In [1]:

```
from __future__ import print_function
import struct
import numpy as np
import matplotlib.pyplot as plt
import math
import tensorflow as tf
from tensorflow.python.framework import ops


def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.fromstring(f.read(), dtype=np.uint8).reshape(shape)


trainImages = read_idx('train-images-idx3-ubyte')
trainLabels = read_idx('train-labels-idx1-ubyte')
testImages = read_idx('t10k-images-idx3-ubyte')
testLabels = read_idx('t10k-labels-idx1-ubyte')

train_label = trainLabels.reshape(trainImages.shape[0],1)
test_label = testLabels.reshape(testImages.shape[0],1)

print('trainImages size:' + str(trainImages.shape))
print('trainLabels size:' + str(train_label.shape))
print('testImages size:' + str(testImages.shape))
print('testLabels size:' + str(test_label.shape))
```

```
//anaconda3/envs/tensorflow/lib/python3.5/importlib/_bootstrap.py:222:
RuntimeWarning: compiletime version 3.6 of module 'tensorflow.python.f
ramework.fast_tensor_util' does not match runtime version 3.5
  return f(*args, **kwds)

trainImages size:(60000, 28, 28)
trainLabels size:(60000, 1)
testImages size:(10000, 28, 28)
testLabels size:(10000, 1)

//anaconda3/envs/tensorflow/lib/python3.5/site-packages/ipykernel_laun
cher.py:14: DeprecationWarning: The binary mode of fromstring is depre
cated, as it behaves surprisingly on unicode inputs. Use frombuffer in
stead
```

In [2]:

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

train_label_one_hot = one_hot_matrix(train_label,10).T
test_label_one_hot = one_hot_matrix(test_label,10).T
train_data = trainImages/255
test_data = testImages/255
train_data = train_data.reshape(train_data.shape[0],28,28,1)
test_data = test_data.reshape(test_data.shape[0],28,28,1)

print('train_image size:' + str(train_data.shape))
print('train_label size:' + str(train_label_one_hot.shape))
print('test_image size:' + str(test_data.shape))
print('test_label size:' + str(test_label_one_hot.shape))
```

```
train_image size:(60000, 28, 28, 1)
train_label size:(60000, 10)
test_image size:(10000, 28, 28, 1)
test_label size:(10000, 10)
```

In [3]:

```
#Creating Placeholders

'''
Arguments:
n_H -- scalar, height of an input image
n_W -- scalar, width of an input image
n_C -- scalar, number of channels of the input
n_y -- scalar, number of classes

'''
def create_placeholder(n_H,n_W,n_C,n_y):
    X = tf.placeholder(tf.float32, [None, n_H, n_W, n_C])
    Y = tf.placeholder(tf.float32, [None, n_y])

    return X,Y
```

In [4]:

```
def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    #number of training examples
    m = X.shape[0]
    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). (Not including the end case)
    #number of mini batches of size mini_batch_size in your partitionning
    num_complete_minibatches = math.floor(m/mini_batch_size)
    for i in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[i * mini_batch_size : i * mini_batch_size + mini_b
        mini_batch_Y = shuffled_Y[i * mini_batch_size : i * mini_batch_size + mini_b
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    #Step 3: 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 : m,:,:
        mini_batch_Y = shuffled_Y[num_complete_minibatches * mini_batch_size : m,:]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

In [5]:

```
#Initializing Parameters
def initialize_parameters(n_size):

    tf.set_random_seed(1)

    W1 = tf.get_variable("W1", [3, 3, 1, n_size], initializer=tf.contrib.layers.xavi
    W2 = tf.get_variable("W2", [3, 3, n_size, n_size], initializer=tf.contrib.layers
    W3 = tf.get_variable("W3", [3, 3, n_size, n_size], initializer=tf.contrib.layers
    W4 = tf.get_variable("W4", [3, 3, n_size, n_size], initializer=tf.contrib.layers
    W5 = tf.get_variable("W5", [3, 3, n_size, 10], initializer=tf.contrib.layers.xav

    parameters = {"W1": W1,
                  "W2": W2,
                  "W3": W3,
                  "W4": W4,
                  "W5": W5}

    return parameters
```

In [6]:

```
#Forward Propagation
def forward_propagation(X,parameters):

    W1 = parameters['W1']
    W2 = parameters['W2']
    W3 = parameters['W3']
    W4 = parameters['W4']
    W5 = parameters['W5']

    #Layer 1
    Z1 = tf.nn.conv2d(X,W1, strides = [1,1,1,1], padding = 'SAME')
    A1 = tf.nn.leaky_relu(Z1,alpha=0.2)

    #Layer 2
    Z2 = tf.nn.atrous_conv2d(A1,W2, rate=2, padding = 'SAME')
    A2 = tf.nn.leaky_relu(Z2,alpha=0.2)

    #Layer 3
    Z3 = tf.nn.atrous_conv2d(A2,W3, rate=4, padding = 'SAME')
    A3 = tf.nn.leaky_relu(Z3,alpha=0.2)

    #Layer 4
    Z4 = tf.nn.atrous_conv2d(A3,W4, rate=8, padding = 'SAME')
    A4 = tf.nn.leaky_relu(Z4,alpha=0.2)

    #Layer 5
    Z5 = tf.nn.conv2d(A4,W5, strides = [1,1,1,1], padding = 'SAME')
    A5 = tf.nn.leaky_relu(Z5,alpha=0.2)

    #Layer 6 Global Average Pool
    Z6=  tf.reduce_mean(A5, axis=[1,2])

    #Flatten the CAN output so that we can connect it with fully connected layers
    Z7 = tf.contrib.layers.flatten(Z6)
    #Z7 = tf.contrib.layers.fully_connected(A6, 10, activation_fn=tf.nn.softmax)

    return Z7
```

In [7]:

```
#Compute Cost
def compute_cost(Z7,Y):

    logits = Z7
    labels = Y
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, lab

    return cost
```

In [8]:

```
def model(X_train, Y_train, X_test, Y_test,n_size,learning_rate = 0.005,
          num_epochs = 10, minibatch_size = 64, print_cost = True):


    #Rerun model without overwriting tf variables
    ops.reset_default_graph()
    tf.set_random_seed(1)
    seed = 3
    (m, n_H, n_W, n_C) = X_train.shape
    n_y = Y_train.shape[1]
    costs = []

    X,Y = create_placeholder(n_H,n_W,n_C,n_y)

    parameters = initialize_parameters(n_size)

    #Forward propagation: Build the forward propagation
    Z7 = forward_propagation(X,parameters)

    #Cost function: Add cost function to tensorflow graph
    cost = compute_cost(Z7, Y)

    #Backpropagation: Descent of Gradient Usine AdamOptimizer
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

    # Initialize all the variables
    init = tf.global_variables_initializer()

    with tf.Session() as sess:

        #Session to compute tensorflow graph
        sess.run(init)

        #Training Loop
        for epoch in range(num_epochs):

            #Define a cost related to an mini_batch
            minibatch_cost = 0
            #Number of minibatches of size minibatch_size in the train set
            num_minibatches = int(m / minibatch_size)
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_train, minibatch_size, seed

            for minibatch in minibatches:

                (minibatch_X, minibatch_Y) = minibatch

                #The line that runs the graph on a minibatch.
                _ , temp_cost = sess.run([optimizer, cost], feed_dict={X:minibatch_X

                #Total epoch cost for all minibatches combined
                minibatch_cost += temp_cost / num_minibatches

            # Print the cost every epoch
            if print_cost == True:
                print ("Cost after epoch %i: %f" % (epoch, minibatch_cost))

        parameters = sess.run(parameters)
        print("Parameters have been trained!")
```

```
        #Calculate the correct predictions
        #Returns the index with the largest value
        correct_prediction = tf.equal(tf.argmax(Z7,1), tf.argmax(Y,1))

        #Calculate accuracy on the test set
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, dtype="float"))


        test_accuracy = np.zeros(5)

        for i in range(5):

            test_accuracy[i] = accuracy.eval({X: X_test[i*2000:(i+1)*2000-1,:,:,:],

        test_accuracy1 = np.mean(test_accuracy)

        print("No. of channels=%d, Test Accuracy:%.2f%%" % (n_size, test_accuracy1 *

        train_accuracy=np.zeros(30)

        for j in range(30):

            train_accuracy[j] = accuracy.eval({X: X_train[j*2000:(j+1)*2000-1,:,:,:]

        train_accuracy1 = np.mean(train_accuracy)

        print("No. of channels=%d, Train Accuracy:%.2f%%" % (n_size, train_accuracy1

        return train_accuracy1,test_accuracy1,parameters
```

In [9]:

```
train_accuracy1, test_accuracy1, parameters = model(train_data, train_label_one_hot,
```

```
WARNING:tensorflow:From <ipython-input-7-7f8267c91fb5>:6: softmax_cros
s_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecate
d and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See @{tf.nn.softmax_cross_entropy_with_logits_v2}.

Cost after epoch 0: 0.361774
Cost after epoch 1: 0.106151
Cost after epoch 2: 0.076115
Cost after epoch 3: 0.058259
Cost after epoch 4: 0.052203
Cost after epoch 5: 0.047445
Cost after epoch 6: 0.042710
Cost after epoch 7: 0.037390
Cost after epoch 8: 0.039149
Cost after epoch 9: 0.032402
Parameters have been trained!
No. of channels=16, Test Accuracy:99.21%
No. of channels=16, Train Accuracy:99.21%
```

In [ ]:

```
```