```
Programming Exercise 4: Logistic Regression (with tensorflow)
=============================================================
```

```python
# Run this cell to import all the libraries you need for this exercise
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import random
```

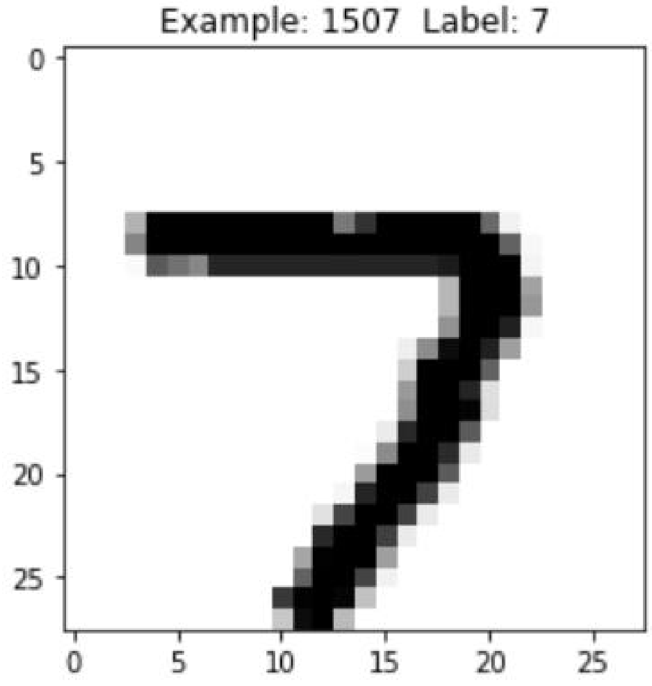
```python
# STEP 1: Read in data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

```python
def TRAIN_SIZE(num):
    print ('Total Training Images in Dataset = ' + str(mnist.train.images.shape))
    print ('--------------------------------------------------')
    x_train = mnist.train.images[:num,:]
    print ('x_train Examples Loaded = ' + str(x_train.shape))
    y_train = mnist.train.labels[:num,:]
    print ('y_train Examples Loaded = ' + str(y_train.shape))
    print('')
    return x_train, y_train

def TEST_SIZE(num):
    print ('Total Test Examples in Dataset = ' + str(mnist.test.images.shape))
    print ('--------------------------------------------------')
    x_test = mnist.test.images[:num,:]
    print ('x_test Examples Loaded = ' + str(x_test.shape))
    y_test = mnist.test.labels[:num,:]
    print ('y_test Examples Loaded = ' + str(y_test.shape))
    return x_test, y_test

def display_digit(num,x_train,y_train):
    print(y_train[num])
    label = y_train[num].argmax(axis=0)
    image = x_train[num].reshape([28,28])
    plt.title('Example: %d  Label: %d' % (num, label))
    plt.imshow(image, cmap=plt.get_cmap('gray_r'))
    plt.show()
```

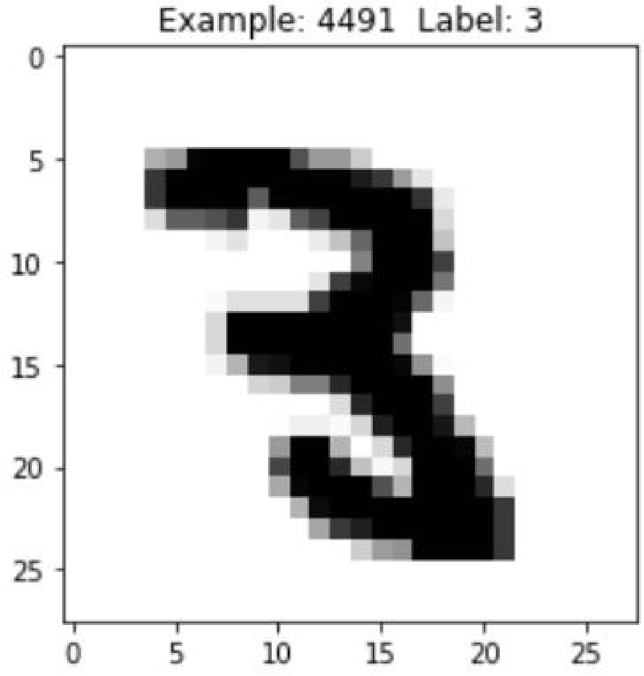
```
Total Training Images in Dataset = (55000, 784)
--------------------------------------------------
x_train Examples Loaded = (5500, 784)
y_train Examples Loaded = (5500, 10)

Total Test Examples in Dataset = (10000, 784)
--------------------------------------------------
x_test Examples Loaded = (1000, 784)
y_test Examples Loaded = (1000, 10)
```

```
# Display 30 random digits
for i in range(3):
    selected = random.randint(0,5500-1)
    display_digit(selected, X_train, Y_train)
```

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```




```
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```



```
# n features, k classes
n = 784
k = 10

# INSTRUCTIONS: Complete the code for Steps 3-7

# STEP 3: Create placeholders for features and labels
# ======== YOUR CODE ==========
#


# STEP 4: Create weights and bias
# ======== YOUR CODE ==========
#

# STEP 5: Predict Y from X and W, b
# ======== YOUR CODE ==========
#


# STEP 6: Define loss function
# ======== YOUR CODE ==========
#

# STEP 7: Gradient Descent
# ======== YOUR CODE ==========
#

# STEP 8: Calculate accuracy
```

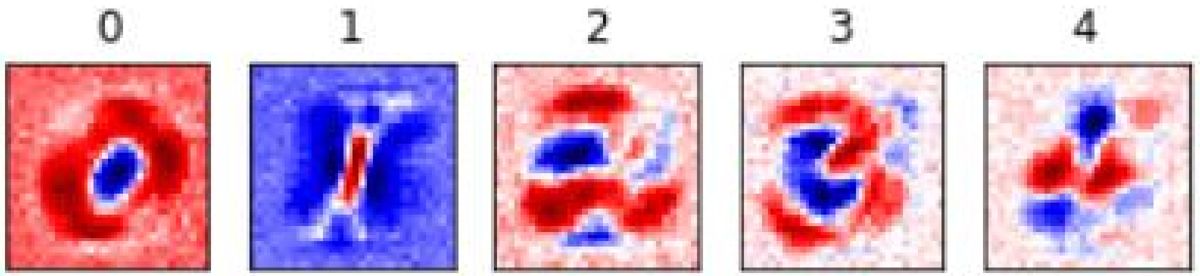
$$P(Y = i|x, W, b) = softmax_i(Wx + b) = \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}$$

$$y_{pred} = \mathrm{argmax}_i P(Y = i|x, W, b)$$

GradientDescentOptimizer

```python
# Initializing the variables
init = tf.global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Train model
    for i in range(train_steps+1):
        sess.run([optimizer, loss], feed_dict={X: X_train,Y: Y_train})
        if i%100 == 0:
            print('Training Step: ', i)
            print('Accuracy:', sess.run(accuracy, feed_dict={X: X_test

    # Obtain weights
    W_values = sess.run(w)
```

```
Training Step:  1300
Accuracy: [ 0.851 85.1  ]
Training Step:  1400
Accuracy: [ 0.855 85.5  ]
Training Step:  1500
Accuracy: [ 0.855 85.5  ]
Training Step:  1600
Accuracy: [ 0.855 85.5  ]
Training Step:  1700
Accuracy: [ 0.856 85.6  ]
Training Step:  1800
Accuracy: [ 0.857 85.7  ]
Training Step:  1900
Accuracy: [ 0.858 85.8  ]
Training Step:  2000
Accuracy: [ 0.859 85.9  ]
Training Step:  2100
Accuracy: [ 0.86 86.  ]
Training Step:  2200
Accuracy: [ 0.86 86.  ]
Training Step:  2300
Accuracy: [ 0.863 86.3  ]
Training Step:  2400
Accuracy: [ 0.864 86.4  ]
Training Step:  2500
Accuracy: [ 0.865 86.5  ]
```



EXERCISE 5

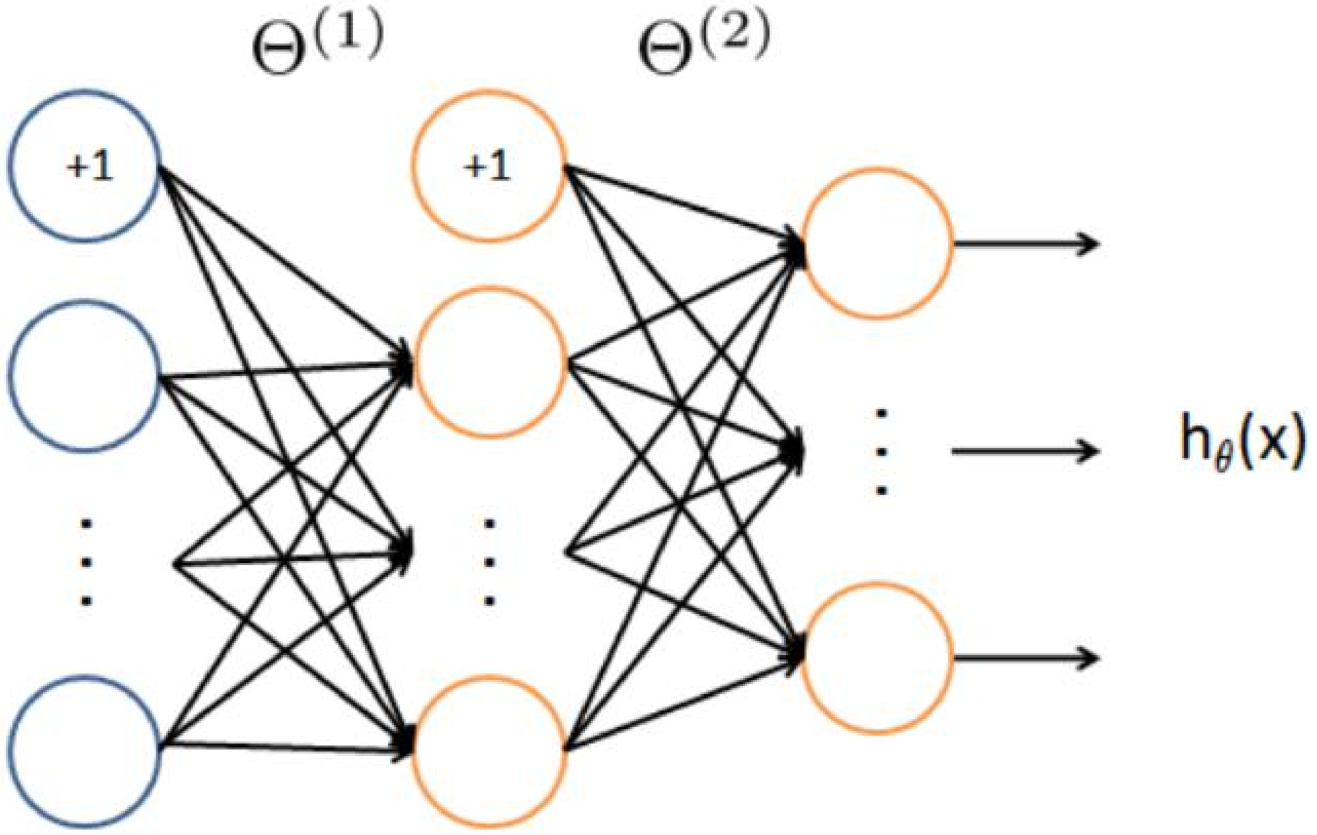
```
Programming Exercise 5: Neural Networks (without tensorflow)
============================================================

In this exercise, we are going to build a neural network with
1) an input layer matching the size of our instance data (400 + the bias unit),
2) a hidden layer with 25 units (26 with the bias unit), and
3) an output layer with 10 units corresponding to our one-hot encoding for the class labels.
```

```python
# Run this cell to import all the libraries you need for this exercise
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
```

```python
# STEP 1: Read in data
data = loadmat('ex5data1.mat')
```

```python
# Assuming m examples...
# X should be a (m x 400)-matrix, each row containing the 400 features of an example
# and y should be a (m x 1)-matrix containing the labels associated with the m examples

X = data['X']
y = data['y']
print(X.shape, y.shape)
```

```
(5000, 400) (5000, 1)
```



