cloudclassifyANNColor.py	7/23/2024 9:54 PM	PY File	13 KB

“cloudclassifyANNColor.py” is the contains the final version of the cloud classify class

tester.py	7/23/2024 9:31 PM	PY File	3 KB

“tester.py” is the driver file used to run instances of the cloudclassifier class and output photos to file and output accuracy and other info to the console

non_max_suppression.py	7/14/2024 12:28 PM	PY File	2 KB

This module contains a fast non maximal suppression algorithm provided by the authors of the opencv packt publishing book

noisegenerator.py	7/21/2024 9:44 PM	PY File	1 KB

This is a script I used to generate instances of the “NEG” class of random noise to use as a negative detection. I wrote some other helper scripts the organize files and rename things that I either deleted or left in other folders.

cluster_vocab.npy	7/23/2024 9:20 PM	NPY File	11 KB
labels.npy	7/23/2024 9:21 PM	NPY File	2 KB
noisegenerator.py	7/21/2024 9:44 PM	PY File	1 KB
non_max_suppression.py	7/14/2024 12:28 PM	PY File	2 KB
outputlogs.py	7/23/2024 10:10 PM	PY File	25 KB
README.txt	7/23/2024 10:16 PM	Text Document	1 KB
samples.npy	7/23/2024 9:21 PM	NPY File	96 KB
tester.py	7/23/2024 9:31 PM	PY File	3 KB

These three numpy files are generated by the code when it has access to the dataset. Deleting them will recluster and create a new vocab. I couldn’t get opencv’s built in ann to save an instance of itself to file as it is supposed to be capable of, however since it trains pretty quickly, I just had it save the samples of the BOW descriptors it trains off of to file, since actually extracting the descriptors was the limiting factor in prep time. Changing parameters in the test file usually requires deleting the numpy files, unless the BOW CLUSTERS weren't changed.

Lets look at cloudclassifyANNColor.py

```python
import cv2 as cv
import numpy as np
import os
import itertools
from non_max_suppression import non_max_suppression_fast as nms

DATA_PATH = '../../Data/TestPhotos/'
CLASSES = ['NEG','Sky','Cumulus','Cirrus','Stratus']
NUM_CLASSES = len(CLASSES)

TRAINING_SAMPLES = 'samples.npy'
TRAINING_LABELS = 'labels.npy'
VOCAB_PATH = 'cluster_vocab.npy'

BOW_NUM_TRAINING_SAMPLES_PER_CLASS = 70
ANN_NUM_TRAINING_SAMPLES_PER_CLASS = 70

FLANN_INDEX_KDTREE = 1

```

Here are all the macros and imports needed in the class.

```python
class CloudClassify(object):
    def __init__(self):
        print("hello")
        self._inputImage = None
        self._classifier = None
        self._sift = None
        self._flann = None

        self._BOW_CLUSTERS = NUM_CLASSES * 4
        self._COLOR_BINS = 5
        self._INPUT_LAYERS = self._BOW_CLUSTERS + self._COLOR_BINS*3
        self._vocab = None
        self._bow_kmeans_trainer = None
        self._bow_extractor = None

        self._ann = None

        #Default values
        self._EPOCHS = 20
        self._ANN_CONF_THRESHOLD = 0.3
        self._SKY_WINDOW = 0.03, 0.08
        self._NEG_WINDOW = 0.03, 0.08

        self._ANN_LAYERS = [self._BOW_CLUSTERS, 64, NUM_CLASSES] # input are bow descriptors, output are classes

        self._NMS_OVERLAP_THRESHOLD = 0.3

        self._sky = None
        self._output = None
        self._READY = False
```

Here is an initializer and an initialization of all of the member variables with default values. These default values can be mutated by outside classes using the setters later in the file.

```
def run(self, inputFilePath=""):
    if not os.path.exists(inputFilePath):
        print("Couldn't find input image file")
    else:
        self._inputImage = cv.imread(inputFilePath)
        return self.detect_and_classify(self._inputImage, inputFilePath)
```

Here is the function for taking an input image by its file path and outputting a new image with the detection boxes drawn onto it along with the predominant cloud type. It is really a wrapper for detect_and_classify which does all the magic.

```
def get_path_data(self, data_class, i):
    path = DATA_PATH + data_class + "/" + data_class + str(i) + "R.JPG"
    return path

def prepare(self):
    self.initialize_classifiers()

def set_parameters(self, epochs, conf_thresh, sky_window, neg_window, nms_thresh):
    self._EPOCHS = epochs
    self._ANN_CONF_THRESHOLD = conf_thresh
    self._SKY_WINDOW = sky_window
    self._NEG_WINDOW = neg_window
    self._NMS_OVERLAP_THRESHOLD = nms_thresh

def set_architecture(self, clusters, color_bins, inner_layers):
    self._BOW_CLUSTERS = clusters
    self._COLOR_BINS = color_bins
    input_layers = int(self._BOW_CLUSTERS + self._COLOR_BINS*3)
    layers = [input_layers]
    layers.extend(inner_layers)
    layers.append(NUM_CLASSES)
    print(layers)
    self._ANN_LAYERS = layers
```

Here is a helper function and a wrapper for initializing all the different machine learning objects needed for the program like the BOW extractor and the ANN. There are also two setter functions for the driver file or other classes to use for easy parameter changing.

```python
    def initialize_classifiers(self):
        if not os.path.isdir(DATA_PATH):
            print('data not found')
            exit(1)
        else:
            self._sift = cv.xfeatures2d.SIFT_create()
            index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=9)
            search_params = {}
            self._flann = cv.FlannBasedMatcher(index_params, search_params)
            self._bow_extractor = cv.BOWImgDescriptorExtractor(self._sift, self._flann)

            self._ann = cv.ml.ANN_MLP_create()

            if os.path.exists(VOCAB_PATH):
                print('Loading vocab...')
                self.load_vocab(VOCAB_PATH)
            else:
                print('Reclustering...')
                self.prepare_vocab()

            self.train()
            self._READY = True
            print("CLASSIFIER READY")
```

Here the feature detector and feature matcher are initialized for the BOW extractor along with the initialization of the ann. Then the vocab is prepared and the ANN is trained on it

```python
    def load_vocab(self, path):
        self._vocab = np.load(path)
        self._bow_extractor.setVocabulary(self._vocab)

    def extract_bow_descriptors(self, img):
        features = self._sift.detect(img)
        return self._bow_extractor.compute(img, features)

    def add_sample_bow(self, path):
        #print("Sampling: ", path)
        gray = cv.imread(path, cv.IMREAD_GRAYSCALE)
        gray.astype('uint8')
        keypoints, descriptors = self._sift.detectAndCompute(gray, None)
        if descriptors is not None:
            self._bow_kmeans_trainer.add(descriptors)

    def prepare_vocab(self):
        print("Preparing vocab for BOW Extractor...")
        self._bow_kmeans_trainer = cv.BOWKMeansTrainer(self._BOW_CLUSTERS)

        for class_name in CLASSES:
            for i in range(BOW_NUM_TRAINING_SAMPLES_PER_CLASS):
                path = self.get_path_data(class_name, i+1)
                self.add_sample_bow(path)

        self._vocab = self._bow_kmeans_trainer.cluster()
        self._bow_extractor.setVocabulary(self._vocab)
        np.save(VOCAB_PATH,self._vocab)
        print("Saving vocab for later...")
```

Here are some more helper functions for loading vocabulary, extracting BOW descriptors, adding a sample to the BOW extractor, and preparing the vocab if needed.

```
    def return_hists(self,roi):
        bgr_planes = cv.split(roi)
        c = []
        blue = cv.calcHist(bgr_planes,
                           [0],
                           None,
                           [self._COLOR_BINS],
                           (0,256),
                           accumulate=False)
        green = cv.calcHist(bgr_planes,
                           [1],
                           None,
                           [self._COLOR_BINS],
                           (0,256),
                           accumulate=False)
        red = cv.calcHist(bgr_planes,
                           [2],
                           None,
                           [self._COLOR_BINS],
                           (0,256),
                           accumulate=False)
        c.extend(blue)
        c.extend(green)
        c.extend(red)
        c = np.array(c)
        c = c.flatten()
        #normal_c = (c-np.min(c))/(np.max(c)-np.min(c))
        length = np.linalg.norm(c)
        normal_c = c/length
        return normal_c
```

