Computer vision

Project Phase 2

Prepared By:

Ibrahim Ashraf Ibrahim 19P7817

AbdulRahman Ayman Mahmoud Abbas ElSherbiny 19P6458

Sohayla Ihab Abdelmawgoud Ahmed Hamed 19P7343

Contents

GitHub link: https://github.com/PerfectionistAF/SVHN-Recognition.git

1.0 The methods we used through the project

1-loadDataSet(file_path: str)

2-NormalizeImage(img)

3-ExtractFeatures(img)

4-MatchFeatures(img1, img2)

5-testImages(v=False)

2.0 Here we will explain each function used in the code in details :

2.1 The second function used is loadDataSet(file_path: str):

Here it's code:

```
def loadDataSet(file_path: str):
    # Open the file in read-only mode
    f = open(file_path, 'r')
    # Load the contents of the file as JSON data
    data = json.load(f)
    # Return the loaded data
    return data
```

2.1.1 Explanation of the function :

This code defines a function loadDataSet that takes a file_path string as input and returns the JSON data loaded from the file located at that path. Here's a detailed explanation of how the function works:

1-The first line of code opens the specified file in read-only mode using the built-in open() function, and assigns the resulting file object to the variable f.

2-The second line of code loads the contents of the file as JSON data using the json.load() function, which reads the file object f and returns a Python object representing the parsed JSON data.

3-The third line of code assigns the loaded JSON data to the variable data.

4-Finally, the function returns the loaded data by returning the value of the data variable.

Note that if the specified file does not exist or there is an error reading from it, this code will raise an exception. It is good practice to surround the file operations with a try-except block to handle such errors gracefully.

2.2 The second function used is NormalizeImage(img)

Here it's code:

```
def NormalizeImage(img):

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    rgbPlanes = cv2.split(gray)

    normalizedPlanes = []
    for plane in rgbPlanes:

        dilatedImage = cv2.dilate(plane, np.ones((7, 7), np.uint8))

        blurredImage = cv2.medianBlur(dilatedImage, 21)

        planeDifferenceImage = 255 - cv2.absdiff(plane, blurredImage)

        normalizedImage = cv2.normalize(planeDifferenceImage, None, alpha=0,
      beta=255, norm_type=cv2.NORM_MINMAX,
                                        dtype=cv2.CV_8UC1)

        normalizedPlanes.append(normalizedImage)

    normalizedResult = cv2.merge(normalizedPlanes)

    return normalizedResult
```

2.2.1 Explanation of the function :

The function NormalizeImage takes an image as input and returns the normalized version of it. Here is a step-by-step explanation of what the code does:

Convert the input image from BGR to grayscale using cv2.cvtColor function.

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

This is done so that we can split the image into its RGB planes in the next step.

Split the grayscale image into its RGB planes using cv2.split function.

rgbPlanes = cv2.split(gray)

This is done so that we can apply normalization to each plane separately.

Apply dilation, median filtering, and absolute difference operations to each RGB plane separately, and then normalize the result using cv2.normalize function.

a. For each plane, apply dilation operation using cv2.dilate function with a 7x7 kernel size.

dilatedImage = cv2.dilate(plane, np.ones((7, 7), np.uint8))

b. Then, apply median filtering to the dilated image using cv2.medianBlur function with a kernel size of 21.

blurredImage = cv2.medianBlur(dilatedImage, 21)

c. Calculate the absolute difference between the original plane and the blurred image using cv2.absdiff function, and subtract the result from 255 to obtain the plane difference image.

planeDifferenceImage = 255 - cv2.absdiff(plane, blurredImage)

d. Finally, normalize the plane difference image using cv2.normalize function with alpha=0, beta=255, and NORM_MINMAX normalization type. The resulting image is stored in the normalizedImage variable.

normalizedImage = cv2.normalize(planeDifferenceImage, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8UC1)

Store the normalized RGB planes in a list called normalizedPlanes.

normalizedPlanes.append(normalizedImage)

Merge the normalized RGB planes into a single image using cv2.merge function and return the result.

normalizedResult = cv2.merge(normalizedPlanes)

return normalizedResult

This resulting image will have enhanced contrast and brightness, which can be useful for image processing tasks such as object detection or recognition.

