Deep Neural Network for Image Classification: Application

When you finish this, you will have finished the last programming assignment of the course!

You will use use the functions you implemented in the previous assignment to build a deep network, and apply it to cat vs non-cat classification. Hopefully, your new feature vectors from the previous assignment will perform better than your logistic regression implementation.

After this assignment you will be able to:

- Build and apply a deep neural network to supervised learning.

Let's get started!

1 - Packages

Let's first import all the packages that you will need during this assignment.

- [numpy](www.numpy.org) is the fundamental package for scientific computing with Python.
- [matplotlib](http://matplotlib.org) is a library to plot graphs in Python.
- h5py is a common package to interact with a dataset that is stored on an H5 file.
- [PIL](http://www.pythonware.com/products/pil/) and [scipy](https://www.scipy.org/) are used here to test your model with your own picture at the end.
- dnn_app_utils provides the functions implemented in the "Building your Deep Neural Network: Step by Step" assignment in the previous week.
- np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

```
import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy.ndimage import imresize
from dnn_app_utils_v3 import *

%matplotlib inline
plt.rcParams['figure.figure-size'] = [5.0, 4.0] # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

2 - Dataset

You will use the same "Cat vs non-cat" dataset as in "Logistic Regression as a Neural Network" (Assignment 2). The model you had built had 70% test accuracy on classifying cats vs non-cats images. Hopefully, your new model will perform a better!

Problem Statement: You are given a dataset ("data.h5") containing:

- a training set of m_train images labelled as cat (1) or non-cat (0)
- a test set of m_test images labelled as cat and non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by running the cell below.

```
train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
```

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to see other images.

```
# Example of a picture
index = 7
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[:, index]) + ". It's a " + classes[np.squeeze(train_y[:, index])].decode("utf-8") + " picture.")
```

```
y = 1. It's a non-cat picture.
```

Many software bugs in deep learning come from having matrix/vector dimensions inconsistent. You can fix these bugs quickly if you spend enough time on dimensions.

```
# Explore your dataset
m_train = train_x_orig.shape[0]
num_px = train_x_orig.shape[1]
m_test = test_x_orig.shape[0]

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_x_orig shape: " + str(train_x_orig.shape))
print ("train_y shape: " + str(train_y.shape))
print ("test_x_orig shape: " + str(test_x_orig.shape))
print ("test_y shape: " + str(test_y.shape))
```

```
Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)
train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)
```

So, you now have access to 209 images in the form of an array of shape (64, 64, 3). Let's also flatten and standardize the images, as we did with the datasets in the previous week.

```
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T # The "-1" makes reshape flatten the remaining t
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T

# Standardize data to have feature values between 0 and 1.
train_x = train_x_flatten/255.
test_x = test_x_flatten/255.

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))

train_x's shape: (12288, 209)
test_x's shape: (12288, 50)
```

$12,288$ equals $64 \times 64 \times 3$ which is the size of one reshaped image vector.

3 - Architecture of your model

Now that you are familiar with the dataset, it is time to build a deep neural network to distinguish cat images from non-cat images.

You will build two different models:

- 2-layer neural network
- L-layer deep neural network

You will then compare the performance of these models, and also try out different values for L.

Let's look at the two architectures:

3.1 - 2-layer neural network

Figure 2: 2-layer neural network.
The model can be summarized as: "INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT".

Detailed Architecture of figure 2:

- The input is a (64,64,3) image which is flattened to a vector of size $(12288,1)$.
- The corresponding vector: $[x_0,x_1,...,x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(7,12288)$.
- You then add a bias term and take its ReLU to get the following vector: $[a_0^{[1]}, a_1^{[1]},...,a_{6}^{[1]}]^T$.
- You then repeat the same process.
- You multiply the resultant vector by $W^{[2]}$ and add your intercept (bias).
- Finally, you take the sigmoid of the result. If it is greater than 5.0, you classify it to be a cat.

3.2 - L-layer deep neural network

It is hard to represent an L-layer deep neural network with the above representation. However, here is a simplified network representation:

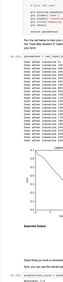


Figure 3: L-layer neural network.
The model can be summarized as: "[LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID".

Detailed Architecture of figure 3:

- The input is a (64,64,3) image which is flattened to a vector of size $(12288,1)$.
- The corresponding vector: $[x_0,x_1,...,x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then you add the intercept $b^{[1]}$. The result is called the linear unit.
- Next, you take the ReLU of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$ depending on the model architecture.
- Finally, you take the linear unit of the last layer $(W^{[L]}, b^{[L]})$. This last linear unit is then passed to the sigmoid.

3.3 - General methodology

As usual you will follow the Deep Learning methodology to build the model:

1. Initialize parameters / Define hyperparameters
2. Loop for num_iterations:
   a. Forward propagation
   b. Compute cost function
   c. Backward propagation
   d. Update parameters (using parameters, and gradients from backprop)
4. Use trained parameters to predict labels

Let's now implement those two models!

4 - Two-layer neural network

Question: Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_cost(AL, Y):
    ...
    return cost
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

