21.1.3 Automatically Downloading Example Images

The first step in building our captcha breaker is to download the example captcha images themselves. If we were to right click on the captcha image next to the text *"Security Image"* in Figure 21.1 above, we would obtain the following URL:

`https://www.e-zpassny.com/vector/jcaptcha.do`

If you copy and paste this URL into your web browser and hit refresh multiple times, you'll notice that this is a dynamic program that generates a new captcha each time you refresh. Therefore, to obtain our example captcha images we need to request this image a few hundred times and save the resulting image.

To automatically fetch new captcha images and save them to disk we can use `download_images.py`:

```
1  # import the necessary packages
2  import argparse
3  import requests
4  import time
5  import os
6
7  # construct the argument parse and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-o", "--output", required=True,
10      help="path to output directory of images")
11 ap.add_argument("-n", "--num-images", type=int,
12      default=500, help="# of images to download")
13 args = vars(ap.parse_args())
```

Lines 2-5 import our required Python packages. The `requests` library makes working with HTTP connections easy and is heavily used in the Python ecosystem. If you do not already have `requests` installed on your system, you can install it via:

```
$ pip install requests
```

We then parse our command line arguments on **Lines 8-13**. We'll require a single command line argument, `--output`, which is the path to the output directory that will store our raw captcha images (we'll later hand label each of the digits in the images).

A second optional switch `--num-images`, controls the number of captcha images we're going to download. We'll default this value to 500 total images. Since there are four digits in each captcha, this value of 500 will give us $500 \times 4 = 2,000$ total digits that we can use for training our network.

Our next code block initializes the URL of the captcha image we are going to download along with the total number of images generated thus far:

```
15 # initialize the URL that contains the captcha images that we will
16 # be downloading along with the total number of images downloaded
17 # thus far
18 url = "https://www.e-zpassny.com/vector/jcaptcha.do"
19 total = 0
```

We are now ready to download the captcha images:

```
21 # loop over the number of images to download
22 for i in range(0, args["num_images"]):
```

```
23      try:
24          # try to grab a new captcha image
25          r = requests.get(url, timeout=60)
26
27          # save the image to disk
28          p = os.path.sep.join([args["output"], "{}.jpg".format(
29              str(total).zfill(5))])
30          f = open(p, "wb")
31          f.write(r.content)
32          f.close()
33
34          # update the counter
35          print("[INFO] downloaded: {}".format(p))
36          total += 1
37
38      # handle if any exceptions are thrown during the download process
39      except:
40          print("[INFO] error downloading image...")
41
42      # insert a small sleep to be courteous to the server
43      time.sleep(0.1)
```

On **Line 22** we start looping over the `--num-images` that we wish to download. A request is made on **Line 25** to download the image. We then save the image to disk on **Lines 28-32**. If there was an error downloading the image, our `try/except` block on **Line 39 and 40** catches it and allows our script to continue. Finally, we insert a small sleep on **Line 43** to be courteous to the web server we are requesting.

You can execute `download_images.py` using the following command:

```
$ python download_images.py --output downloads
```

This script will take awhile to run since we have (1) are making a network request to download the image and (2) inserted a 0.1 second pause after each download.

Once the program finishes executing you'll see that your `download` directory is filled with images:

```
$ ls -l downloads/*.jpg | wc -l
500
```

However, these are just the *raw captcha images* – we need to *extract* and *label* each of the digits in the captchas to create our training set. To accomplish this, we'll use a bit of OpenCV and image processing techniques to make our life easier.

21.1.4 Annotating and Creating Our Dataset

So, how do you go about labeling and annotating each of our captcha images? Do we open up Photoshop or GIMP and use the "select/marquee" tool to copy out a given digit, save it to disk, and then repeat *ad nauseam*? If we did, it might take us *days* of non-stop working to label each of the digits in the raw captcha images.

Instead, a better approach would be to use basic image processing techniques inside the OpenCV library to help us out. To see how we can label our dataset more efficiently, open a new file, name it `annotate.py`, and inserting the following code:

```
1  # import the necessary packages
2  from imutils import paths
3  import argparse
4  import imutils
5  import cv2
6  import os
7
8  # construct the argument parse and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--input", required=True,
11      help="path to input directory of images")
12 ap.add_argument("-a", "--annot", required=True,
13      help="path to output directory of annotations")
14 args = vars(ap.parse_args())
```

Lines 2-6 import our required Python packages while **Lines 9-14** parse our command line arguments. This script requires two arguments:

- `--input`: The input path to our raw captcha images (i.e., the `downloads` directory).
- `--annot`: The output path to where we'll be storing the labeled digits (i.e., the `dataset` directory).

