Optical Character Recognition (OCR) and Text Detection with Tesseract and EAST

Name: Yu Liwei

ID: 3160576

Introduction

In this project, I worked on Optical Character Recognition (OCR) using Python, along with the Tesseract OCR engine and the EAST text detection model. The idea behind this project was to create a system that can read an image, find where the text is in the image, and then use OCR to extract that text. The extracted text is saved to a file, and the processed image is displayed, showing the areas where text was found.

Libraries and Packages Used

This project required several Python libraries, each with a specific purpose. These libraries made it easier to manipulate and process the images, detect the areas with text, and extract the text using OCR.

1. OpenCV (cv2)

OpenCV is one of the most powerful and widely-used libraries for real-time image processing and computer vision. In this project, OpenCV played a key role. I used it for loading the images into the program, converting them to grayscale, applying various filters to enhance the image quality, and drawing rectangles around the areas where text was detected. OpenCV also has built-in functions for image resizing and transformations, which were essential for preparing the images before running the text detection and OCR steps.

2. NumPy (np)

NumPy is another key library in this project, mainly used for handling the pixel data of the images. In Python, images are represented as arrays of pixel values, and NumPy makes it easy to manipulate and process these arrays. For example, I used NumPy to sharpen the image by applying custom filters (also known as kernels) that enhance the edges of the text. NumPy was also used to normalize the image's pixel values, which helped improve the contrast and make the text stand out more clearly from the background.

3. Pytesseract

Pytesseract is the Python wrapper for Google’s Tesseract OCR engine, which is the main tool I used for recognizing and extracting text from the images. Tesseract itself is a well-established OCR engine that supports a wide range of languages, including English and Chinese, which was important for this project since I needed to extract multilingual text. Pytesseract supports various configurations, like specifying the page segmentation mode, which determines how Tesseract treats the layout of the text.

4. os

The os library is a standard Python library used for interacting with the file system. In this project, I used it to create directories for storing the extracted text files, manage file paths, and ensure that the results from each image were saved in an organized way.

Image Preprocessing

Image preprocessing is a critical step in OCR and text detection. The quality of the input image directly impacts the performance of the OCR engine, so it's important to clean up the image and make the text as clear and distinct as possible.

1. Grayscale Conversion

The first step in preprocessing was converting the image to grayscale. Since color is not important for text detection, converting the image to grayscale simplifies the data and makes the following steps more efficient. By focusing only on the intensity of the pixels, I could reduce the amount of data that needed to be processed.

Here's the code for converting an image to grayscale:

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

The `cvtColor` function in OpenCV takes the original image, which is in BGR (Blue, Green, Red) format, and converts it into a grayscale image. This step is essential because it removes unnecessary information (color) and leaves only the brightness of each pixel, which is all that's needed for text detection.

2. Sharpening the Image

Once the image was converted to grayscale, I applied a sharpening filter to enhance the edges in the image. Sharpening is particularly useful for making text more readable, as it increases the contrast between the text and the background, making the letters stand out more clearly.

Here's how I applied a sharpening filter using a convolution kernel:

```
kernel_sharpen = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
sharpened = cv2.filter2D(gray, -1, kernel_sharpen)
```

In this code, the `filter2D` function from OpenCV applies a sharpening kernel to the grayscale image. The kernel I used is a 3x3 matrix that emphasizes the differences between neighboring pixels, effectively enhancing the edges in the image.

3. Contrast Stretching

After sharpening the image, I applied contrast stretching to further highlight the text. Contrast stretching is a technique that increases the difference between the light and dark areas of the image, making the text stand out more clearly against the background. This step is especially helpful for images with poor contrast, where the text might be hard to distinguish.

Here's the code for contrast stretching:

```
contrast_stretched = cv2.normalize(sharpened, None, alpha=0, beta=255,
norm_type=cv2.NORM_MINMAX)
```

The `normalize` function in OpenCV adjusts the pixel values of the image, mapping them to a new range (in this case, from 0 to 255). By normalizing the image in this way, I was able to improve the contrast and make the text more visible.

4. Denoising with Bilateral Filtering

One of the biggest challenges in OCR is dealing with noise in the image. Noise can come from many sources, such as low-quality scans, lighting conditions, or complex backgrounds. To reduce the noise without losing important details like the edges of the text, I applied bilateral filtering.