$a^{(1)} = x$ (add $a_0^{(1)}$)

$z^{(2)} = \Theta^{(1)} a^{(1)}$

$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

$z^{(3)} = \Theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)}) = h_\theta(x)$

Input Layer Hidden Layer Output Layer

```python
# standard sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Forward propagation
# Assuming:
#   input_size = 400  (number of neurons in input layer, also number of features for each example)
#   hidden_size = 25  (number of neurons in hidden layer)
#   num_labels = 10   (number of neurons in output layer)
# Inputs of the function:
#   X       (m x 400)-matrix   containing the input examples
#   theta1  (25 x 401)-matrix  containing the weights between the input layer of 401 neurons and the hidden lay
#   theta2  (10 x 26)-matrix   containing the weights between the hidden layer of 25 neurons and the output lay
# Outputs of the function
#   a1      (m x 401)-matrix   essentially X with an additional column of 1s inserted as column 0
#   z2      (m x 25)-matrix    contains a1 and theta1 multiplied together
#   a2      (m x 26)-matrix    apply sigmoid function to z2 with an additional column of 1s inserted as column
#   z3      (m x 10)-matrix    contains a2 and theta2 multiplied together
#   h       (m x 10)-matrix    contains the result of applying sigmoid to z3
def forward_propagate(X, theta1, theta2):
    m = X.shape[0]
    a1 =      ### FILL-IN
    z2 =    ### FILL-IN
    a2 =    ### FILL-IN
    z3 =    ### FILL-IN
    h =     ### FILL-IN
    return a1, z2, a2, z3, h
```

```python
# Cost function
def cost(params, input_size, hidden_size, num_labels, X, y, learning_rate):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    # reshape the parameter array into parameter matrices for each layer
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size + 1))))

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # compute the cost
    J = 0
    for i in range(m):
        first_term = np.multiply(-y[i,:], np.log(h[i,:]))
        second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
        J += np.sum(first_term - second_term)

    J = J / m

    # Add regularization term
    J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:], 2)) + np.sum(np.power(theta2[:,1:], 2)))

    return J
```

```python
# initial setup
input_size = 400
hidden_size = 25
num_labels = 10
learning_rate = 1

# randomly initialize a parameter array of the size of the full network's parameters
params = (np.random.random(size=hidden_size * (input_size + 1) + num_labels * (hidden_size + 1)) - 0.5) * 0.25

m = X.shape[0]
X = np.matrix(X)
y = np.matrix(y)

# unravel the parameter array into parameter matrices for each layer
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1))))
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size + 1))))
print(theta1.shape, '', theta2.shape)
```

```
(25, 401)  (10, 26)
```

```python
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
print(a1.shape, z2.shape, a2.shape, z3.shape, h.shape)
```

```
(5000, 401) (5000, 25) (5000, 26) (5000, 10) (5000, 10)
```

```python
print(cost(params, input_size, hidden_size, num_labels, X, y_onehot, learning_rate))
```

```
7.242497933475441
```

```python
##### end of cost function logic, below is the new part #####

# backpropagation
# taking the examples one at a time
for t in range(m):
    a1t = a1[t,:]  # (1, 401)
    z2t = z2[t,:]  # (1, 25)
    a2t = a2[t,:]  # (1, 26)
    ht = h[t,:]  # (1, 10)
    yt = y[t,:]  # (1, 10)

    d3t = ht - yt  # (1, 10)

    z2t = np.insert(z2t, 0, values=np.ones(1))  # (1, 26)

    #     d2t  (1 x 26)-matrix  containing deltas of the hidden layer
          ### FILL-IN

    delta1 = delta1 + (d2t[:,1:]).T * a1t     # (25, 401)
    delta2 = delta2 + d3t.T * a2t              # (10, 26)

delta1 = delta1 / m
delta2 = delta2 / m

# add the gradient regularization term
delta1[:,1:] = delta1[:,1:] + (theta1[:,1:] * learning_rate) / m
delta2[:,1:] = delta2[:,1:] + (theta2[:,1:] * learning_rate) / m
```

$$\delta_j^{(3)} = a_j^{(3)} - y_j$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

```
correct = [1 if a == b else 0 for (a, b) in zip(y_pred, y)]
accuracy = (sum(map(int, correct)) / float(len(correct)))
print('accuracy = ', accuracy*100,'%')
```

```
accuracy =  98.98 %
```

EXERCISE 6

```python
# Run this cell to import all the libraries you need for this exercise
import numpy as np
import tensorflow as tf
from scipy.io import loadmat
```

```python
# Read in data
data = loadmat('ex5data1.mat')
```

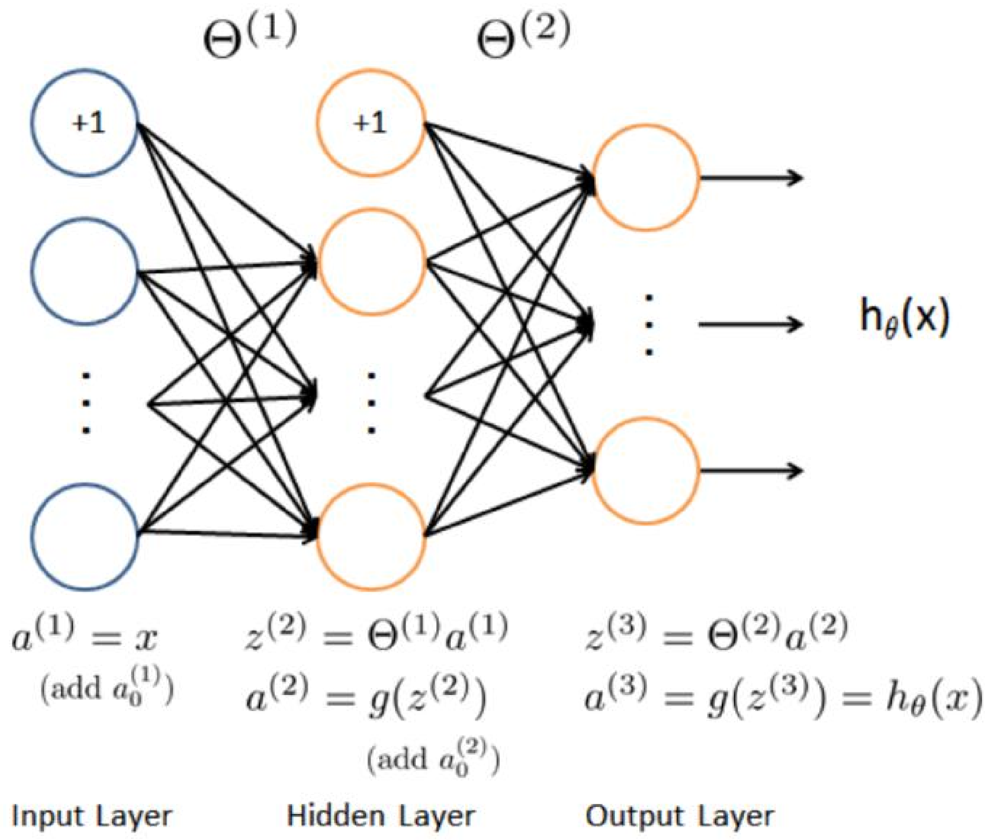
```python
X_data = data['X']
y_data = data['y']
print(X_data.shape, y_data.shape)
```

```
(5000, 400) (5000, 1)
```

```python
# Turn y into a one-hot vector format
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)
y_onehot = encoder.fit_transform(y_data)

# print one of the y[0] out
print(y_data[0], y_onehot[0,:])
```

```
[10] [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

```
# FILL-IN: Define input x and corresponding correct answer y_


# FILL-IN: Define W1, b1, y1 of layer 1


# FILL-IN: Define W2, b2, y2 of layer 2

# FILL-IN: Define output y


# FILL-IN: Define cross_entropy and train_step
```



```
# Initializing the variables
init = tf.global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    for _ in range(10000):
        sess.run(train_step, feed_dict={x: X_data, y_: y_onehot})

    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    print('accuracy: ', sess.run(accuracy, feed_dict={x: X_data, y_: y_onehot})*100, '%')
```

```
accuracy:  99.1599977016449 %
```

CONGRATULATIONS - YOU ARE DONE WITH Exercise 6 !!!