Our next code block grabs the paths to all images in the `--input` directory and initializes a dictionary named `counts` that will store the total number of times a given digit (the key) has been labeled (the value):

```
16 # grab the image paths then initialize the dictionary of character
17 # counts
18 imagePaths = list(paths.list_images(args["input"]))
19 counts = {}
```

The actual annotation process starts below:

```
21 # loop over the image paths
22 for (i, imagePath) in enumerate(imagePaths):
23      # display an update to the user
24      print("[INFO] processing image {}/{}".format(i + 1,
25           len(imagePaths)))
26
27      try:
28           # load the image and convert it to grayscale, then pad the
29           # image to ensure digits caught on the border of the image
30           # are retained
31           image = cv2.imread(imagePath)
32           gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33           gray = cv2.copyMakeBorder(gray, 8, 8, 8, 8,
34                cv2.BORDER_REPLICATE)
```

On **Line 22** we start looping over each of the individual `imagePaths`. For each image, we load it from disk (**Line 31**), convert it to grayscale (**Line 32**), and pad the borders of the image with eight pixels in every direction (**Line 33 and 34**). Figure 21.2 below shows the difference between the original image (*left*) and the padded image (*right*).

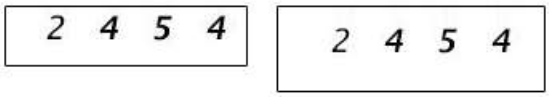

Figure 21.2: **Left:** The original image loaded from disk. **Right:** Padding the image to ensure we can extract the digits *just in case* any of the digits are touching the border of the image.

We perform this padding *just in case* any of our digits are touching the border of the image. If the digits *were* touching the border, we wouldn't be able to extract them from the image. Thus, to prevent this situation, we purposely pad the input image so it's *not possible* for a given digit to touch the border.

We are now ready to binarize the input image via Otsu's thresholding method (Chapter 9, *Practical Python and OpenCV*):

```
36             # threshold the image to reveal the digits
37             thresh = cv2.threshold(gray, 0, 255,
38                 cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

This function call automatically thresholds our image such that our image is now *binary* – black pixels represent the *background* while white pixels are our *foreground* as shown in Figure 21.3.

Figure 21.3: Thresholding the image ensures the foreground is *white* while the background is *black*. This is a typical assumption/requirement when working with many image processing functions with OpenCV.

Thresholding the image is a critical step in our image processing pipeline as we now need to find the *outlines* of each of the digits:

```
40             # find contours in the image, keeping only the four largest
41             # ones
42             cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
43                 cv2.CHAIN_APPROX_SIMPLE)
44             cnts = cnts[0] if imutils.is_cv2() else cnts[1]
45             cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:4]
```

Lines 42 and 43 find the contours (i.e., outlines) of each of the digits in the image. Just in case there is "noise" in the image we sort the contours by their area, keeping only the four largest one (i.e., our digits themselves).

Given our contours we can extract each of them by computing the bounding box:

```
47             # loop over the contours
48             for c in cnts:
```

```
49                # compute the bounding box for the contour then extract
50                # the digit
51                (x, y, w, h) = cv2.boundingRect(c)
52                roi = gray[y - 5:y + h + 5, x - 5:x + w + 5]
53
54                # display the character, making it larger enough for us
55                # to see, then wait for a keypress
56                cv2.imshow("ROI", imutils.resize(roi, width=28))
57                key = cv2.waitKey(0)
```

On **Line 48** we loop over each of the contours found in the thresholded image. We call `cv2.boundingRect` to compute the bounding box (x, y)-coordinates of the digit region. This region of interest (ROI) is then extracted from the grayscale image on **Line 52**. I have included a sample of example digits extracted from their raw captcha images as a montage in Figure 21.4.

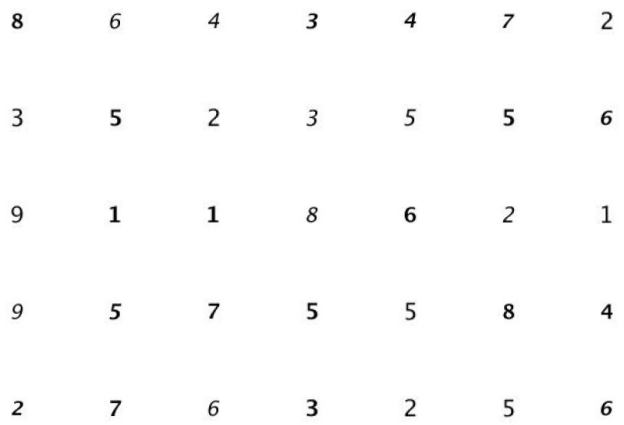

Figure 21.4: A sample of the digit ROIs extracted from our captcha images. Our goal will be to label these images in such a way that we can train a custom Convolutional Neural Network on them.

Line 56 displays the digit ROI to our screen, resizing it to be large enough for us to see easily. **Line 57** then waits for a keypress on your keyboard – but choose your keypress wisely! The key you press will be used as the *label* for the digit.

To see how the labeling process works via the `cv2.waitKey` call, take a look at the following code block:

```
59                # if the '`' key is pressed, then ignore the character
60                if key == ord("`"):
61                    print("[INFO] ignoring character")
62                    continue
63
64                # grab the key that was pressed and construct the path
65                # the output directory
66                key = chr(key).upper()
67                dirPath = os.path.sep.join([args["annot"], key])
68
```