Bilateral filtering is a technique that smooths the image while preserving edges, making it ideal for text detection.

Here's the code for applying bilateral filtering:

```
denoised_image = cv2.bilateralFilter(contrast_stretched, d=9, sigmaColor=75,
sigmaSpace=75)
```

In this code, the `bilateralFilter` function takes several parameters: `d` controls the size of the filter, and `sigmaColor` and `sigmaSpace` control the degree of smoothing. The result is a cleaner image with less noise, but the edges of the text are still sharp and clear.

5. Thresholding (Binarization)

The final preprocessing step was thresholding, which converts the image into a binary (black-and-white) format. This makes the text even easier to recognize by creating a clear separation between the text (black) and the background (white). For this project, I used adaptive thresholding, which calculates a different threshold value for each small region of the image, allowing for better results on images with uneven lighting.

Here's the code for adaptive thresholding:

```
binary_image = cv2.adaptiveThreshold(
    denoised_image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11,
2)
```

In this code, the `adaptiveThreshold` function takes the denoised image and converts it to a binary format. The `ADAPTIVE_THRESH_GAUSSIAN_C` method calculates the threshold for small regions of the image, and the `THRESH_BINARY` flag ensures that the output is either black or white.

Text Detection Using the EAST Model

Once the image was preprocessed, the next step was to detect the regions of the image that contain text. For this, I used the EAST (Efficient and Accurate Scene Text Detector) model, which is a deep learning-based approach to real-time text detection.

The EAST model is highly effective because it not only detects the location of text but also provides the orientation of the text. This makes it possible to detect text in complex scenes, such as images with rotated or curved text.

1. Loading the EAST Model

The EAST model is provided as a pre-trained model in the form of a frozen TensorFlow graph. I loaded this model into the program using OpenCV's `dnn` module, which allows us to work with deep learning models.

Here's the code for loading the EAST model:

```
net = cv2.dnn.readNet("frozen_east_text_detection.pb")
```

In this code, the `readNet` function loads the pre-trained EAST model from a file. The model is designed to detect text areas in images and works well in both simple and complex scenes.

2. Resizing the Image

Before passing the image to the EAST model, I needed to resize it to a fixed size. The EAST model expects the input image to have specific dimensions (320x320 pixels in this case), so resizing ensures that the input is compatible with the model.

Here's the code for resizing the image:

```
newW, newH = (320, 320)
image = cv2.resize(image, (newW, newH))
```

This step ensures that the image fits the model's input requirements. After resizing, I also calculated scaling factors to later map the detected regions back to the original image size.

3. Creating an Input Blob

To pass the image into the model, I needed to convert it into a format that the neural network could process. This is done by creating a blob, which is a 4D array that holds the image data in a way that the model can understand.

Here's the code for creating an input blob:

```
blob = cv2.dnn.blobFromImage(image, 1.0, (W, H), (123.68, 116.78, 103.94),
swapRB=True, crop=False)
```

The `blobFromImage` function takes the resized image and converts it into a blob. The function also normalizes the pixel values by subtracting the mean values for the Red, Green, and Blue channels (123.68, 116.78, and 103.94, respectively).

4. Forward Pass to Get Text Scores and Geometry

The EAST model produces two outputs: `scores` and `geometry`. The `scores` represent the probability that a given region contains text, while the `geometry` provides the bounding box coordinates for those regions.

Here's the code for running a forward pass through the network:

```
layerNames = ["feature_fusion/Conv_7/Sigmoid", "feature_fusion/concat_3"]
(scores, geometry) = net.forward(layerNames)
```

In this code, I specified the output layers of the network, which return the text scores and geometry. The `scores` give a confidence value for each region, and the `geometry` provides the coordinates for drawing bounding boxes around the detected text.

5. Extracting Text Boxes

Using the confidence scores, I filtered out regions that had a low probability of containing text. For this project, I only kept regions with a confidence score of 0.5 or higher.

Here's the code for filtering text boxes:

```
if scoresData[x] < 0.5:
    continue
```

This simple `if` statement ensures that only the most likely text regions are kept. This helps reduce false positives and improves the overall accuracy of the text detection process.

6. Non-Maximum Suppression

After detecting the text regions, I applied Non-Maximum Suppression (NMS) to reduce overlapping bounding boxes. NMS selects the best bounding box for each region, ensuring that each text area is only represented by a single box.