```
# GRADED FUNCTION: two_layer_model

def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
    '''
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples)
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
    num_iterations -- number of iterations of optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- If set to True, this will print the cost every 100 iterations
    GRADIENT_CHECK -- true to run gradient check on 1 batch
    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    '''

    np.random.seed(1)
    grads = {}
    costs = []                              # keep track of the cost
    m = X.shape[1]                          # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'd previously implemented
    parameters = initialize_parameters(n_x, n_h, n_y)
    parameters = initialize_parameters(n_x,n_h,n_y)

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1". Outputs: "A1, cache1, A2, cach
        A1, cache1 = linear_activation_forward(X, W1, b1, 'relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, 'sigmoid')

        # Compute cost
        cost = compute_cost(A2, Y)

        # Initialize backward propagation
        dA2 = -(np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also add intermediate "cache
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, 'sigmoid')

        # Backward propagation: LINEAR -> RELU -> LINEAR
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, 'relu')

        # Set diffs
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

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

        # Retrieve W1, b1, W2, b2 from parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

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

    # plot the cost

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per 100)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

```
parameters, costs = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), num_iterations = 2500, print_cost =
```

```
Cost after iteration 0: 0.693049735659989
Cost after iteration 100: 0.646544137960671
Cost after iteration 200: 0.632548618211256
Cost after iteration 300: 0.609543111754701
Cost after iteration 400: 0.567747875645925
Cost after iteration 500: 0.526405091374345
Cost after iteration 600: 0.501404644289294
Cost after iteration 700: 0.473408660874418
Cost after iteration 800: 0.431779646468505
Cost after iteration 900: 0.379351580455294
Cost after iteration 1000: 0.342413800085322
Cost after iteration 1100: 0.305626692442097
Cost after iteration 1200: 0.268922094432392
Cost after iteration 1300: 0.239974754956447
Cost after iteration 1400: 0.228686906951741
Cost after iteration 1500: 0.214048036468649
Cost after iteration 1600: 0.207890766566901
Cost after iteration 1700: 0.201091618823036
Cost after iteration 1800: 0.195035316007193
Cost after iteration 1900: 0.190773584149167
Cost after iteration 2000: 0.186610954941285
Cost after iteration 2100: 0.182580876671048
Cost after iteration 2200: 0.178602945925657
Cost after iteration 2300: 0.175518638094391
Cost after iteration 2400: 0.172294761201948
```

Expected Output:

Cost after iteration 0	0.693049735659989
Cost after iteration 100	0.646544137960671
...	**...**
Cost after iteration 2400	0.172304941201948

Good! Now you built a two-layer implementation!Otherwise it might have failed. Fix bugs now to make this work.

Now, you can use the trained parameters to classify images from the dataset. To see your predictions on the training and test sets, run the cell below.

```
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 1.0
```

Expected Output:

```
Accuracy: 1.0
```

```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.72
```

Expected Output:

```
Accuracy: 0.72
```

Note: You may notice that running the model on fewer iterations (say 1500) gives better accuracy on the test set. This is called "early stopping" and we will talk about it in the next course. Early stopping is also a way to prevent overfitting.

Congratulations! It seems that your 2-layer neural network has better performance (72% than the logistic regression implementation (70%, assignment week 2). Let's see if you can do even better!

