PROBLEM STATEMENT -

Complete the function below. The model below should: create placeholders, initialize parameters, forward propagate, compute the cost, create an optimizer.

In [1]:

```
import math
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
import tensorflow as tf
from tensorflow.python.framework import ops

%matplotlib inline
np.random.seed(1)
```

```
WARNING:tensorflow:From C:\Users\USER\anaconda3\Lib\site-packages\keras\src
\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecat
ed. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

In [2]:
```python
def load_dataset():
    train_dataset = h5py.File('datasets/train_signs.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train

    test_dataset = h5py.File('datasets/test_signs.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set

    classes = np.array(test_dataset["list_classes"][:]) # the list of classes

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0])
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_ori


def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    """
    Creates a list of random minibatches from (X, Y)

    Arguments:
    X -- input data, of shape (input size, number of examples) (m, Hi, Wi, Ci)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1,
    mini_batch_size - size of the mini-batches, integer
    seed -- this is only for the purpose of grading, so that you're "random mi

    Returns:
    mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
    """

    m = X.shape[0]                  # number of training examples
    mini_batches = []
    np.random.seed(seed)

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[permutation,:,:,:]
    shuffled_Y = Y[permutation,:]

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[k * mini_batch_size : k * mini_batch_size +
        mini_batch_Y = shuffled_Y[k * mini_batch_size : k * mini_batch_size +
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[num_complete_minibatches * mini_batch_size :
        mini_batch_Y = shuffled_Y[num_complete_minibatches * mini_batch_size :
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