Here's the code for Non-Maximum Suppression:

```
boxes = cv2.dnn.NMSBoxes(rectangles, confidences, score_threshold=0.4,
nms_threshold=0.3)
```

The `NMSBoxes` function takes the list of rectangles (bounding boxes) and their confidence scores and applies NMS to remove overlapping boxes. This step helps clean up the results and ensures that each detected text area is represented by a single, accurate bounding box.

Optical Character Recognition (OCR) with Tesseract

Once the text regions were detected, the next step was to use Tesseract to recognize and extract the text from these regions. Tesseract is a powerful OCR engine that supports multiple languages, making it ideal for this project.

1. Extracting Regions of Interest (ROI)

For each detected text region, I cropped the image to isolate the Region of Interest (ROI). This allows Tesseract to focus on a specific part of the image, improving the accuracy of the OCR.

Here's the code for extracting the ROI:

```
roi = denoised_image[startY:endY, startX:endX]
```

This simple operation crops the image to focus on the detected text region. By feeding only the ROI into Tesseract, I was able to improve the accuracy of the OCR and reduce the chances of misrecognizing non-text areas.

2. Performing OCR

With the ROI extracted, I passed it to Tesseract for text recognition. I used the `--oem 1` option to select the LSTM-based OCR engine, which is known for its high accuracy. I also used the `--psm 6` option, which tells Tesseract to treat the ROI as a block of text.

Here's the code for performing OCR with Tesseract:

```
text = pytesseract.image_to_string(roi, lang='chi_sim+eng', config='--oem 1 --psm 6')
```

In this code, the `image_to_string` function takes the ROI and returns the recognized text. The `lang` parameter specifies the languages to use (in this case, Chinese and English), and the `config` parameter allows me to fine-tune the OCR settings.

3. Saving the Recognized Text

Once Tesseract finished recognizing the text, I saved the results to a text file. I used the `os` library to create a directory for the output files and ensure that each image had its own corresponding text file.

Here's the code for saving the recognized text:

```
save_recognized_text(detected_text, output_filename)
```

This function saves the recognized text to a file in the `output_text` directory, ensuring that the results are organized and easy to access.

Visualization and Results

To visualize the results, I displayed the processed images, including the detected text regions, the binary image, and the denoised image. OpenCV's `imshow` function allowed me to show these images in separate windows.

Here's the code for visualizing the results:

```
cv2.imshow("Text Detection with EAST", output_image)
cv2.imshow("Binary Image", binary_image)
cv2.imshow("Denoised Image", denoised_image)
```

The processed images provided a clear visual representation of how the text was detected and extracted. The bounding boxes around the text regions made it easy to see where the text was located in the image, and the binary image showed how the text stood out after preprocessing.

Once I reviewed the results, I could press any key to close the image windows:

```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Conclusion

This project shows how modern computer vision techniques, combined with traditional OCR, can be used to effectively detect and recognize text in images. By using a combination of image preprocessing techniques, the EAST model for text detection, and Tesseract for OCR, I was able to create a system that accurately detects and extracts text from images.

The preprocessing steps—grayscale conversion, sharpening, contrast stretching, denoising, and thresholding—played a crucial role in improving the quality of the images for OCR. The EAST model was highly effective for detecting text regions, even in complex scenes with noisy backgrounds or rotated text. Tesseract OCR provided high accuracy in recognizing text in both English and Chinese, making it a versatile tool for multilingual OCR tasks.

In future studies, I plan to explore ways to optimize the speed and efficiency of the system, especially when dealing with large image datasets. This could involve experimenting with different text detection models or implementing parallel processing to speed up the OCR pipeline. Additionally, exploring other OCR engines or machine learning models could help improve accuracy in challenging scenarios, such as low-resolution images or heavily distorted text.

Overall, this project highlights the potential of combining traditional and modern techniques to solve real-world problems like text recognition in images, and I'm excited to continue exploring this field.

References

1. **EAST: An Efficient and Accurate Scene Text Detector**

   https://arxiv.org/abs/1704.03155

2. **Tesseract OCR Engine Documentation**

   https://github.com/tesseract-ocr/tesseract

3. **OpenCV Library for Python**

   https://docs.opencv.org/4.x/

4. **NumPy Library for Scientific Computing**

   https://numpy.org/doc/stable/