5 - L-layer Neural Network

Question: Use the helper functions you have implemented previously to build an L-layer neural network with the following structure: *[LINEAR -> RELU]$\times$(L-1) -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters_deep(layer_dims):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_cost(AL, Y):
    ...
    return cost
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

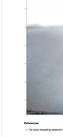
```
np.random.seed(1)
train_costs = []
parameters = initialize_parameters_deep(layers_dims)

for i in range(0, num_iterations):
    AL, caches = L_model_forward(X, parameters)
    cost = compute_cost(AL, Y)
    grads = L_model_backward(AL, caches, Y)
    parameters = update_parameters(parameters, grads, learning_rate)
    if print_cost and i % 100 == 0:
        print ("Cost after iteration %d: %f" %(i, cost))
    if print_cost and i % 100 == 0:
        costs.append(cost)

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

pred_train = predict(train_x, train_y, parameters)
pred_test = predict(test_x, test_y, parameters)
```

```
Cost after iteration 0: 0.771749
Cost after iteration 100: 0.672053
Cost after iteration 200: 0.648263
Cost after iteration 300: 0.611507
Cost after iteration 400: 0.567047
Cost after iteration 500: 0.540138
Cost after iteration 600: 0.527929
Cost after iteration 700: 0.465477
Cost after iteration 800: 0.369126
Cost after iteration 900: 0.391747
Cost after iteration 1000: 0.315187
Cost after iteration 1100: 0.272070
Cost after iteration 1200: 0.237824
Cost after iteration 1300: 0.199601
Cost after iteration 1400: 0.189541
Cost after iteration 1500: 0.161099
Cost after iteration 1600: 0.148213
Cost after iteration 1700: 0.137774
Cost after iteration 1800: 0.129740
Cost after iteration 1900: 0.121223
Cost after iteration 2000: 0.116062
Cost after iteration 2100: 0.111305
Cost after iteration 2200: 0.108210
Cost after iteration 2300: 0.104915
Cost after iteration 2400: 0.101070
```



Expected Output:

Cost after iteration 0	0.771749
Cost after iteration 100	0.672053
...	**...**
Cost after iteration 2400	0.101070

```
print("train accuracy: {}".+ str(np.sum((pred_train == train_y)/m)))
```

```
train accuracy: 0.9856459330143541
```

Expected Output:

```
"Train Accuracy": 0.9856459330143541
```

```
print("test accuracy: {}".+ str(np.sum((pred_test == test_y)/m)))
```

```
test accuracy: 0.8
```

Expected Output:

```
"Test Accuracy": 0.8
```

Congrats! It seems that your 2-layer neural network has better performance (80% than your 2-layer neural network (72%) on the same test set.

This is good performance for this task. Nice job!

Though in the next course on "Improving deep neural networks" you will learn how to obtain even higher accuracy by systematically searching for better hyperparameters (learning rate, layers, dim, num_iterations, and others).also in the next course you'll learn the latest techniques.

6) Results Analysis

First, take a look at some images the L-layer model labeled incorrectly. This will show a few mislabeled images.

```
print_mislabeled_images(classes, test_x, test_y, pred_test)
```

A few types of images the model tends to do poorly on include:

- Cat body in an unusual position
- Cat against a background of a similar color
- Dog picture with the face of a cat
- Great angle of a cat
- Image of a cat in a strange pose
- Scale (extreme cat is very large or small in image)

7) Test with your own image (optional/ungraded exercise)

Congratulations on finishing this assignment. You can use your own image and see the output of your model. To do that:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" folder
3. Change your image's name in the following code
4. Run the code and check if the algorithm is right (1 = cat, 0 = non-cat)!

```
## START CODE HERE ##
my_image = "my_image.jpg" # change this to the name of your image file
my_label_y = [1, 0] # the true class of your image (1 -> cat, 0 -> non-cat)
## END CODE HERE ##

fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px),interp='cubic').reshape((num_px*num_px*3,1))
my_predicted_image = predict(my_image, my_label_y, parameters)

plt.imshow(image)
print ("y = " + str(np.squeeze(my_predicted_image)) + ", your L-layer model predicts a " + classes[np.squeeze(my_predicted
```

```
y = 1.0, your L-layer model predicts a "cat" picture.
```


References:

- for data resampling several images: http://beyondvision.com/questions/38798/dimensionset-of-images-in-python