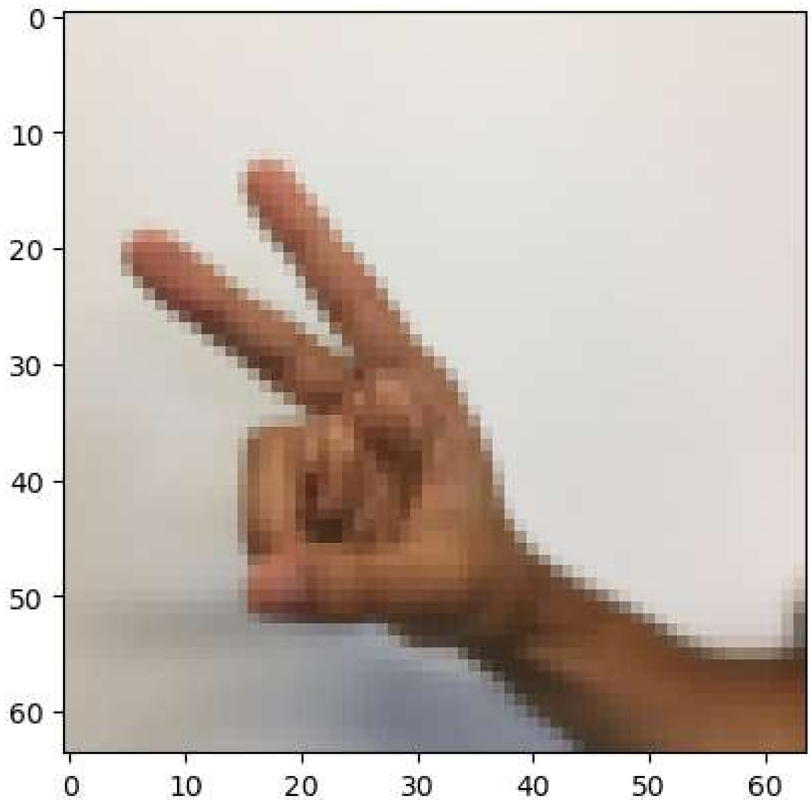
```
def convert_to_one_hot(Y, C):
    Y = np.eye(C)[Y.reshape(-1)].T
    return Y


def forward_propagation_for_predict(X, parameters):
    """
    Implements the forward propagation for the model: LINEAR -> RELU -> LINEAR

    Arguments:
    X -- input dataset placeholder, of shape (input size, number of examples)
    parameters -- python dictionary containing your parameters "W1", "b1", "W2
                  the shapes are given in initialize_parameters

    Returns:
    Z3 -- the output of the last LINEAR unit
    """

    # Retrieve the parameters from the dictionary "parameters"
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    W3 = parameters['W3']
    b3 = parameters['b3']
                                                          # Numpy Equivalents
    Z1 = tf.add(tf.matmul(W1, X), b1)                     # Z1 = np.dot(W1, X
    A1 = tf.nn.relu(Z1)                                   # A1 = relu(Z1)
    Z2 = tf.add(tf.matmul(W2, A1), b2)                    # Z2 = np.dot(W2, a
    A2 = tf.nn.relu(Z2)                                   # A2 = relu(Z2)
    Z3 = tf.add(tf.matmul(W3, A2), b3)                    # Z3 = np.dot(W3,Z2

    return Z3

def predict(X, parameters):

    W1 = tf.convert_to_tensor(parameters["W1"])
    b1 = tf.convert_to_tensor(parameters["b1"])
    W2 = tf.convert_to_tensor(parameters["W2"])
    b2 = tf.convert_to_tensor(parameters["b2"])
    W3 = tf.convert_to_tensor(parameters["W3"])
    b3 = tf.convert_to_tensor(parameters["b3"])

    params = {"W1": W1,
              "b1": b1,
              "W2": W2,
              "b2": b2,
              "W3": W3,
              "b3": b3}

    x = tf.placeholder("float", [12288, 1])

    z3 = forward_propagation_for_predict(x, params)
    p = tf.argmax(z3)

    sess = tf.Session()
```

```
    prediction = sess.run(p, feed_dict = {x: X})

    return prediction

#def predict(X, parameters):
#
#    W1 = tf.convert_to_tensor(parameters["W1"])
#    b1 = tf.convert_to_tensor(parameters["b1"])
#    W2 = tf.convert_to_tensor(parameters["W2"])
#    b2 = tf.convert_to_tensor(parameters["b2"])
##    W3 = tf.convert_to_tensor(parameters["W3"])
##    b3 = tf.convert_to_tensor(parameters["b3"])
#
##    params = {"W1": W1,
##              "b1": b1,
##              "W2": W2,
##              "b2": b2,
##              "W3": W3,
##              "b3": b3}
#
#    params = {"W1": W1,
#              "b1": b1,
#              "W2": W2,
#              "b2": b2}
#
#    x = tf.placeholder("float", [12288, 1])
#
#    z3 = forward_propagation(x, params)
#    p = tf.argmax(z3)
#
#    with tf.Session() as sess:
#        prediction = sess.run(p, feed_dict = {x: X})
#
#    return prediction
```

In [3]:
```
# Loading the data (signs)
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset()
```

In [4]:
```
# Example of a picture
index = 6
plt.imshow(X_train_orig[index])
print ("y = " + str(np.squeeze(Y_train_orig[:, index])))
```

```
y = 2
```



In [5]:
```
X_train = X_train_orig/255.
X_test = X_test_orig/255.
Y_train = convert_to_one_hot(Y_train_orig, 6).T
Y_test = convert_to_one_hot(Y_test_orig, 6).T
print ("number of training examples = " + str(X_train.shape[0]))
print ("number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))
conv_layers = {}
```

```
number of training examples = 1080
number of test examples = 120
X_train shape: (1080, 64, 64, 3)
Y_train shape: (1080, 6)
X_test shape: (120, 64, 64, 3)
Y_test shape: (120, 6)
```

In [6]:
```python
# GRADED FUNCTION: create_placeholders

def create_placeholders(n_H0, n_W0, n_C0, n_y):
    """
    Creates the placeholders for the tensorflow session.

    Arguments:
    n_H0 -- scalar, height of an input image
    n_W0 -- scalar, width of an input image
    n_C0 -- scalar, number of channels of the input
    n_y -- scalar, number of classes

    Returns:
    X -- placeholder for the data input, of shape [None, n_H0, n_W0, n_C0] and
    Y -- placeholder for the input labels, of shape [None, n_y] and dtype "flo
    """

    ### START CODE HERE ### (≈2 lines)
    X = tf.keras.Input(dtype=tf.float32, shape=(None, n_H0, n_W0, n_C0), name=
    Y = tf.keras.Input(dtype=tf.float32, shape=(None, n_y), name="Y")
    ### END CODE HERE ###

    return X, Y
```