2.3 The Third function used is ExtractFeatures(img):

Here it's code:

```
def ExtractFeatures(img):
    sift = cv2.SIFT_create()

    try:
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    except:
        keyPoints, descriptors = sift.detectAndCompute(img, None)
        return keyPoints, descriptors
    keyPoints, descriptors = sift.detectAndCompute(gray, None)
    return keyPoints, descriptors
```

2.3.1 Explanation of the function :

The function named ExtractFeatures that takes an image (img) as input and returns the keypoints and descriptors detected using the Scale-Invariant Feature Transform (SIFT) algorithm.

The first line of the function creates an instance of the SIFT class from OpenCV's feature detection module. This is done using the cv2.SIFT_create() method.

Next, the function tries to convert the input image to grayscale using OpenCV's cv2.cvtColor() method with the flag cv2.COLOR_BGR2GRAY. If the conversion is unsuccessful (for example, if the input image is already in grayscale), then an exception is raised and the keyPoints and descriptors are computed directly from the input image without converting it to grayscale.

If the conversion is successful, the next line detects the keypoints and computes the descriptors using the SIFT algorithm on the grayscale image. The detectAndCompute() method of the SIFT object is used for this purpose, with the grayscale image passed as the first argument and None passed as the second argument (which means that no mask is used).

Finally, the function returns the detected keypoints and their corresponding descriptors as a tuple (keyPoints, descriptors).

2.4 The fourth function used is MatchFeatures(img1, img2, v=False):

Here it's code:

```
def MatchFeatures(img1, img2, v=False):
    try:
        bruteForceMatcher = cv2.BFMatcher()

        keyPoints1, descriptors1 = ExtractFeatures(img1)
        keyPoints2, descriptors2 = ExtractFeatures(img2)

        matches = bruteForceMatcher.knnMatch(descriptors1, descriptors2, k=2)

        optimizedMatches = []
        for firstImageMatch, secondImageMatch in matches:
            if firstImageMatch.distance < 1 * secondImageMatch.distance:
                optimizedMatches.append(firstImageMatch)

        similarity_scores = [match.distance for match in optimizedMatches]
        max_distance = max(similarity_scores)
        min_distance = min(similarity_scores)
        normalized_scores = [(max_distance - score) / ((max_distance -
min_distance) + 0.0000001) for score in
                             similarity_scores]
        matched_image = cv2.drawMatches(img1, keyPoints1, img2, keyPoints2,
optimizedMatches, None,

flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

        if v:
            cv2.imshow('Digit', matched_image)
            cv2.waitKey(0)
            cv2.destroyAllWindows()

        return sum(normalized_scores) / len(normalized_scores)
    except:
        return math.inf
```

2.4.1Explanation of the function :

The purpose of this code is to match features between two images and compute a similarity score based on the matched features.

Here's a step-by-step breakdown of what's happening:

A brute-force matcher is created using the OpenCV library. This will be used later on to perform matching between the descriptors extracted from the two images.

```
bruteForceMatcher = cv2.BFMatcher()
```

The key points and descriptors are extracted from both images using a function called ExtractFeatures. This function is not included in the code provided, but it's safe to assume that it uses some sort of feature detection algorithm (such as SIFT or SURF) to identify key points in the image and extract descriptors for those points.

```
keyPoints1, descriptors1 = ExtractFeatures(img1)
keyPoints2, descriptors2 = ExtractFeatures(img2)
```

Using the brute-force matcher created earlier, the descriptors from both images are matched and a list of matches is returned. Each match consists of two "nearest neighbor" matches, and the ratio test is used to filter out ambiguous matches.

```
matches = bruteForceMatcher.knnMatch(descriptors1, descriptors2, k=2)
optimizedMatches = []
for firstImageMatch, secondImageMatch in matches:
    if firstImageMatch.distance < 1 * secondImageMatch.distance:
        optimizedMatches.append(firstImageMatch)
```

After filtering out ambiguous matches, the similarity scores are computed based on the distances between the matched descriptors. The scores are then normalized between 0 and 1 using the minimum and maximum distance values.

```
similarity_scores = [match.distance for match in optimizedMatches]
max_distance = max(similarity_scores)
min_distance = min(similarity_scores)
normalized_scores = [(max_distance - score) / ((max_distance - min_distance) + 0.0000001) for score in similarity_scores]
```

If the v parameter is set to True, the matched key points are drawn on the image and displayed.

```
matched_image = cv2.drawMatches(img1, keyPoints1, img2, keyPoints2, optimizedMatches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

if v:

cv2.imshow('Digit', matched_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

Finally, the average normalized score is computed and returned as a measure of similarity between the two images. If an exception occurs during any of these steps, the function returns infinity.

return sum(normalized_scores) / len(normalized_scores)

2.5 The fifth function used is testImages(v=False):

Here it's code:

```
def testImages(v=False):
    print("Loading DataSet File..")
    dataSet = loadDataSet('training.json')
    print("DataSet File Loaded!!\n")
    accuracy = []
    total = len(os.listdir("testImages"))
    i = 0
    for filename in os.listdir("testImages"):
        i += 1
        loadPercent = (i / total) * 100
        if not v:
            sys.stdout.write(
                f"\rComputing Accuracy: [{'=' * math.floor(loadPercent / 10)}{' ' * (10 -
math.floor(loadPercent / 10))}]"
                f"{round(loadPercent, 1)}%")

        imgReal = cv2.imread(os.path.join("testImages", filename))
        boxes = dataSet[int(filename.split(".")[0]) - 1]['boxes']
        normalizedImage = NormalizeImage(imgReal)

        for idxBox, box in enumerate(boxes):
            (x, y, w, h) = int(box['left']), int(box['top']), int(box['width']),
int(box['height'])
            img = imgReal[y:y+h, x:x+w]
            label = str(box['label']).split(".")[0]
            digit = ""
            score = math.inf
            for idx, digitFilename in enumerate(os.listdir("digitTemplates")):
                template = os.path.join("digitTemplates", digitFilename)
                digitTemplate = cv2.imread(template)

                desired_height = img.shape[0]
                aspect_ratio = digitTemplate.shape[1] / digitTemplate.shape[0]
                desired_width = int(desired_height * aspect_ratio)
                resized_image = cv2.resize(digitTemplate, (desired_width, desired_height))

                sim = MatchFeatures(resized_image, normalizedImage[y:y+h, x:x+w], v)
                if sim < score:
                    score = sim
                    digit = digitFilename.split(".")[0]

            if v:
                image = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                aspect_ratio = image.shape[1] / image.shape[0]
                image = cv2.resize(image, (int(500 * aspect_ratio), 500))
                image = cv2.putText(image, digit, (image.shape[1] // 2, image.shape[0] // 2),
cv2.FONT_HERSHEY_SIMPLEX,
                                    3, (0, 255, 0), 5, cv2.LINE_AA)
                cv2.imshow('Digit', image)
                cv2.waitKey(0)
                cv2.destroyAllWindows()
                print(f"\nImage {filename.split('.')[0]}, Box:{idxBox}: Label = {label},"
                      f" Predicted Outcome = {digit}")

            accuracy.append(digit == label)

    if not v:
        sys.stdout.write(f"\rComputing Accuracy: [{'=' * 10}] 100%")
    return sum(accuracy)/len(accuracy)
```

2.5.1 Explanation of the function :

This function is used to test the accuracy of a digit recognition algorithm on a dataset of test images. Here's a step-by-step breakdown of what's happening:

The loadDataSet function is called to load the dataset file (training.json) and store it in the dataSet variable.

