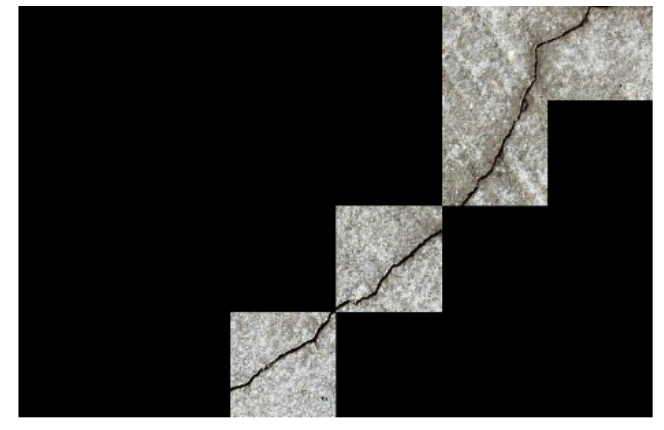
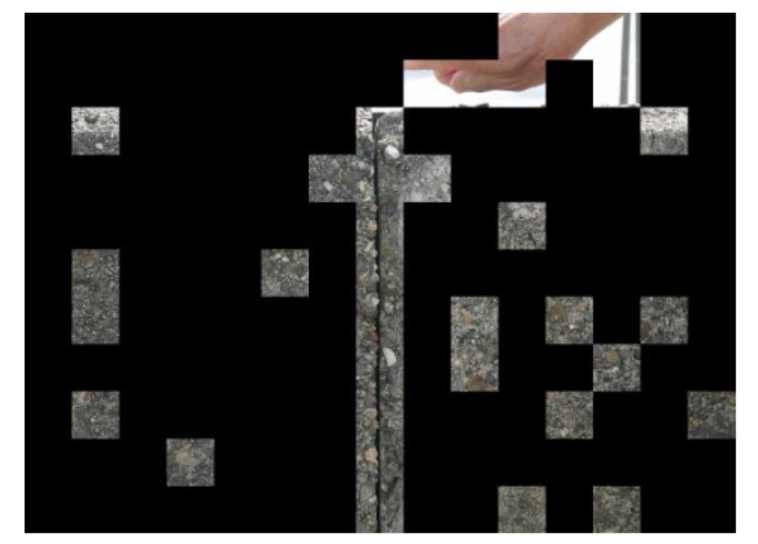
The model acheived 85% accuracy on the validation set.







The dataset I used :

<https://drive.google.com/file/d/1kC60RGO3rcScVk7HY-s7tTMJeMbADfh1/view>

Selected Code:

"""

Code to train the model

"""

import tensorflow as tf

import numpy as np

import time

from datetime import timedelta

from dataset import load\_cached

#from matplotlib.image import imread

import cv2,sys,argparse

#Initialzing the conv and max\_pool layers

#####################################################

def new\_conv\_layer(input,              # The previous layer.

                   num\_input\_channels, # Num. channels in prev. layer.

                   filter\_size,        # Width and height of each filter.

                   num\_filters):        # Number of filters.

    # Shape of the filter-weights for the convolution.

    shape = [filter\_size, filter\_size, num\_input\_channels, num\_filters]

    # Create new weights aka. filters with the given shape.

    weights = tf.Variable(tf.truncated\_normal(shape, stddev=0.05))

    # Create new biases, one for each filter.

    biases = tf.Variable(tf.constant(0.05, shape=[num\_filters]))

    layer = tf.nn.conv2d(input=input,

                         filter=weights,

                         strides=[1, 2, 2, 1],

                         padding='VALID')

    # A bias-value is added to each filter-channel.

    layer += biases

    return layer

####################################################

def max\_pool(layer,ksize,strides):

    layer = tf.nn.max\_pool(value=layer,

                           ksize=ksize,

                           strides = strides,

                           padding = 'VALID')

    return layer

####################################################

def new\_fc\_layer(input,           # The previous layer.

                 num\_inputs,      # Num. inputs from prev. layer.

                 num\_outputs,     # Num. outputs

                 use\_relu=True):  # Use Rectified Linear Unit (ReLU)?

    # Create new weights and biases.

    weights =tf.Variable(tf.truncated\_normal([num\_inputs, num\_outputs], stddev=0.05))

    biases = tf.Variable(tf.constant(0.05, shape=[num\_outputs]))

    #Include Drop-out as well to avoid overfitting

    #x\_drop = tf.nn.dropout(input, keep\_prob=keep\_prob\_input)

    # Calculate the layer as the matrix multiplication of

    # the input and weights, and then add the bias-values.

    layer = tf.matmul(input, weights) + biases

    # Use ReLU?

    if use\_relu:

        layer = tf.nn.relu(layer)

    return layer

####################################################

def flatten\_layer(layer):

    # Get the shape of the input layer.

    layer\_shape = layer.get\_shape()

    # The shape of the input layer is assumed to be:

    # layer\_shape == [num\_images, img\_height, img\_width, num\_channels]

    # The number of features is: img\_height \* img\_width \* num\_channels

    num\_features = layer\_shape[1:4].num\_elements()

    layer\_flat = tf.reshape(layer, [-1, num\_features])

    # The shape of the flattened layer is now:

    # [num\_images, img\_height \* img\_width \* num\_channels]

    return layer\_flat, num\_features

####################################################

class Model:

    def \_\_init\_\_(self,in\_dir,save\_folder=None):

        dataset = load\_cached(cache\_path='my\_dataset\_cache.pkl', in\_dir=in\_dir)

        self.num\_classes = dataset.num\_classes

        image\_paths\_train, cls\_train, self.labels\_train = dataset.get\_training\_set()

        image\_paths\_test, self.cls\_test, self.labels\_test = dataset.get\_test\_set()

        ##############################IMAGE PARAMETERS#####################################

        self.img\_size = 128

        self.num\_channels = 3

        self.train\_batch\_size = 64

        self.test\_batch\_size = 64

        ###################################################################################

        self.x = tf.placeholder(tf.float32, shape=[None, self.img\_size,self.img\_size,self.num\_channels], name='x')

        self.x\_image = tf.reshape(self.x, [-1, self.img\_size, self.img\_size, self.num\_channels])

        self.y\_true = tf.placeholder(tf.float32, shape=[None, self.num\_classes], name='y\_true')

        self.y\_true\_cls = tf.argmax(self.y\_true, axis=1) #The True class Value

        self.keep\_prob = tf.placeholder(tf.float32)

        self.keep\_prob\_2 = tf.placeholder(tf.float32)

        self.y\_pred\_cls = None

        self.train\_images= self.load\_images(image\_paths\_train)

        self.test\_images= self.load\_images(image\_paths\_test)

        self.save\_folder=save\_folder

        self.optimizer,self.accuracy = self.define\_model()

    def load\_images(self,image\_paths):

        # Load the images from disk.

        images = [cv2.imread(path,1) for path in image\_paths]

        # Convert to a numpy array and return it in the form of [num\_images,size,size,channel]

        #print(np.asarray(images[0]).shape)

        return np.asarray(images)

    def define\_model(self):

        #Convolution Layer 1

        filter\_size1 = 10          # Convolution filters are 10 x 10

        num\_filters1 = 24         # There are 24 of these filters.

        # Convolutional Layer 2

        filter\_size2 = 7          # Convolution filters are 7 x 7

        num\_filters2 = 48         # There are 48 of these filters.

        # Convolutional Layer 3

        filter\_size3 = 11          # Convolution filters are 11 x 11

        num\_filters3 = 96         # There are 96 of these filters.

        # Fully-connected layer

        fc\_size = 96

        layer\_conv1 = new\_conv\_layer(input=self.x\_image,

                                     num\_input\_channels=self.num\_channels,

                                     filter\_size=filter\_size1,

                                     num\_filters=num\_filters1)

        #Max Pool Layer

        ksize1 = [1,4,4,1]

        strides1 = [1,2,2,1]

        layer\_max\_pool1 = max\_pool(layer\_conv1,ksize1,strides1)

        #Convolutional Layer 2

        layer\_conv2 = new\_conv\_layer(input=layer\_max\_pool1,

                                     num\_input\_channels=num\_filters1,

                                     filter\_size=filter\_size2,

                                     num\_filters=num\_filters2)

        #Max Pool Layer

        ksize2 = [1,2,2,1]

        strides2 = [1,1,1,1]

        layer\_max\_pool2 = max\_pool(layer\_conv2,ksize2,strides2)

        #Convolutional Layer 3

        layer\_conv3 = new\_conv\_layer(input=layer\_max\_pool2,

                                     num\_input\_channels=num\_filters2,

                                     filter\_size=filter\_size3,

                                     num\_filters=num\_filters3)

        #Flatten

        layer\_flat, num\_features = flatten\_layer(layer\_conv3)

        #Relu Layer

        layer\_relu = tf.nn.relu(layer\_flat)

        #Fully-Connected Layer1

        layer\_fc1 = new\_fc\_layer(input=layer\_relu,

                                 num\_inputs=num\_features,

                                 num\_outputs=fc\_size,

                                 use\_relu=True)

        #Fully-Connected Layer2

        layer\_fc2 = new\_fc\_layer(input=layer\_fc1,

                                 num\_inputs=fc\_size,

                                 num\_outputs=self.num\_classes,

                                 use\_relu=False)

        #Predict the class

        y\_pred = tf.nn.softmax(layer\_fc2)

        self.y\_pred\_cls = tf.argmax(y\_pred, dimension=1,name="predictions")

        #Cost Function

        cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(logits=layer\_fc2, labels=self.y\_true)

        cost = tf.reduce\_mean(cross\_entropy)

        optimizer = tf.train.AdamOptimizer(learning\_rate=1e-4).minimize(cost)

        #Predict

        correct\_prediction = tf.equal(self.y\_pred\_cls, self.y\_true\_cls)

        accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

        return optimizer, accuracy

    def random\_batch(self):

        # Number of images in the training-set.

        num\_images = len(self.train\_images)

        # Create a random index.

        idx = np.random.choice(num\_images,

                               size=self.train\_batch\_size,

                               replace=False)

        # Use the random index to select random x and y-values.

        x\_batch = self.train\_images[idx]

        y\_batch = self.labels\_train[idx]

        return x\_batch, y\_batch

    def print\_test\_accuracy(self,sess):

        # Number of images in the test-set.

        num\_test = len(self.test\_images)

        # Allocate an array for the predicted classes which

        # will be calculated in batches and filled into this array.

        cls\_pred = np.zeros(shape=num\_test, dtype=np.int)

        i = 0

        while i < num\_test:

            # The ending index for the next batch is denoted j.

            j = min(i + self.test\_batch\_size, num\_test)

            images = self.test\_images[i:j]

            labels = self.labels\_test[i:j]

            # Create a feed-dict with these images and labels.

            feed\_dict = {self.x: images,

                 self.y\_true: labels,

                 self.keep\_prob: 1,

                 self.keep\_prob: 1}

            cls\_pred[i:j] = sess.run(self.y\_pred\_cls, feed\_dict=feed\_dict)

            # Set the start-index for the next batch to the

            # end-index of the current batch.

            i = j

        # Create a boolean array whether each image is correctly classified.

        correct = (self.cls\_test == cls\_pred)

         # Classification accuracy is the number of correctly classified

        # images divided by the total number of images in the test-set.

        acc = float(correct.sum()) / num\_test

        # Print the accuracy.

        msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"

        print(msg.format(acc, correct.sum(), num\_test))

    def optimize(self, num\_iterations):

        # Ensure we update the global variable rather than a local copy.

        global total\_iterations

        total\_iterations = 0

        saver = tf.train.Saver()

        # Start-time used for printing time-usage below.

        start\_time = time.time()

        with tf.Session() as sess:

            #global\_step\_int = tf.train.get\_global\_step(sess.graph)

            sess.run(tf.global\_variables\_initializer())

            for i in range(total\_iterations,

                           total\_iterations + num\_iterations):

                # Get a batch of training examples.

                # x\_batch now holds a batch of images and

                # y\_true\_batch are the true labels for those images.

                x\_batch, y\_true\_batch = self.random\_batch()

                feed\_dict\_train = {self.x: x\_batch,

                                   self.y\_true: y\_true\_batch}

                                   #self.keep\_prob: 0.5,

                                   #self.keep\_prob: 0.5}

                sess.run([self.optimizer], feed\_dict=feed\_dict\_train)

                # Print status every 100 iterations.

                if i % 100 == 0:

                    # Calculate the accuracy on the training-set.

                    feed\_dict\_acc = {self.x: x\_batch,

                                     self.y\_true: y\_true\_batch}

                                     #self.keep\_prob: 1,

                                     #self.keep\_prob: 1}

                    acc = sess.run(self.accuracy, feed\_dict=feed\_dict\_acc)

                    # Message for printing.

                    msg = "Optimization Iteration: {0:>6}, Training Accuracy: {1:>6.1%}"

                    # Print it.

                    print(msg.format(i + 1, acc))

                    # Update the total number of iterations performed.

                    total\_iterations += num\_iterations

                    # Ending time.

                    end\_time = time.time()

                if i%100 ==0:

                    #Calculate the accuracy on the test set every 100 iterations

                    self.print\_test\_accuracy(sess)

                if i%500 == 0:

                    #Saves every 500 iterations

                    saver.save(sess, os.path.join(self.save\_folder,'model')) #Change this according to your convenience

            # Difference between start and end-times.

            time\_dif = end\_time - start\_time

            self.print\_test\_accuracy(sess)

            # Print the time-usage.

            print("Time usage: " + str(timedelta(seconds=int(round(time\_dif)))))

            saver.save(sess, os.path.join(self.save\_folder,'model\_complete'))

def parse\_arguments():

    parser = argparse.ArgumentParser(description='Training Network')

    parser.add\_argument('--in\_dir',dest='in\_dir',type=str,default='cracky')

    parser.add\_argument('--iter',dest='num\_iterations',type=int,default=1500)

    parser.add\_argument('--save\_folder',dest='save\_folder',type=str,default=os.getcwd())

    return parser.parse\_args()

def  main(args):

    args=parse\_arguments()

    num\_iterations = args.num\_iterations

    model = Model(args.in\_dir,args.save\_folder)

    model.optimize(num\_iterations)

if \_\_name\_\_ == '\_\_main\_\_':

    main(sys.argv)