#Forward propagation to calculate current loss

A = sigmoid(np.dot(w.T, X) + b)  # compute activation

cost = (- 1 / m) * np.sum(Y * np.log(A) + (1 - Y) * (np.log(1 - A)))   #compute cost


#Backward propagation to calculate current gradient
```

dw = (1 / m) * np.dot(X, (A - Y).T)

db = (1/m)*np.sum(A-Y)

Update the parameters using gradient descent rule for w and b

For i in range(num_iterations):

w = w-learning_rate*dw

b = b-learning_rate*db

Compute vector "A" predicting the probabilities of a cat being present in the picture

A = sigmoid(np.dot(w.T,X)+b)

for i in range(A.shape[1]):

Convert probabilities A[0,i] to actual predictions p[0,i]

Y_prediction[0,i]=1 if A[0,i]>0.5 else 0

<u>Neural network with a single hidden layer:</u>

“””

Arguments:

X --input dataset of shape (input size, number of examples)

Y –labels of shape (output size, number of examples)

n_x -- size of the input layer

n_h – size of the hidden layer

n_y – size of the output layer

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)

“””

```
# Layer sizes

n_x=X.shape[0] # size of input layer

n_h=4 # setting size of hidden layer to 4

n_y=Y.shape[0] # size of output layer

# Initialize parameters

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(shape=(n_y,1))

# Implement Forward Propagation to calculate A2 (probabilities)

Z1=np.dot(W1,X)+b1

A1=np.tanh(Z1)

Z2=np.dot(W2,A1)+b2

A2=sigmoid(Z2)

# Compute the cross-entropy cost

m=Y.shape[1]

logprobs=np.multiply(np.log(A2,Y)+np.multiply(np.log(1-A2),(1-Y))

cost=-(1/m)*np.sum(logprobs)

# Implement Backward propagation

dZ2=A2-Y

dW2=(1/m)*np.dot(dZ2,A1.T)

db2=(1/m)*np.sum(dZ2,axis=1,keepdims=True)

dZ1=np.multiply(np,dot(W2.TdZ2),1-np.power(A1,2))

dW1=(1/m)*np.dot(dZ1,X.T)

db1=(1/m) )*np.sum(dZ1,axis=1,keepdims=True)
```

```
# Update parameters

W1 = W1-learning_rate*dW1

b1 = b1-learning_rate*db1

W2 = W2-learning_rate*dW2

 b2 = b2-learning_rate*db2
```

Deep Neural network:

```
# Functions:

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)

    """

    W1 = np.random.randn(n_h,n_x)*0.01

    b1 = np.zeros(shape=(n_h,1))

    W2 = np.random.randn(n_y,n_h)*0.01

    b2 = np.zeros(shape=(n_y,1))
```

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

```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in our network
    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                  Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                  bl -- bias vector of shape (layer_dims[l], 1)
    """
    parameters = {}
    L = len(layer_dims)           # number of layers in the network

    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
```

```
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 dictionary 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 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 dictionary 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 L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation
    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()
    Returns:
    AL -- last post-activation value
```

```
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of them, indexed from 0 to L-2)
                the cache of linear_sigmoid_forward() (there is one, indexed L-1)
    """
    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in the neural network
    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
       A, cache =
linear_activation_forward(A,parameters['W'+str(l)],parameters['b'+str(l)],activation="relu")
        caches.append(cache)
    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache =
linear_activation_forward(A,parameters['W'+str(L)],parameters['b'+str(L)],activation="sigmoid")
    caches.append(cache)
    assert(AL.shape == (1,X.shape[1]))
    return AL, caches
--
def compute_cost(AL, Y):
    """
    Implement the cost function
    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (1, number of examples)
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)
```

```
    Returns:
    cost -- cross-entropy cost
    """
    m = Y.shape[1]
    # Compute loss from aL and y.
    cost = -(1/m)*np.sum(np.multiply(Y,np.log(AL))+np.multiply(1-Y,np.log(1-AL)))
    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).
    assert(cost.shape == ())
    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,cache[0].T)/m

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

    dA_prev = np.dot(cache[1].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):

    """

    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:

    dA -- post-activation gradient for current layer l

    cache -- tuple of values (linear_cache, activation_cache) we store for computing backward
propagation efficiently

    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    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

    """

    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 L_model_backward(AL, Y, caches):

    """

    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR -> SIGMOID group

    Arguments:

    AL -- probability vector, output of the forward propagation (L_model_forward())

    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)

    caches -- list of caches containing:

                every cache of linear_activation_forward() with "relu" (it's caches[l], for l in range(L-1) i.e l =
0...L-2)

                the cache of linear_activation_forward() with "sigmoid" (it's caches[L-1])

    Returns:

    grads -- A dictionary with the gradients

             grads["dA" + str(l)] = ...

             grads["dW" + str(l)] = ...

             grads["db" + str(l)] = ...

    """

    grads = {}

    L = len(caches) # the number of layers

    m = AL.shape[1]

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

```
    # Initializing the backpropagation

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

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"],
grads["dWL"], grads["dbL"]

    current_cache = caches[L-1]

    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] =
linear_activation_backward(dAL,current_cache,activation="sigmoid")

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

        # lth layer: (RELU -> LINEAR) gradients.

        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs: "grads["dA" + str(l + 1)] , 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 + 2)],
current_cache, activation = "relu")

        grads["dA" + str(l + 1)] = 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):

    """

    Update parameters using gradient descent

    Arguments:

    parameters -- python dictionary containing your parameters

    grads -- python dictionary containing your gradients, output of L_model_backward
```

```
    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 parameters
--
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
print_cost=False):#lr was 0.009
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.
    Arguments:
    X -- data, numpy array of shape (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples)
    layers_dims -- list containing the input size and each layer size, of length (number of layers + 1).
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps
    Returns:
    parameters -- parameters learnt by the model. They can then be used to predict.
    """
```

```
 costs = []                         # keep track of cost
  # 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))
     if print_cost and i % 100 == 0:
        costs.append(cost)
    return parameters
```

--