This helper function is used for taking a histogram along each channel of a ROI and turning it into a one dimensional normalized vector encoding all of that information for use in the ANN later.

```
    def get_combined_input(self,roi,descriptors):
        colors = np.array(self.return_hists(roi))
        combined_sample = []
        combined_sample.extend(descriptors)
        combined_sample.extend(colors)
        return np.array(combined_sample,np.float32)
```

This helper function takes in the BOW descriptors and the color vector and combines them into a one dimensional vector for use as input to the ANN.

```
    def train(self):
        print('Training ANN on vocab...')
        self._ann.setLayerSizes(np.array(self._ANN_LAYERS))
        self._ann.setActivationFunction(cv.ml.ANN_MLP_SIGMOID_SYM, 0.6, 1.0)
        self._ann.setTrainMethod(cv.ml.ANN_MLP_BACKPROP, 0.07, 0.07)
        self._ann.setTermCriteria(
            (cv.TERM_CRITERIA_MAX_ITER | cv.TERM_CRITERIA_EPS, 100, 1.0))

        samples = []
        labels = []

        if os.path.exists(TRAINING_SAMPLES) and os.path.exists(TRAINING_LABELS):
            print("Loading existing records")
            samples = np.load(TRAINING_SAMPLES)
            labels = np.load(TRAINING_LABELS)
        else:
            print("Retaking descriptors for records...")
            for class_id in range(NUM_CLASSES):
                class_name = CLASSES[class_id]
                for i in range(ANN_NUM_TRAINING_SAMPLES_PER_CLASS):
                    path = self.get_path_data(class_name, i)
                    current = cv.imread(path)
                    descriptors = self.extract_bow_descriptors(current)
                    if descriptors is None:
                        continue
                    combined_sample = self.get_combined_input(current,descriptors[0])
                    samples.append(combined_sample)
                    labels.append([class_id])
            samples = np.array(samples,np.float32)
            labels = np.array(labels,np.float32)

            np.save(TRAINING_SAMPLES,samples)
            np.save(TRAINING_LABELS,labels)

            print("Records saved...")
```

This is part of the function used to train the ANN, all of the combined input vectors are taken from each file in the dataset. Training samples and labels are created and saved.

```python

        for e in range(self._EPOCHS):
            for sample, class_id in zip(samples, labels):
                sample = np.array(sample,np.float32)
                identity = np.array(np.zeros(NUM_CLASSES),np.float32)
                identity[int(class_id)] = 1.0
                data = cv.ml.TrainData_create(sample, cv.ml.COL_SAMPLE, identity)
                if self._ann.isTrained():
                    self._ann.train(
                        data,
                        cv.ml.ANN_MLP_UPDATE_WEIGHTS | cv.ml.ANN_MLP_NO_OUTPUT_SCALE)
                else:
                    self._ann.train(
                        data,
                        cv.ml.ANN_MLP_NO_INPUT_SCALE | cv.ml.ANN_MLP_NO_OUTPUT_SCALE)

        print("ANN ready")
```

Then the ANN is trained based on the EPOCHS macro for a number of epochs.

```python
    def detect_and_classify(self, img, inputPath):
        if self._READY:
            print("Detecting and Classifying ", inputPath)
            original_img = cv.imread(inputPath)
            pos_rects = []
            for resized in self.pyramid(img):
                scale = original_img.shape[0] / float(resized.shape[0])
                print("scale: ", resized.shape)
                for x, y, roi in self.sliding_window(resized):
                    descriptors = self.extract_bow_descriptors(roi)
                    if descriptors is None:
                        continue
                    combined_input = self.get_combined_input(roi,descriptors[0])
                    prediction = self._ann.predict(np.array([combined_input]))
                    class_id = int(prediction[0])
                    confidence = prediction[1][0][class_id]
                    sky_conf = prediction[1][0][1]
                    neg_conf = prediction[1][0][0]
                    if ( confidence > self._ANN_CONF_THRESHOLD
                         and sky_conf < self._SKY_WINDOW[1]
                         and sky_conf > self._SKY_WINDOW[0]
                         and neg_conf < self._NEG_WINDOW[1]
                         and neg_conf > self._NEG_WINDOW[0] ):
                        h, w, channels = roi.shape
                        pos_rects.append(
                            [int(x * scale),
                             int(y * scale),
                             int((x+w) * scale),
                             int((y+h) * scale),
                             confidence,
                             sky_conf,
                             neg_conf,
                             class_id])
```

This is the main functionality of the program. The input image is resized into a pyramid and for each image in the pyramid a sliding window goes along it, taking a combined input vector from the window then having the ANN get a prediction. IF the prediction meets a threshold and certain tolerances for the negative classes, then it is taken as a positive classification.

Then the positive rectangle’s coordinates and information are added to a list.

```
                             class_id])
            pos_rects = nms(np.array(pos_rects), self._NMS_OVERLAP_THRESHOLD)
            counts = np.zeros(NUM_CLASSES, dtype=float)
            for x0, y0, x1, y1, score, sky_conf, neg_conf, class_id in pos_rects:
                cv.rectangle(original_img, (int(x0), int(y0)), (int(x1), int(y1)),
                              (10, 220, 255), 4)
                text = CLASSES[int(class_id)] + ' ' \
                    + ('%.2f' % score) + ' ' + ('%.2f' % sky_conf) \
                    + ' ' + ('%.2f' % neg_conf)
                counts[int(class_id)] += (float(x1-x0)) #* score)
                cv.putText(original_img, text, (int(x0), int(y0) + 20),
                            cv.FONT_HERSHEY_SIMPLEX, 1, (10, 220, 255), 4)
            predominant_id = 0
            current_max = 0.0
            for i in range(NUM_CLASSES):
                if counts[i] > current_max:
                    current_max = counts[i]
                    predominant_id = i
            print("PREDOMINANT: ", CLASSES[predominant_id])
            print("COUNTS: ", counts)
            predominant_text = "PREDOMINANT: " + \
                               CLASSES[int(predominant_id)] + ": " + \
                               ('%.2f' % current_max)
            cv.putText(original_img, predominant_text, (20,120),
                       cv.FONT_HERSHEY_SIMPLEX, 2, (20, 255, 50), 4)
            return original_img, int(predominant_id)
        else:
            print("not trained")
            exit(1)

```

Then the positive rectangles are filtered by the NMS algorithm.
‘

Then every final positive rectangle is looped through and drawn onto the output image along with the confidence for the class, and confidences for the negative classes. A count of the area (using side length since it is a proportion of area and every sliding window has the same proportions) is used to keep an area score of each cloud type in the photo and the maximum is taken and returned as the predominant cloud type along with the output image.

```python
    def sliding_window(self, img, step=10, window_size=(75, 50)):
        img_h, img_w, channels = img.shape
        window_w, window_h = window_size
        for y in range(0, img_w, step):
            for x in range(0, img_h, step):
                roi = img[y:y+window_h, x:x+window_w]
                roi_h, roi_w, channels = roi.shape
                if roi_w == window_w and roi_h == window_h:
                    yield (x, y, roi)

    def pyramid(self, img, scale_factor=1.2, min_size=(200, 200),
                max_size=(700, 700)):
        h, w, channels = img.shape
        min_w, min_h = min_size
        max_w, max_h = max_size
        while w >= min_w and h >= min_h:
            if w <= max_w and h <= max_h:
                yield img
            w /= scale_factor
            h /= scale_factor
            img = cv.resize(img, (int(w), int(h)),
                            interpolation=cv.INTER_AREA)
```

Lastly here are the helper functions for resizing the input and sliding a window across.

Now lets look at the “tester.py” file:

File Edit Format Run Options Window Help

```python
import cv2 as cv
import numpy as np
import os
import sys
import cloudclassifyANNColor as cc
import itertools

CLUSTERS = 21
COLOR_BINS = 28 # per color channel
HIDDEN_LAYERS = [75]

EPOCHS = 300
CONF_THRESH = 0.8
SKY_WINDOW = -0.12, 0.12
NEG_WINDOW = -0.12, 0.12

NMS_THRESH = 0.18

NUM_TESTS = 15
ANN_CLASSES = ['NEG','Sky','Cumulus','Cirrus','Stratus']
TEST_CLASSES = ['Cirrus','Cumulus','Stratus']
TEST_LOCATION = '../../Data/TestPhotos/TESTS/'
OUTPUT_LOCATION = '../../Data/Outputs/'

```

All of these macros are used to change parameters of the instance of cloudclassifyANNColor. Changing them yields difference results in accuracy for different classes and also speed.

```python
cloud = cc.CloudClassify()
cloud.set_parameters(
    epochs = EPOCHS,
    conf_thresh = CONF_THRESH,
    sky_window = SKY_WINDOW,
    neg_window = NEG_WINDOW,
    nms_thresh = NMS_THRESH)
cloud.set_architecture(CLUSTERS, COLOR_BINS, HIDDEN_LAYERS)
cloud.prepare()
```

Here the instance of the cloud classifier has its parameters and architecture set then it is prepared for running.

```python

def test_class(class_name):
    obstructed_accuracy = 0.0
    o_total = 0.0
    unobstructed_accuracy = 0.0
    u_total = 0.0
    for i in range(NUM_TESTS):
        print("Testing ", class_name, " ", i+1)
        u_file = "UNOBSTRUCTED/" + class_name + str(i+1) + ".JPG"
        o_file = "OBSTRUCTED/" + class_name + str(i+1) + ".JPG"
        u_path = TEST_LOCATION + u_file
        o_path = TEST_LOCATION + o_file
        #print("Testing ",u_path)
        u_output, u_predominant = cloud.run(u_path)
        if (ANN_CLASSES[u_predominant] == class_name):
            u_total += 1.0
        o_output, o_predominant = cloud.run(o_path)
        if (ANN_CLASSES[o_predominant] == class_name):
            o_total += 1.0
        cv.imwrite(OUTPUT_LOCATION+u_file, u_output)
        cv.imwrite(OUTPUT_LOCATION+o_file, o_output)
    obstructed_accuracy = o_total / NUM_TESTS
    unobstructed_accuracy = u_total / NUM_TESTS
    print("Obstructed accuracy: ", obstructed_accuracy)
    print("Unobstructed accuracy: ", unobstructed_accuracy)
    return (obstructed_accuracy, unobstructed_accuracy)

```

This is a helper function for testing all of the obstructed and unobstructed test photos for a given testing class.

```

accuracies = []

for class_name in TEST_CLASSES:
    un, ob = test_class(class_name)
    accuracies.append([un, ob])

print()
print("Parameters: ")
print("BOW Clusters: ", CLUSTERS)
print("Bins per channel: ", COLOR_BINS)
print("Hidden layers: ", HIDDEN_LAYERS)
print("Epochs: ", 300)
print("Confidence threshold: ",  CONF_THRESH)
print("Sky tolerances: ", SKY_WINDOW)
print("Negative tolerances: ", NEG_WINDOW)
print("NMS Thresh: ", NMS_THRESH)
print()
print()

print("Final Accuracies: ")
print()
for accs, class_name in zip(accuracies, TEST_CLASSES):
    print(class_name, " unobstructed: ", accs[0])
    print(class_name, " obstructed: ", accs[1])
    print()

```

Finally each class is tested and the results and information are output to the console.

Again, deleting the .npy files will allow you to change certain parameters and recluster and retake training samples. Once the necessary npy files are already there, the is much less overhead and the program goes right into getting results. If there is an problem in running the program, try deleting the npy files and let it retrain itself.