```
dataSet = loadDataSet('training.json')
```

A list called accuracy is initialized to store the accuracy of the predicted digits for each image in the test dataset.

```
accuracy = []
```

The function iterates through each image in the "testImages" folder and extracts the bounding boxes for that image from the dataSet.

```
for filename in os.listdir("testImages"):
    # Read the real image
    imgReal = cv2.imread(os.path.join("testImages", filename))
    # Get the bounding boxes for the current image
    boxes = dataSet[int(filename.split(".")[0]) - 1]['boxes']
```

For each bounding box in the current image, the function extracts the region of interest (ROI) from the real image based on the bounding box.

```
(x, y, w, h) = int(box['left']), int(box['top']), int(box['width']), int(box['height'])
img = imgReal[y:y+h, x:x+w]
```

The label for the current digit is extracted from the box dictionary and the predicted digit is initialized to an empty string.

```
label = str(box['label']).split(".")[0]
digit = ""
```

A loop iterates over all digit templates in the "digitTemplates" directory and loads each template image.

```
for idx, digitFilename in enumerate(os.listdir("digitTemplates")):
    # Load the digit template image
    template = os.path.join("digitTemplates", digitFilename)
    digitTemplate = cv2.imread(template)

    # Resize the digit template to match the size of the ROI
```

```
    desired_height = img.shape[0]
    aspect_ratio = digitTemplate.shape[1] / digitTemplate.shape[0]
    desired_width = int(desired_height * aspect_ratio)
    resized_image = cv2.resize(digitTemplate, (desired_width, desired_height))
```

For each digit template, the function matches features between the resized digit template and the normalized ROI using the MatchFeatures function. The predicted digit is updated with the digit template that produces the best match.

```
sim = MatchFeatures(resized_image, normalizedImage[y:y+h, x:x+w], v)
if sim < score:
    # Update the predicted digit if a better match is found
    score = sim
    digit = digitFilename.split(".")[0]
```

If v is true, the predicted digit is displayed on the ROI image along with the label (if enabled).

```
if v:
    # Display the predicted digit on the ROI image (if enabled)
    image = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    aspect_ratio = image.shape[1] / image.shape[0]
    image = cv2.resize(image, (int(500 * aspect_ratio), 500))
    image = cv2.putText(image, digit, (image.shape[1] // 2, image.shape[0] // 2),
cv2.FONT_HERSHEY_SIMPLEX,
                        3, (0, 255, 0), 5, cv2.LINE_AA)
    cv2.imshow('Digit', image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    print(f"\nImage {filename.split('.')[0]}, Box:{idxBox}: Label = {label},"
          f" Predicted Outcome = {digit}")
```

The accuracy of the predicted digit is computed by comparing it with the label for the current digit. The accuracy is stored in the accuracy list.

```
accuracy.append(digit == label)
```

After all images and bounding boxes have been processed, the function computes the overall accuracy of the algorithm by calculating the ratio of correct predictions to total predictions.

```
return sum(accuracy)/len(accuracy)
```

2.6 Generally what does the function does :

The function testImages loads a dataset file and a set of test images containing digits, and performs digit recognition by comparing each digit in each image with a set of pre-defined digit templates. It does this by first normalizing the test image using a combination of dilation, median blur, and absolute difference operations. Then, it extracts features (key points and descriptors) from both the normalized test image and each digit template, and matches these features using a brute-force matcher with ratio test filtering. The function computes similarity scores based on the distances between matched descriptors, and returns the average normalized score as a measure of similarity. Finally, the function compares the predicted digit with the actual label for each bounding box in each test image, and computes the accuracy of the digit recognition model. The function also has an optional parameter to display the predicted digit on the ROI image for each bounding box.

3.0 Output snippets: