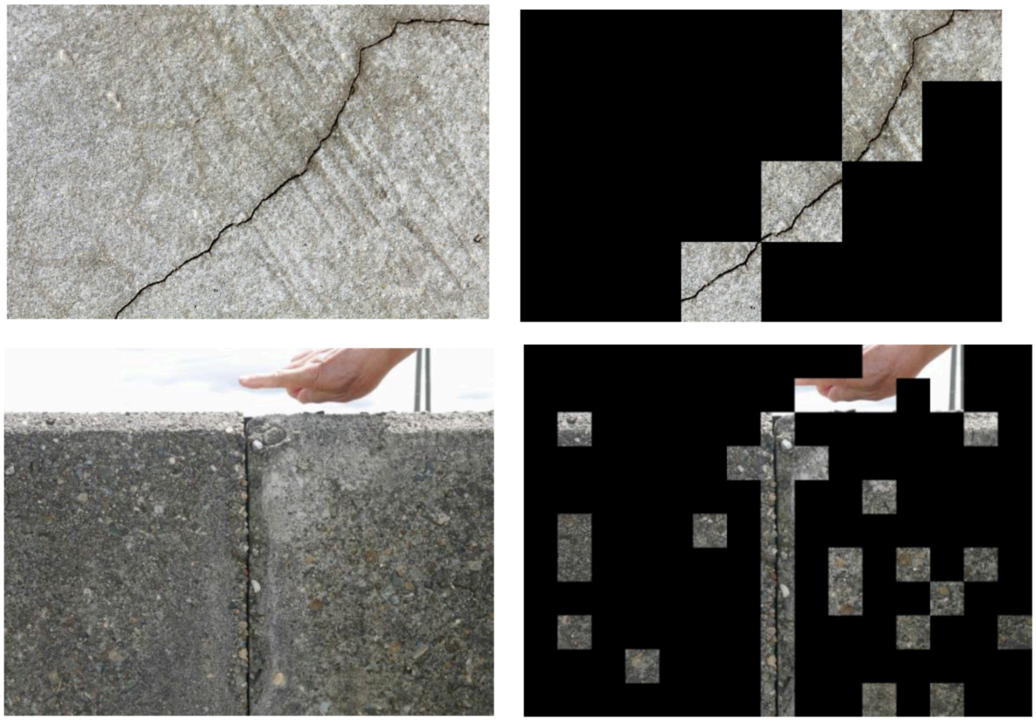
The model acheived 85% accuracy on the validation set.



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)
```