Project 2- 24-25
Report

Sanjiv Sarvaesh Sivapalani
Daim Shraif
Group 29

Summary details of the file retrieval.

1. First, we connected to the servers using OpenVPN to access and retrieve the CSV file containing the image filenames on the raspberry pi. After transferring the file locally, we used a Python script ”fetch.py“ using get to automate the process of fetching the images from the server via the URL: https://cs7ns1.scss.tcd.ie/?shortname={username}&myfilename={filename}

2. A GET request to the server is made, post which the file is saved to the given download folder. Firstly we defined the server endpoints variable by specifying the base_url= https://cs7ns1.scss.tcd.ie/ and shortname=”{filename}”, so these variables define the server base URL and user-specific name, which will be used to construct download links for each filename in the csv list. Then we made an empty “filename []” list to store all the image filenames that needed to be downloading.

3. Next we read and populate the filenames from the csv file using a loop where it reads each line while removing any extra whitespace around the filename using strip() and appending it to the filename[].

4. Finally, a while loop to check the folder “captcha” to identify files which are already downloaded using os.listdir(). It converts both filenames and downloaded lists to sets and subtracts sets from the filename and again converts the results back to the list(to avoid having duplicates while ensuring files already present in the captcha folder are skipped).

   Time Taken to Fetch the image datas:

   Start time: Thu Oct 22 12:16:12 2024

   End time: Thu Oct 22 13:05:28 2024

   Total time taken: 2956.41 seconds.

Scalability Problems faced while image retrieval:

1. **Repeated Retrieval Attempts**: Even after some files are successfully downloaded, the loop continuously checks the entire filenames list, attempting to redownload images that may already exist locally. This results in redundant requests and wasted bandwidth.

2. **Frequent directory scans**: os.listdir("captchas") runs in each iteration, causing inefficiency as the image count increases

3. **Slow Download Feedback:** Downloading images one by one slows the process, particularly with network fluctuations, risking indefinite looping or timeouts during server or network bottlenecks.

Summary details of data generation/preparation:

1. Symbol Set Identification: The symbols and characters in this dataset were chosen after studying common patterns in CAPTCHA datasets, with additional help from communicating and interacting on Piazza. The final character set includes a mix of numbers, uppercase letters, and symbols, specifically "123456789aBCdeFghjkMnoPQRsTUVwxYZ+%|#][{}-. ".

2. Font identification/selection: After carefully reviewing the dataset, we selected and confirmed fonts such as JJester, Ring of Kerry (for Sanjiv), and Crazy Style, Faux Omniglot (for Daim). Please note that many people had a query about picking between Eamon and Faux Omniglot. we picked the latter; it was not a complete match but it was the closest to the font present in the captchas. This will affect the results later.

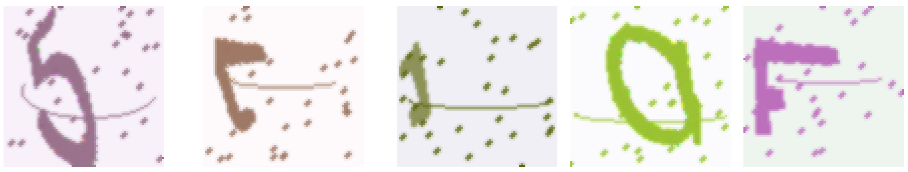
3. Data generation/preparation: We generated 2500 images using ImageCaptcha for each character in the symbol set using the identified font. For each image we generated, we set random font choices with a fixed dimension for width and height at 80x80. By doing so we get different set of images with font variations along, and each character in the font appear in multiple unique orientations, which will be very helpful for diversifying the training data.

   Time taken for Image Generation for training:

   Start time: Sun Oct 27 12:34:40 2024

   End time: Sun Oct 27 12:40:39 2024

   Total time taken: 358.77 seconds



Above images are a few samples after generation.

Problems Faced:

1. **Folders for Symbols**: Windows restricts certain symbols like `|`, `#`, and `[` from being used in folder names, so each symbol must be mapped to a compatible name (e.g., `|` becomes "pipe"). This setup requires careful handling to avoid errors and ensure that every character has a valid storage path.

2. Randomized font and size choices for each image slow down generation when scaling to thousands, due to frequent calls for random selection and font switching.

3. Tracking generation times and logging each image add complexity; as images increase, log files grow, making performance tracking harder without efficient logging.

4. Creating unique folders for each character requires repeated checks, which adds computational load, especially with many characters and high sample count.

Summary details of creation of training datasets and Validation:

1. Out of the 2500 generated images for each character, we have classified them into the training dataset and validation dataset (2000 for train and 500 for validation) in the ratio of 80:20

2. To improve our model's accuracy and prevent possible overfitting, we wrote code to separate 500 image files randomly from the generated dataset of 2500 files for each character using **`images_to_move = random.sample(images, num_images_to_move),** so now we will have 2000 images in the train folder and 500 images in the validation folder, ready for preprocessing.

Problems faced:

1. Creating separate validation folders for each class increases computational load because of repeated checks and folder creation commands

2. Randomly selecting images for validation from each class folder demands memory and processing power, leading to slower performance as the dataset size increases.

3. Choosing random images for validation in each class folder requires memory and processing, which slows down as the dataset gets larger.

Summary details of Preprocessing:

1. Firstly we initialized a timer to track the processing time.
2. For the testing dataset our main focus was on image preprocessing and character segmentation.
    a. Numerous approaches for Image preprocessing
        i. Grayscale: We converted the captchas to a grayscale, ie. converting from 3 channels to 1 channel. This will help later on since we removed color as a factor for predicting and mainly kept intensity values for the model or any other algorithm used to focus on.

        ii. Gaussian filtering: We then also tried applying a Gaussian filter on the grayscale images to smooth out the noise present in the captchas. This resulted in the noise still being present making it difficult to segment it later on.

        iii. Median filtering: Apart from Gaussian, we also tried a non-linear approach, median filtering, to remove the pepper-like noise present in the captchas, however, this resulted in distorting the edges of the captchas.

iv. Dilation: We also tried dilation on the grayscale images, we exhausted multiple kernels to work on the dataset, and multiple iterations. A problem faced here was that throughout the dilation, the character started to thicken at its border such that a '*w*' would start to look like a '-', or a pipe, angle bracket, the number '1', would start looking the same.

v. Erosion: apart from dilation, we used erosion as well to slim out the thickness present in the characters, this would work out well on some of the fonts that are thick, like a crazy font, but it started eating up characters leaving nothing for characters present in faux omniglot.

vi. Binary Inverted Threshold: this technique was pivotal for this project because it allowed us to remove noise and bring out edges. This function is allowed to set all the pixels greater than the threshold 0, and those below are set to the maximum value, 255.

vii. Binary Threshold: this technique was also pivotal for this project because it allowed us to remove noise and bring out edges. This function allowed us to set all the values below a certain threshold to white, 255, in a grayscale image.

viii. Morphology: we also carried out morphological operations on the binary images, this function allowed us to carry out multiple operations together in a sequence, like erosion followed by dilation, dilation followed by erosion, and the difference between dilation and erosion. This allowed us to enhance and clean the images. However, we still had to figure out the best set of operations to preprocess.

ix. Remove circles: this was a custom function that we tried where we would identify circles, and noise, using HoughCircles once identified the circles and the right size of the circle through numerous trials and errors, we then took the average of the intensity *outside* the circle to replace the noise with the color. We first replaced it with 255, white, but there were some spots on the character itself, so replacing it with white would mean removing valuable data, that is why we decided to average the surrounding area to replace the intensity of the noise.

x. All of these approaches were carried out in all possible ways, in all possible order. We tried running these algorithms on the inverted binary and the regular binary, in the end we decided to stick with grayscale, dilation and thresholding.

b. Numerous approaches for image segmentation (all of the approaches that will be mentioned were tried out on inverted binary and regular binary outputs)

   i. Contour: we tried contour to segment the images into single characters. Contouring allowed us to identify the boundaries of the images, however after multiple attempts, it was tough segmentation using contour as most of the time it could not separate characters that were next to each other.

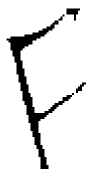
ii. MSER (Maximally Stable Extremal Regions): we tried MSER which is a region based segmentation algorithm to segment the captchas. It detects regions that would be consistent/stable across a specified threshold. There were many arguments to test out with MSER like the delta that refers to the sensitivity of MSER to changes in intensity. We had to also test the min_area and max_area, because we did not want 100 regions detected for a picture with 4 characters on it. In addition, tweaking the min_area and max_area allowed MSER to relax so that it could identify characters large in size

iii. We decided to carry on with MSER, however we faced a lot of challenges during segmentation. There were issues where the algorithm would detect many regions for a captcha with a few characters. This would produce wrong results since a classification model would require x images to classify x classes. To combat that we used an overlap function that would drop out a region if there was an overlap in the area of the 2 regions above a threshold. Another issue was that in the output file, the regions were saved in a different order to the order of the captchas, from left to right. We fixed it by sorting the regions detected against the x-axis.

iv. After getting the individual segments we had to resize it 100x100 for the model, we did not want to use the resize algorithm since we saw it was stretching out the images. so we first tried pasting the image on to a background, but that did not work since there were segments of different sizes that would exceed the 100x100 boundary. Instead, we used a padding function to centre the segment in the centre from all sides.

3. For preprocessing the training/validation data, no segmentation was required as we generated single characters, so instead we followed the same image preprocessing steps before image segmentation in the testing dataset.
4. Records of the total time taken for execution will be saved in a text file assigned to it.

   Time Taken for Preprocessing:
   For Train - Execution time: 148.6505 seconds
   For Validation - Execution time: 45.7553 seconds



Above images are a few samples after Preprocessing

Model training and inference :

1. After preprocessing the train and val, we put them in a separate folder, train_model_data for training it using YOLO with cuda.

2. Then we specified the model as YOLO to run the model.train using model=YOLO(directory).to('cuda') command using ultralytics.
3. For this project, we decided to use yolo11s-cls.pt, which is a pretrained model and so we needed to run a few epochs to finetune the model, specifically 7.
4. We trained 2 models, one for a pair of fonts.
5. YOLO based architecture:
    a. The model employs a backbone network, such as CSPDarknet, to capture and reduce image features across multiple levels, helping it identify diverse patterns, shapes, and textures in different CAPTCHA characters and symbols.
    b. Convolutional layers are structured to add depth efficiently, focusing on essential CAPTCHA features like character contours, font styles, and distortions, crucial for accurate symbol classification.
    c. The model's final layers act as a classification head, outputting probabilities for each CAPTCHA character, enabling quick labeling even with varying fonts and rotations.

6. Overall at the end of the training period, which took 48 min per epoch, we achieved a top-1 validation accuracy score of around 95.6%. There were no signs of overtraining.
7. We then exported the model to a onnx format to deploy it on the raspberry pi for inference
8. Using the pi throughout the project was very difficult in terms of reliability. 70% of the times it would timeout and disconnect and the other 30% it would just be stuck. And so running on the raspberry pi was an achievement in the end.