In [7]:
```python
X, Y = create_placeholders(64, 64, 3, 6)
print ("X = " + str(X))
print ("Y = " + str(Y))
```

```
WARNING:tensorflow:From C:\Users\USER\anaconda3\Lib\site-packages\keras\src
\backend.py:1398: The name tf.executing_eagerly_outside_functions is depreca
ted. Please use tf.compat.v1.executing_eagerly_outside_functions instead.

X = KerasTensor(type_spec=TensorSpec(shape=(None, None, 64, 64, 3), dtype=t
f.float32, name='X'), name='X', description="created by layer 'X'")
Y = KerasTensor(type_spec=TensorSpec(shape=(None, None, 6), dtype=tf.float3
2, name='Y'), name='Y', description="created by layer 'Y'")
```

In [8]:
```
def initialize_parameters():
    """
    Initializes weight parameters to build a neural network with TensorFlow. T
                        W1 : [4, 4, 3, 8]
                        W2 : [2, 2, 8, 16]
    Note that we will hard code the shape values in the function to make the g
    Normally, functions should take values as inputs rather than hard coding.
    Returns:
    parameters -- a dictionary of tensors containing W1, W2
    """

    tf.random.set_seed(1)                                    # so that your "random"

    ### START CODE HERE ###
    initializer = tf.initializers.GlorotUniform(seed=0)
    W1 = tf.Variable(initializer(shape=(4, 4, 3, 8)), name="W1", trainable=Tru
    W2 = tf.Variable(initializer(shape=(2, 2, 8, 16)), name="W2", trainable=Tr
    ### END CODE HERE ###

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

    return parameters
```

In [9]:
```
# Reset the graph
tf.keras.backend.clear_session()

# Test the function
parameters = initialize_parameters()

print("W1[1,1,1] = \n" + str(parameters["W1"][1,1,1].numpy()))
print("W1.shape: " + str(parameters["W1"].shape))
print("\n")
print("W2[1,1,1] = \n" + str(parameters["W2"][1,1,1].numpy()))
print("W2.shape: " + str(parameters["W2"].shape))
```

```
W1[1,1,1] =
[-0.05346771  0.18349849 -0.01215445  0.00138046  0.0012947  -0.02904211
 -0.11260509 -0.143055   ]
W1.shape: (4, 4, 3, 8)


W2[1,1,1] =
[-0.1713624   0.09527719 -0.0744766  -0.02245569  0.24450928 -0.06879854
  0.21546292 -0.08803296 -0.16513646 -0.19527972 -0.22957063  0.15745944
  0.13090086 -0.12304181 -0.05287278  0.03434092]
W2.shape: (2, 2, 8, 16)
```

In [10]:
```
def forward_propagation(X, parameters):
    """
    Implements the forward propagation for the model:
    CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLY

    Note that for simplicity and grading purposes, we'll hard-code some values
    such as the stride and kernel (filter) sizes.
    Normally, functions should take these values as function parameters.

    Arguments:
    X -- input dataset placeholder, of shape (batch_size, input_height, input_
    parameters -- python dictionary containing your parameters "W1", "W2"
                  the shapes are given in initialize_parameters

    Returns:
    Z3 -- the output of the last LINEAR unit
    """

    # Retrieve the parameters from the dictionary "parameters"
    W1 = parameters['W1']
    W2 = parameters['W2']

    ### START CODE HERE ###
    # CONV2D: stride of 1, padding 'SAME'
    Z1 = tf.nn.conv2d(X, W1, strides=[1, 1, 1, 1], padding="SAME")
    # RELU
    A1 = tf.nn.relu(Z1)
    # MAXPOOL: window 8x8, stride 8, padding 'SAME'
    P1 = tf.nn.max_pool(A1, ksize=[1, 8, 8, 1], strides=[1, 8, 8, 1], padding=
    # CONV2D: filters W2, stride 1, padding 'SAME'
    Z2 = tf.nn.conv2d(P1, W2, strides=[1, 1, 1, 1], padding="SAME")
    # RELU
    A2 = tf.nn.relu(Z2)
    # MAXPOOL: window 4x4, stride 4, padding 'SAME'
    P2 = tf.nn.max_pool(A2, ksize=[1, 4, 4, 1], strides=[1, 4, 4, 1], padding=
    # FLATTEN
    F = tf.keras.layers.Flatten()(P2)
    # FULLY-CONNECTED without non-linear activation function (not not call sof
    # 6 neurons in output layer.
    Z3 = tf.keras.layers.Dense(units=6, activation=None)(F)
    ### END CODE HERE ###

    return Z3
```

In [11]:
```
# Reset the graph
tf.keras.backend.clear_session()

# Test the function
X_input = tf.keras.Input(shape=(64, 64, 3))
parameters = initialize_parameters()
Z3 = forward_propagation(X_input, parameters)

# Create a model
model = tf.keras.Model(inputs=X_input, outputs=Z3)

# Test the model
np.random.seed(1)
a = model.predict(np.random.randn(2, 64, 64, 3))
print("Z3 = \n" + str(a))
```

```
WARNING:tensorflow:The following Variables were used in a Lambda layer's cal
l (tf.compat.v1.nn.conv2d), but are not present in its tracked objects:   <t
f.Variable 'W1:0' shape=(4, 4, 3, 8) dtype=float32>. This is a strong indica
tion that the Lambda layer should be rewritten as a subclassed Layer.
WARNING:tensorflow:The following Variables were used in a Lambda layer's cal
l (tf.compat.v1.nn.conv2d_1), but are not present in its tracked objects:
<tf.Variable 'W2:0' shape=(2, 2, 8, 16) dtype=float32>. This is a strong ind
ication that the Lambda layer should be rewritten as a subclassed Layer.
1/1 [==============================] - 0s 243ms/step
Z3 =
[[ 1.3701663   0.32746893 -1.7422659   1.7024851   1.2658226  -0.8624959 ]
 [ 1.5811116   0.20760003 -1.6109662   1.6004725   1.2302411  -0.7460443 ]]
```

In [12]:
```
def compute_cost(Z3, Y):
    """
    Computes the cost

    Arguments:
    Z3 -- output of forward propagation (output of the last LINEAR unit), of s
    Y -- "true" labels vector placeholder, same shape as Z3

    Returns:
    cost - Tensor of the cost function
    """

    ### START CODE HERE ###
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y, lo
    ### END CODE HERE ###

    return cost
```

In [13]:
```
tf.keras.backend.clear_session()

# Create placeholders
X = tf.keras.Input(shape=(64, 64, 3))
Y = tf.keras.Input(shape=(6,))
# Initialize parameters
parameters = initialize_parameters()
# Forward propagation
Z3 = forward_propagation(X, parameters)


# Test the function
np.random.seed(1)
a = compute_cost(np.random.randn(4, 6), np.random.randn(4, 6))
print("cost = " + str(a))
```

```
WARNING:tensorflow:The following Variables were used in a Lambda layer's cal
l (tf.compat.v1.nn.conv2d), but are not present in its tracked objects:   <t
f.Variable 'W1:0' shape=(4, 4, 3, 8) dtype=float32>. This is a strong indica
tion that the Lambda layer should be rewritten as a subclassed Layer.
WARNING:tensorflow:The following Variables were used in a Lambda layer's cal
l (tf.compat.v1.nn.conv2d_1), but are not present in its tracked objects:
<tf.Variable 'W2:0' shape=(2, 2, 8, 16) dtype=float32>. This is a strong ind
ication that the Lambda layer should be rewritten as a subclassed Layer.
cost = tf.Tensor(-1.4658942658878216, shape=(), dtype=float64)
```

In [14]:
```
# GRADED FUNCTION: model

def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.009,
          num_epochs = 100, minibatch_size = 64, print_cost = True):
    """
    Implements a three-layer ConvNet in Tensorflow:
    CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLY

    Arguments:
    X_train -- training set, of shape (None, 64, 64, 3)
    Y_train -- test set, of shape (None, n_y = 6)
    X_test -- training set, of shape (None, 64, 64, 3)
    Y_test -- test set, of shape (None, n_y = 6)
    learning_rate -- learning rate of the optimization
    num_epochs -- number of epochs of the optimization loop
    minibatch_size -- size of a minibatch
    print_cost -- True to print the cost every 100 epochs

    Returns:
    train_accuracy -- real number, accuracy on the train set (X_train)
    test_accuracy -- real number, testing accuracy on the test set (X_test)
    parameters -- parameters learnt by the model. They can then be used to pre
    """

    ops.reset_default_graph()                         # to be able to rerun th
    tf.random.set_seed(1)                             # to keep results consis
    seed = 3                                          # to keep results consis
    (m, n_H0, n_W0, n_C0) = X_train.shape
    n_y = Y_train.shape[1]
    costs = []                                        # To keep track of the c

    # Create Placeholders of the correct shape
    ### START CODE HERE ### (1 line)
    X, Y = create_placeholders(n_H0, n_W0, n_C0, n_y)
    ### END CODE HERE ###

    # Initialize parameters
    ### START CODE HERE ### (1 line)
    parameters = initialize_parameters()
    ### END CODE HERE ###

    # Forward propagation: Build the forward propagation in the tensorflow gra
    ### START CODE HERE ### (1 line)
    Z3 = forward_propagation(X, parameters)
    ### END CODE HERE ###

    # Cost function: Add cost function to tensorflow graph
    ### START CODE HERE ### (1 line)
    cost = compute_cost(Z3, Y)
    ### END CODE HERE ###

    # Backpropagation: Define the tensorflow optimizer. Use an AdamOptimizer t
    ### START CODE HERE ### (1 line)
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(c
    ### END CODE HERE ###

    # Initialize all the variables globally
```

```
    init = tf.global_variables_initializer()

    # Start the session to compute the tensorflow graph
    with tf.Session() as sess:

        # Run the initialization
        sess.run(init)

        # Do the training loop
        for epoch in range(num_epochs):

            minibatch_cost = 0.
            num_minibatches = int(m / minibatch_size) # number of minibatches
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_train, minibatch_size

            for minibatch in minibatches:

                # Select a minibatch
                (minibatch_X, minibatch_Y) = minibatch
                """
                # IMPORTANT: The line that runs the graph on a minibatch.
                # Run the session to execute the optimizer and the cost.
                # The feedict should contain a minibatch for (X,Y).
                """
                ### START CODE HERE ### (1 line)
                _ , temp_cost = sess.run([optimizer, cost], feed_dict={X:minib
                ### END CODE HERE ###

                minibatch_cost += temp_cost / num_minibatches


            # Print the cost every epoch
            if print_cost == True and epoch % 5 == 0:
                print ("Cost after epoch %i: %f" % (epoch, minibatch_cost))
            if print_cost == True and epoch % 1 == 0:
                costs.append(minibatch_cost)

        # Calculate the correct predictions
        predict_op = tf.argmax(Z3, 1)
        correct_prediction = tf.equal(predict_op, tf.argmax(Y, 1))

        # Calculate accuracy on the test set
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
        print(accuracy)
        train_accuracy = accuracy.eval({X: X_train, Y: Y_train})
        test_accuracy = accuracy.eval({X: X_test, Y: Y_test})
        print("Train Accuracy:", train_accuracy)
        print("Test Accuracy:", test_accuracy)

        return train_accuracy, test_accuracy, parameters
```

