WINTER TRAINING REPORT

On

NEURAL NETWORKS AND DEEP LEARNING

Submitted to Guru Gobind Singh Indraprastha University, Delhi (India)
in partial fulfillment of the requirement for the award of the degree of

B.TECH

in

ELECTRONICS AND COMMUNICATIONS ENGINEERING

Submitted By
MAYANK CHOUDHARY

Roll. No. 35515002818

DEPTT. OF ELECTRONICS AND COMMUNICATIONS

MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY ,
NEW DELHI-110058

FEBRUARY 2021

ACKNOWLEDGEMENT

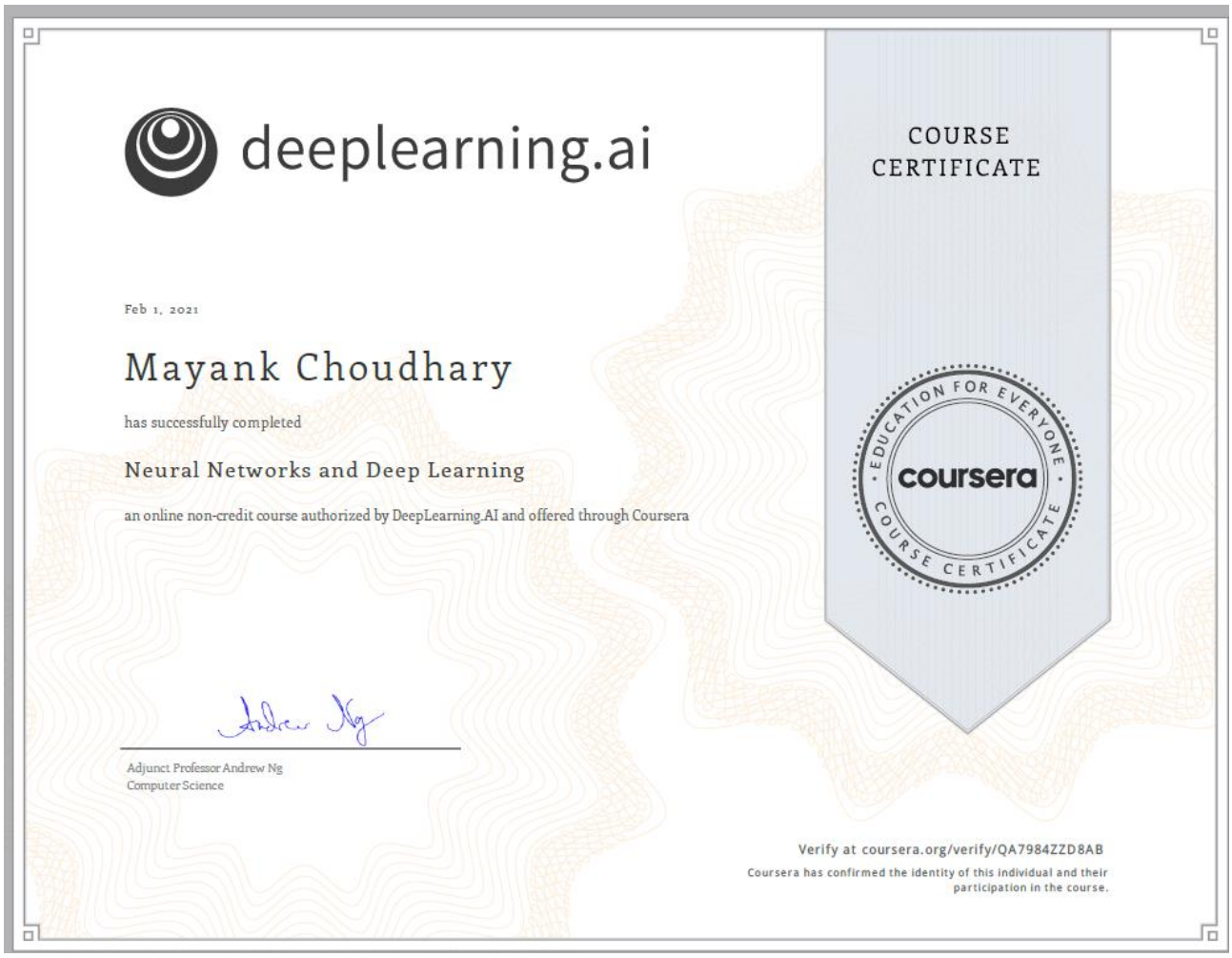
A research work owes its success from commencement to completion, to the people in love with researchers at various stages. Let me in this page express my gratitude to all those who helped us in various stage of this study. First, I would like to express my sincere gratitude indebtedness to **Mr. PUNEET AZAD** (HOD, Department of Information Technology, Maharaja Surajmal Institute of Technology, New Delhi) for allowing me to undergo the summer training of 30 days at …….....**COURSERA**.....

I am grateful to my instructors **Andrew Ng** for this easy to learn course and their way of teaching marks a really strong effect.

Last but not least, I pay my sincere thanks and gratitude to all the Staff Members of Maharaja Surajmal Institute of Technology, New Delhi for their support and for making our training valuable and fruitful.

Submitted By: Mayank Choudhary

CERTIFICATE



CANDIDATE'S DECLARATION

I **MAYANK CHOUDHARY** , Roll No **35515002818**, B.Tech (Semester- 6th) of the Maharaja Surajmal Institute of Technology, New Delhi hereby declare that the Training Report entitled **" NEURAL NETWORKS AND DEEP LEARNING"** is an original work and data provided in the study is authentic to the best of my knowledge. This report has not been submitted to any other Institute for the award of any other degree.

Name of Student : Mayank Choudhary
(Roll No: 35515002818**)**

Organization Introduction

Coursera is a world-wide online learning platform founded in 2012 by Stanford University's computer science professors Andrew Ng and Daphne Koller that offers massive open online courses, specializations, degrees, professional and mastertrack courses.

The course is offered by DeepLearning.AI

Instructor -

Taught by: Andrew Ng, Instructor

Founder, DeepLearning.AI & Co-founder, Coursera

List of Figures

	Page number
Fig.1 Deep neural network	8
Fig.2 Different types of NN	9
Fig.3 Artificial neural network	9
Fig.4 Activation function	10
Fig.5 Recurrent neural network	11
Fig.6 Convolution neural network	11
Fig.7 Logistic regression	12
Fig.8 Gradient descent	12
Fig.9 Vectorization	13
Fig.10 Not a cat image	15
Fig.11 Image to vector conversion	16
Fig.12 2 Layer neural network	17
Fig.13 L Layer neural network	18
Fig.14 2 Layer neural network(learning rate vs iteration)	22
Fig.15 L Layer neural network(learning rate vs iteration)	26
Fig.16 Images	27
Fig.17 Cat image	28

Contents

1. Introduction to Deep Learning

1.1 What is neural network?
1.2 Supervised Learning with Neural Networks

2. Logistic Regression as a Neural Network

2.1 Logistic regression and gradient descent
2.2 Python and Vectorization

3. Shallow Neural Network

3.1 Activation functions
3.2 Back propagation intuition

4. Deep Neural Network

4.1 Deep L-layer neural network
4.2 Forward Propagation in Deep network
4.3 Forward and backward propagation
4.4 Parameters vs Hyperparameters
4.5 Programming assignment of deep neural network application

References

- Coursera online training course
- Google Images

What are neural networks?

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.

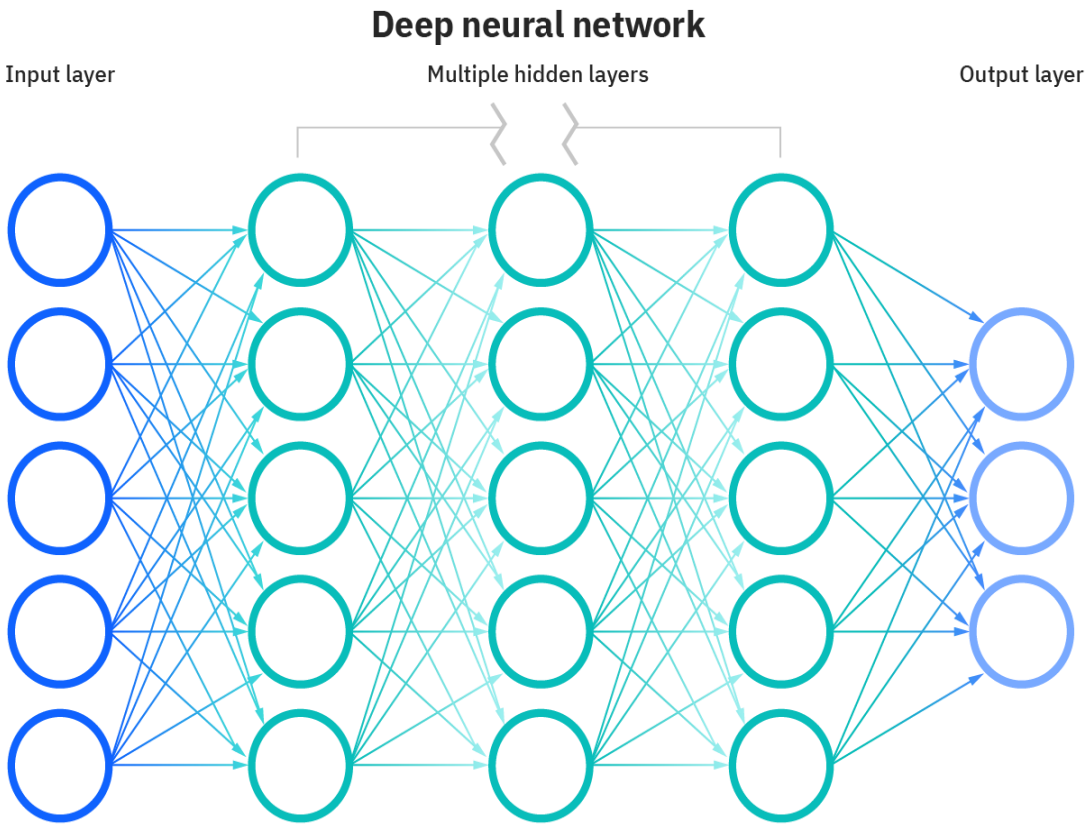


Fig.1

Different types of Neural Networks in Deep Learning

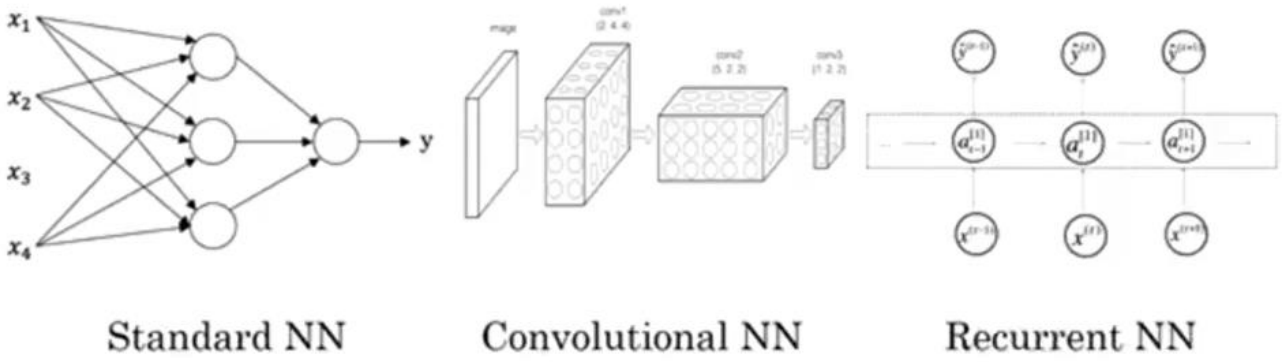
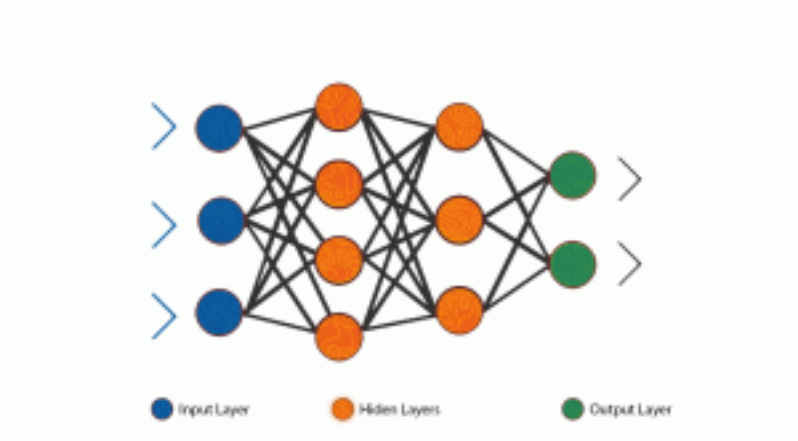


Fig.2

Standard/Artificial Neural Network (ANN)

A single perceptron (or neuron) can be imagined as a Logistic Regression. Artificial Neural Network, or ANN, is a group of multiple perceptrons/ neurons at each layer. ANN is also known as a **Feed-Forward Neural network** because inputs are processed only in the forward direction:



ANN

Fig.3

Artificial Neural Network is capable of learning any nonlinear function. Hence, these networks are popularly known as **Universal Function Approximators**. ANNs have the capacity to learn weights that map any input to the output.

One of the main reasons behind universal approximation is the **activation function**. Activation functions introduce nonlinear properties to the network. This helps the network learn any complex relationship between input and output.

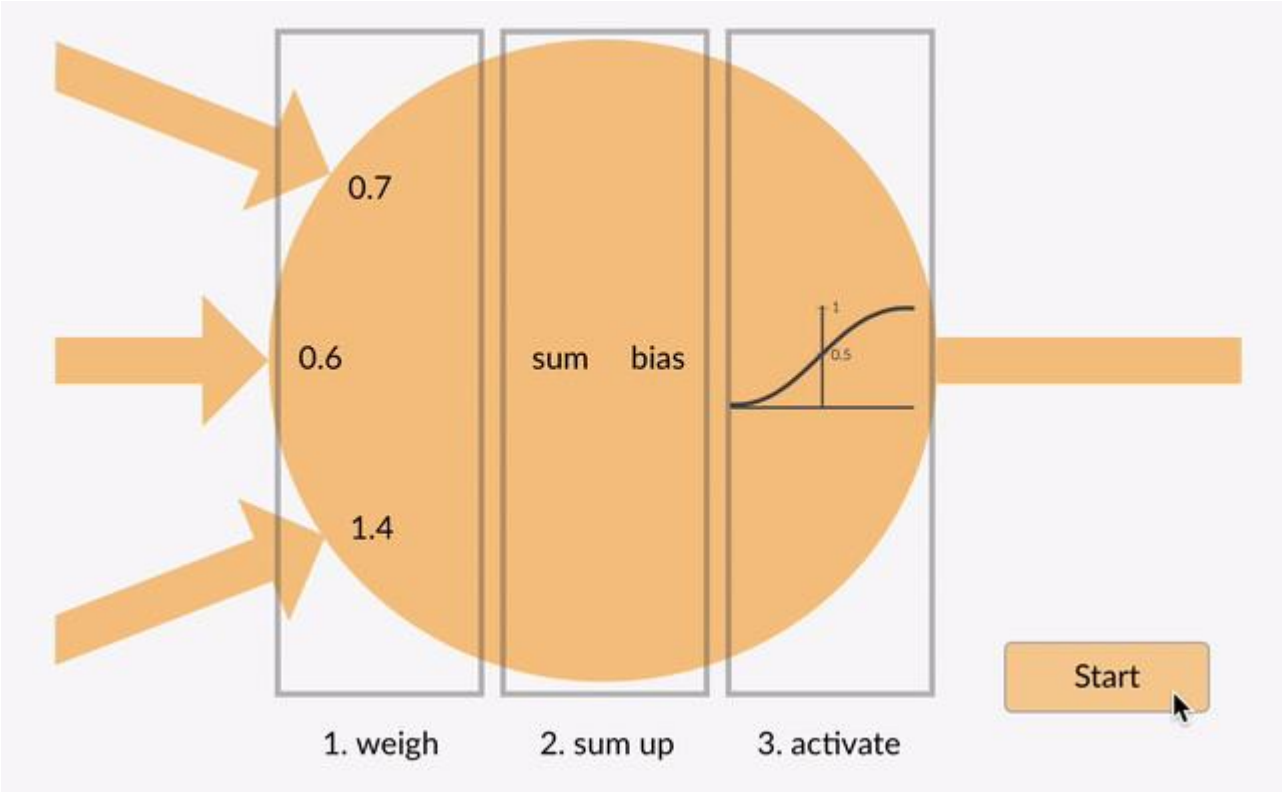


Fig.4

Recurrent Neural Network (RNN)

A looping constraint on the hidden layer of ANN turns to RNN.

- RNN captures the sequential information present in the input data i.e. dependency between the words in the text while making predictions:

- RNNs share the parameters across different time steps. This is popularly known as **Parameter Sharing**. This results in fewer parameters to train and decreases the computational cost

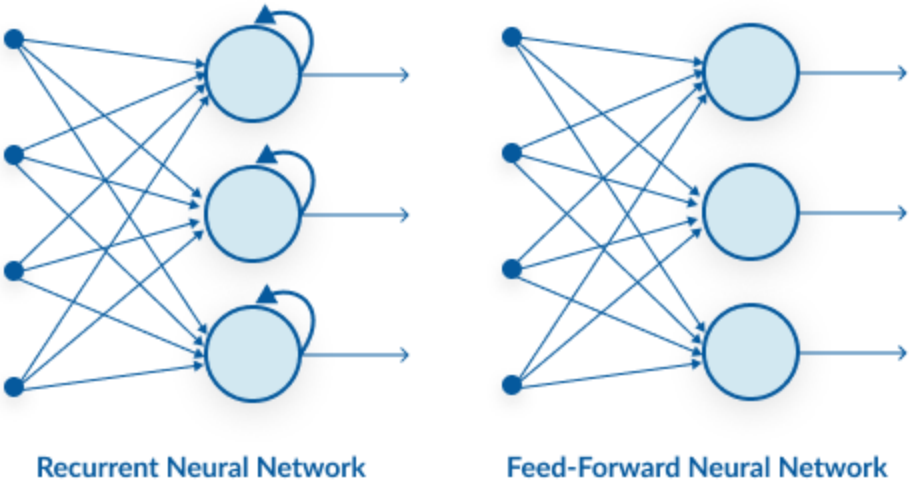


Fig.5

Convolution Neural Network (CNN)

Convolutional neural networks (CNN) are all the rage in the deep learning community right now. These CNN models are being used across different applications and domains, and they're especially prevalent in image and video processing projects.

- CNN learns the filters automatically without mentioning it explicitly. These filters help in extracting the right and relevant features from the input data.
- CNN captures the **spatial features** from an image. Spatial features refer to the arrangement of pixels and the relationship between them in an image. They help us in identifying the object accurately, the location of an object, as well as its relation with other objects in an image.

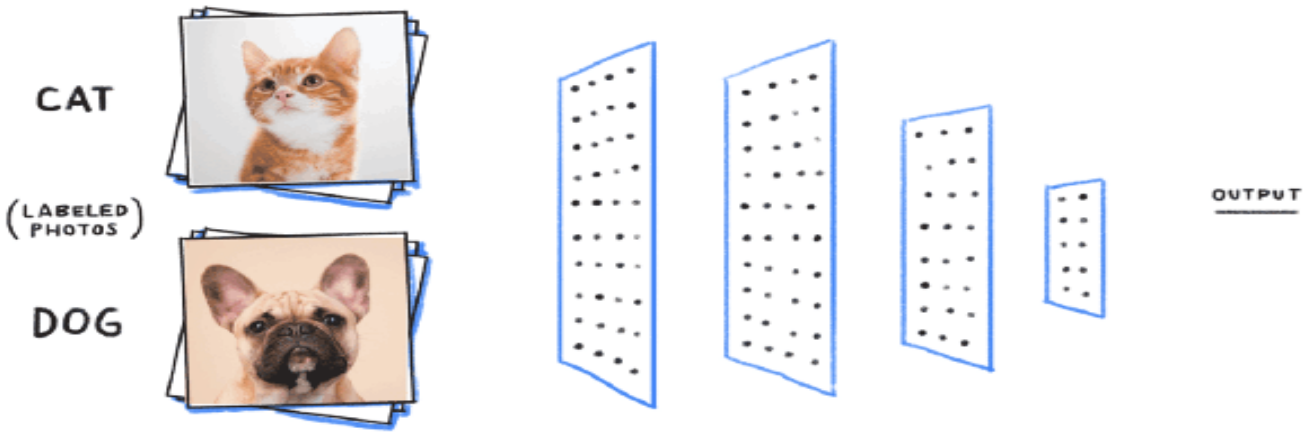


Fig.6

Logistic Regression as a Neural Network

Logistic regression is a binary classification method. It can be modelled as a function that can take in any number of inputs and constrain the output to be between 0 and 1. This means, we can think of Logistic Regression as a one-layer neural network. For a binary output, if the true label is y (y = 0 or y = 1) and y_hat is the predicted output – then y_hat represents the probability that y = 1 - given inputs w and x. Therefore, the probability that y = 0 given inputs w and x is (1 - y_hat), as shown below.

$$P\,(y = 1|\boldsymbol{w}, \boldsymbol{x}) = \hat{y}$$
$$P\,(y = 0|\boldsymbol{w}, \boldsymbol{x}) = 1 - \hat{y}$$

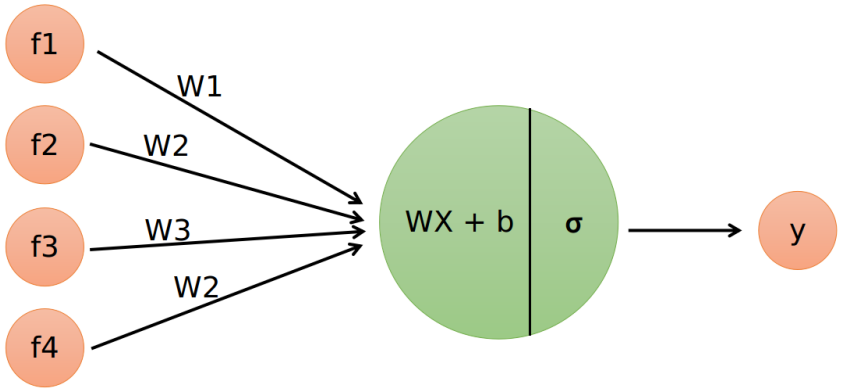


Fig.7

What is a Gradient?

A gradient measures how much the output of a function changes if you change the inputs a little bit

It simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. Said it more mathematically, a gradient is a partial derivative with respect to its inputs.

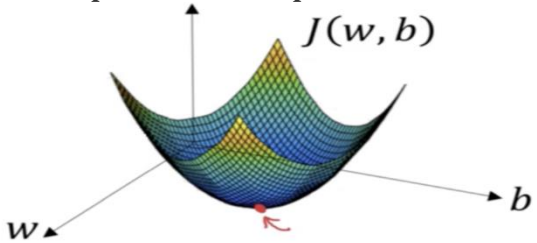


Fig.8

What is Vectorization ?

Vectorization is used to speed up the Python code without using loop. Using such a function can help in minimizing the running time of code efficiently. Various operations are being performed over vector such as *dot product of vectors* which is also known as *scalar product* as it produces single output, outer products which results in square matrix of dimension equal to length X length of the vectors, *Element wise multiplication* which products the element of same indexes and dimension of the matrix remain unchanged.

$$a \cdot b = \underset{(1 \times n)}{\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 \end{bmatrix}} \underset{(n \times 1)}{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}} = \left\{ a_1b_1 + a_2b_2 + a_3b_3 + a_4b_4 + a_5b_5 \right\}$$

Dot Product

Fig.9

Deep Neural Network for Image Classification: Application

When you finish this, you will have finished the last programming assignment of Week 4, and also the last programming assignment of this course!

You will use the functions you'd implemented in the previous assignment to build a deep network, and apply it to cat vs non-cat classification. Hopefully, you will see an improvement in accuracy relative to your previous 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 is the fundamental package for scientific computing with Python.
- matplotlib 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 and scipy 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 to this notebook.
- np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

In [1]:

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

%matplotlib inline
plt.rcParams['figure.figsize'] = (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 ch
    annels (RGB).
```

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

In [2]:

```
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.

In [3]:

```
# Example of a picture
index = 10
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[0,index]) + ". It's a " + classes[train_y[0,index]].decode("utf-8") +  " picture.")
```

```
y = 0. It's a non-cat picture.
```

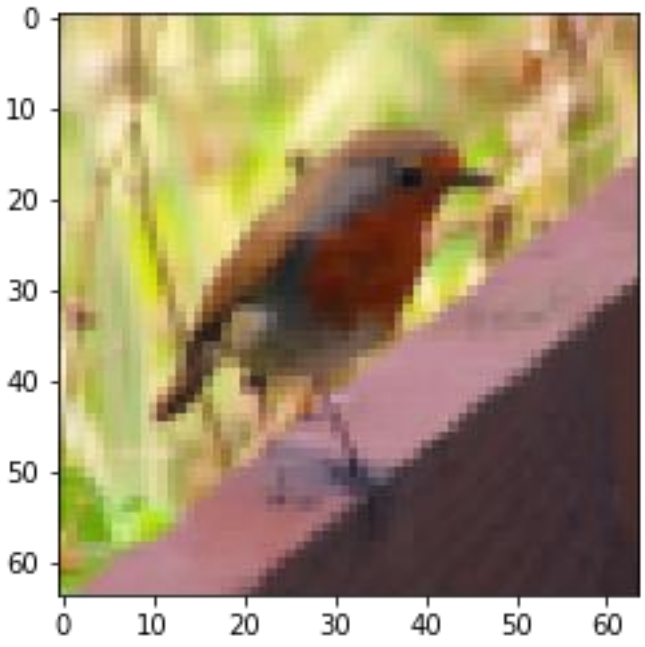


Fig.10

In [4]:

```
# 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)
```

As usual, you reshape and standardize the images before feeding them to the network. The code is given in the cell below.

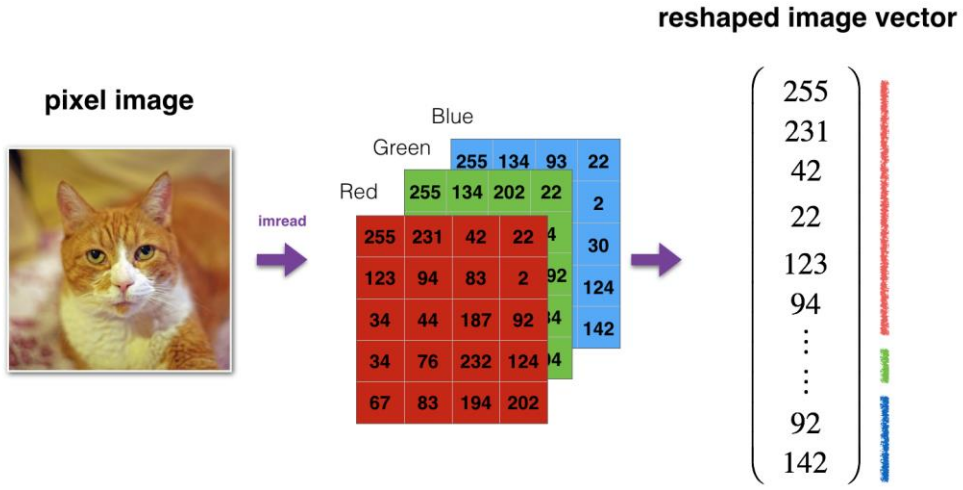


Fig.10: Image to vector conversion.

```
In [5]:
# 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 di
mensions
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)
```

12,28812,288 equals 64×64×364×64×3 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:

- A 2-layer neural network
- An 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

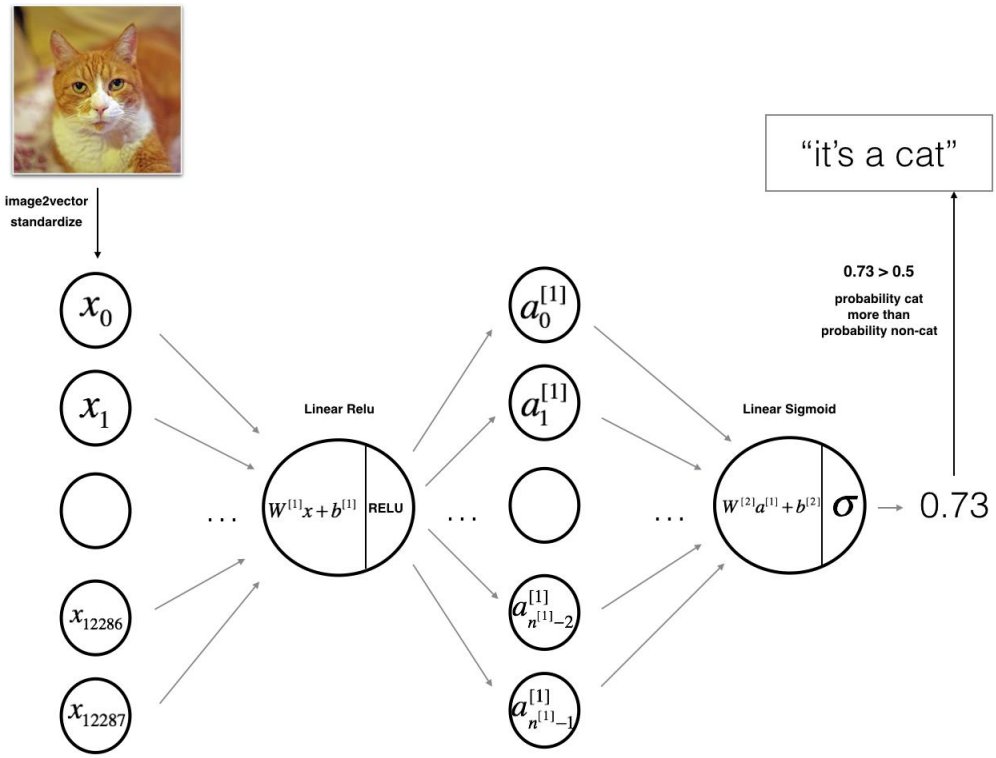


Fig.12: 2-layer neural network.

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 $(12288,1)$(12288,1).
- The corresponding vector: $[x_0,x_1,...,x_{12287}]^T$[x0,x1,...,x12287]T is then multiplied by the weight matrix $W^{[1]}$W[1] of size $(n^{[1]},12288)$(n[1],12288).
- You then add a bias term and take its relu to get the following vector: $[a^{[1]}_0,a^{[1]}_1,...,a^{[1]}_{n^{[1]}-1}]^T$[a0[1],a1[1],...,an[1]−1[1]]T.
- You then repeat the same process.

- You multiply the resulting vector by $W^{[2]}$W[2] and add your intercept (bias).
- Finally, you take the sigmoid of the result. If it is greater than 0.5, 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:

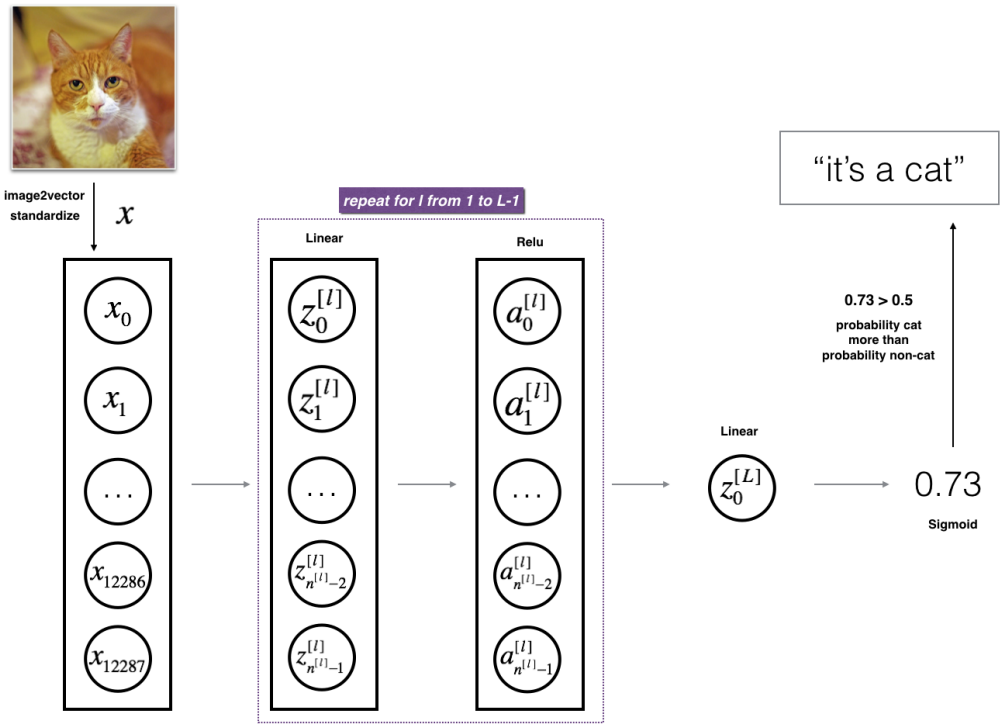


Fig.13: L-layer neural network.

The model can be summarized as: ***[LINEAR -> RELU]*** ×× ***(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: $[x_0,x_1,...,x_{12287}]^T$[x0,x1,...,x12287]T is then multiplied by the weight matrix $W^{[1]}$W[1] and then you add the intercept $b^{[1]}$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 $(W^{[l]},b^{[l]})$(W[l],b[l]) depending on the model architecture.
- Finally, you take the sigmoid of the final linear unit. If it is greater than 0.5, you classify it to be a cat.

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 grads 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
```

In [6]:

```
### CONSTANTS DEFINING THE MODEL ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

In [7]:

```
# 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 1 if cat, 0 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 the 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

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                              # to 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
    ### START CODE HERE ### (≈ 1 line of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    ### END CODE HERE ###

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

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1,
A2, cache2".
        ### START CODE HERE ### (≈ 2 lines of code)
        A1, cache1 = linear_activation_forward(X, W1, b1, 'relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, 'sigmoid')
        ### END CODE HERE ###

        # Compute cost
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(A2, Y)
        ### END CODE HERE ###

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

```
        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0 (not used), dW1, db1"
.
        ### START CODE HERE ### (≈ 2 lines of code)
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, 'sigmoid')
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, 'relu')
        ### END CODE HERE ###

        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Update parameters.
        ### START CODE HERE ### (approx. 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###

        # 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 {}: {}".format(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 hundreds)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

Run the cell below to train your parameters. See if your model runs. The cost should be decreasing. It may take up to 5 minutes to run 2500 iterations. Check if the "Cost after iteration 0" matches the expected output below, if not click on the square (■) on the upper bar of the notebook to stop the cell and try to find your error.

```
In [8]:
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), num_iterations = 2500, print_cost=True)
Cost after iteration 0: 0.6930497356599888
```

```
Cost after iteration 100: 0.6464320953428849
Cost after iteration 200: 0.6325140647912677
Cost after iteration 300: 0.6015024920354665
Cost after iteration 400: 0.5601966311605747
Cost after iteration 500: 0.5158304772764729
Cost after iteration 600: 0.47549013139433255
Cost after iteration 700: 0.43391631512257495
Cost after iteration 800: 0.400797753620389
Cost after iteration 900: 0.3580705011323798
Cost after iteration 1000: 0.3394281538366411
Cost after iteration 1100: 0.3052753636196264
Cost after iteration 1200: 0.2749137728213018
Cost after iteration 1300: 0.24681768210614854
Cost after iteration 1400: 0.19850735037466094
Cost after iteration 1500: 0.17448318112556666
Cost after iteration 1600: 0.17080762978096128
Cost after iteration 1700: 0.11306524562164724
Cost after iteration 1800: 0.09629426845937152
Cost after iteration 1900: 0.08342617959726856
Cost after iteration 2000: 0.07439078704319078
Cost after iteration 2100: 0.06630748132267927
Cost after iteration 2200: 0.05919329501038164
Cost after iteration 2300: 0.05336140348560553
Cost after iteration 2400: 0.048554785628770115
```

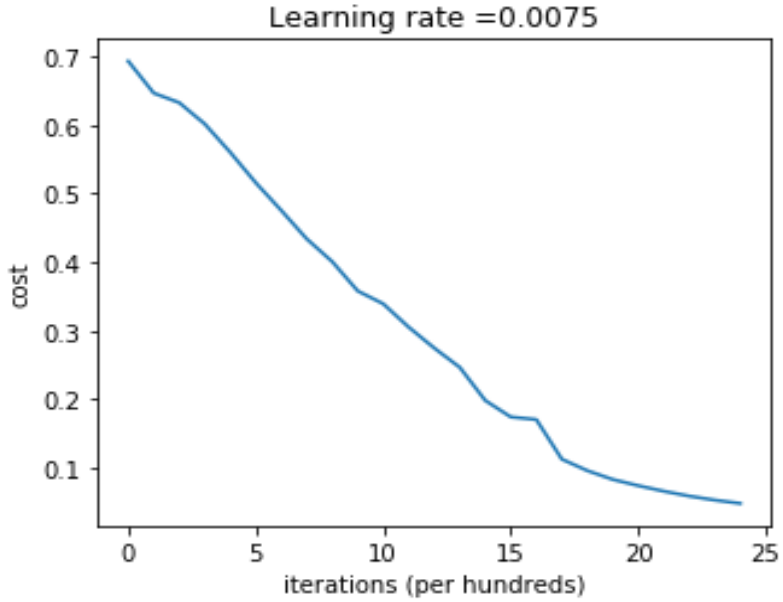


Fig.14: 2 layer neural network learning rate vs iterations

Expected Output:

Cost after iteration 0	0.6930497356599888
Cost after iteration 100	0.6464320953428849

...	...

Cost after iteration 2400	0.048554785628770226

Good thing you built a vectorized implementation! Otherwise it might have taken 10 times longer to train this.

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.

In [9]:

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

```
Accuracy: 1.0
```

Expected Output:

Accuracy	1.0

In [10]:

```
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 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 with an L-layer model.

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]××(L-1) -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:**def** initialize_parameters_deep(layers_dims):

```
        ...

        return parameters

    def L_model_forward(X, parameters):

        ...

        return AL, caches

    def compute_cost(AL, Y):

        ...
```

```
        return cost

    def L_model_backward(AL, Y, caches):

        ...

        return grads

    def update_parameters(parameters, grads, learning_rate):

        ...

        return parameters
```

In [11]:

```
### CONSTANTS ###
layers_dims = [12288, 20, 7, 5, 1] #  4-layer model
```

In [12]:

```
# GRADED FUNCTION: L_layer_model

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 (num_px * num_px * 3, number of examples)
    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.
    """

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization. (≈ 1 line of code)
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

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

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        ### START CODE HERE ### (≈ 1 line of code)
        AL, caches = L_model_forward(X, parameters)
        ### END CODE HERE ###
```

```
        # Compute cost.
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ### (≈ 1 line of code)
        grads =  L_model_backward(AL, Y, caches)
        ### END CODE HERE ###

        # Update parameters.
        ### START CODE HERE ### (≈ 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###

        # 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)

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

    return parameters
```

You will now train the model as a 4-layer neural network.

Run the cell below to train your model. The cost should decrease on every iteration. It may take up to 5 minutes to run 2500 iterations. Check if the "Cost after iteration 0" matches the expected output below, if not click on the square (■) on the upper bar of the notebook to stop the cell and try to find your error.

```
In [13]:
parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations = 2500, print_cost = True)
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: 0.527930
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: 0.272700
Cost after iteration 1200: 0.237419
Cost after iteration 1300: 0.199601
Cost after iteration 1400: 0.189263
Cost after iteration 1500: 0.161189
Cost after iteration 1600: 0.148214
Cost after iteration 1700: 0.137775
Cost after iteration 1800: 0.129740
Cost after iteration 1900: 0.121225
Cost after iteration 2000: 0.113821
Cost after iteration 2100: 0.107839
Cost after iteration 2200: 0.102855
Cost after iteration 2300: 0.100897
Cost after iteration 2400: 0.092878
```

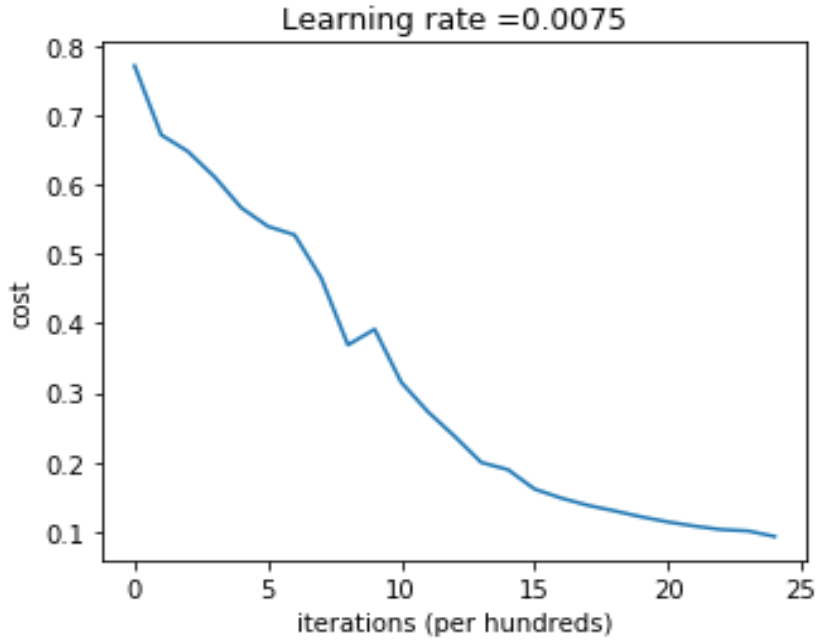


Fig.15: L layer neural network learning rate vs iterations

Expected Output:

Cost after iteration 0	0.771749
Cost after iteration 100	0.672053
...	...
Cost after iteration 2400	0.092878

In [14]:

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

```
Accuracy: 0.985645933014
```

Train Accuracy	0.985645933014

```
In [15]:
```
```
pred_test = predict(test_x, test_y, parameters)
```
```
Accuracy: 0.8
```

Expected Output:

Test Accuracy	0.8

Congrats! It seems that your 4-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_dims, num_iterations, and others you'll also learn in the next course).

6) Results Analysis

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

```
In [16]:
```
```
print_mislabeled_images(classes, test_x, test_y, pred_test)
```

Fig.16 mislabeled images

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

- Cat body in an unusual position
- Cat appears against a background of a similar color
- Unusual cat color and species
- Camera Angle
- Brightness of the picture
- Scale variation (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 "Op
en" to go on your Coursera Hub.

2. Add your image to this Jupyter Notebook's directory, in the "imag
es" 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 = no
    n-cat)!
```

```
In [17]:
```

```
## START CODE HERE ##
my_image = "my_image.jpg" # change this to the name of your image file
my_label_y = [1] # 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)).reshape((num_px*num_px*3,1))
my_image = my_image/255.
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[int(np.squeeze(my
_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
Accuracy: 1.0

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

Fig.17 cat image

References

The complete training report is based on the online training course from Coursera Machine Learning Course……..https://www.coursera.org

Thankyou