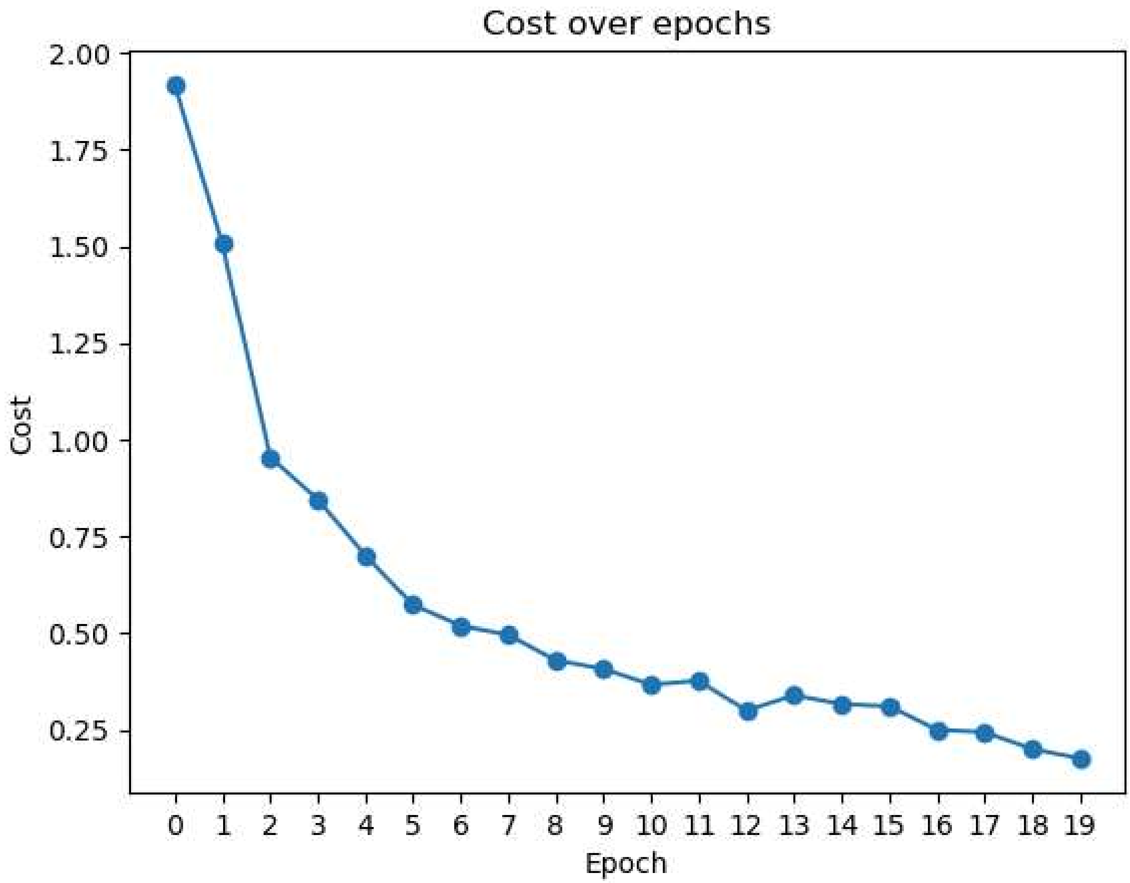
In [15]:

```
_, _, parameters = model(X_train, Y_train, X_test, Y_test)
```

```
Cost after epoch 0: 1.917929
Cost after epoch 5: 1.506757
Cost after epoch 10: 0.955359
Cost after epoch 15: 0.845802
Cost after epoch 20: 0.701174
Cost after epoch 25: 0.571977
Cost after epoch 30: 0.518435
Cost after epoch 35: 0.495806
Cost after epoch 40: 0.429827
Cost after epoch 45: 0.407291
Cost after epoch 50: 0.366394
Cost after epoch 55: 0.376922
Cost after epoch 60: 0.299491
Cost after epoch 65: 0.338870
Cost after epoch 70: 0.316400
Cost after epoch 75: 0.310413
Cost after epoch 80: 0.249549
Cost after epoch 85: 0.243457
Cost after epoch 90: 0.200031
Cost after epoch 95: 0.175452
Tensor("Mean_1:0", shape=(), dtype=float32)
Train Accuracy: 0.940741
Test Accuracy: 0.783333
```

In [17]:

```
# 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()
```



In [ ]: