Character Recognition

2021

# Set-Up

**Solution Overview**

|  |  |
| --- | --- |
| Project |  |
| CharRecognitionConsole | Console program (C#) |
| CharRecognitionFunctions | Azure functions (C#) |
| CharRecognitionLib | Library of routines (C#) |
| WebUI | ASP.Net Core project (ASP.Net version 5, Azure App Service; C#, JavaScript, HTML, CSS) |

All projects have been built on .Net 5.

**The Project can be brought up piece by piece.** Make sure each step works before proceeding to the next step.

**#1. Entering New Characters**

Publish the "WebUI" ASP.Net Core project.

The user visits "WebUI/Pages/AddData" to enter characters.

The data is stored in Azure blob storage. The storage account connection string needs to be at: "CharRecognitionLib.Credentials.Azure\_Blob\_Storage". Furthermore, the storage account needs to have a "image-data" container.

The user can then visit "WebUI/Pages/ViewData" to view characters that have been entered.

**#2. Uploading MNIST Data**

Download and unzip the MNIST data. There should be four files, with the largest one being "train-images.idx3-ubyte".

Run the console program. Menu option #1 "Upload MNIST Data", will upload data to the "image-data" storage container mentioned earlier.

After the MNIST data has been uploaded, visiting "WebUI/Pages/ViewData" will show the characters.

**#3. Standardize one set of images**

The "image-data" pictures are standardized into 8x8 black and white pictures.

The console program menu option #2 "Standardize one image file" can be used to standardize one image file. Enter in "0" for label, and "0" for file number, to standardize "0\_0.bin".

The original pictures would be in "image-data/original/0\_0.bin"

The standardized pictures would in "image-data/standard/0\_0.bin".

After a set of images have been standardized, you can view them via "WebUI/Pages/ViewData".

**#4. Standardize all image files**

The console program's menu option #2 "Standardize one image file" is for debugging. This standardize one file only.

Azure functions is used to standardize all data files.

Setup the Azure functions by publishing the "CharRecognitionFunctions" project.

The URL to access the Azure image standardization function needs to be at: "CharRecognitionLib.Credentials.Azure\_Standardize\_Image"

The console program menu option #3 "Standardize ALL image files" will call the Azure functions, using multiple threads, to standardize all image files.

**#5. Test model using existing images**

The "ML" web page tests the images and allows selecting data for the next round of training.

This page eventually relies on Azure to do the testing. The URL at "Credentials.Azure\_Test\_Model" needs to be updated for the project.

The blob storage needs to have a "char-recognition" container to hold the ML results.

To recap, the blob storage referenced by "Azure\_Blob\_Storage" needs to have two containers:

* "image-data" - holds original and standardized data
* "char-recognition" - holds training data and the final templates for character recognition

**#6. Retraining the model**

Model retraining is distributed over Azure functions, one function trains one label.

By default there are 10 digits, 0 ~ 9, which means there will be ten Azure function calls.

The "Credentials.Azure\_Update\_Template\_Collection" holds the URL of the Azure training function. Hitting the "Retrain" button will eventually trigger this Azure URL.

**#7. Debugging the mistakes**

The "CharRecognitionConsoleApp", option "6. Test model", can be used to debug the model.

The incorrectly classified images, along with the top match template, will be displayed.

Example output:

Image #28 is incorrectly identified.

Image Best Match Template

\_\_\_\*\*\*\_\_ \_\_1\*\*1\_\_

\_\_\*\*\_\_\_\_ \_1\*\*1\_\_\_

\_\_\*\_\_\_\_\_ \_1\*1\_\_\_\_

\_\_\*\*\*\_\_\_ \_1\*111\_\_

\_\_\_\_\*\*\_\_ \_1\*\*\*\*1\_

\_\_\_\_\_\*\*\_ \_1\*11\*1\_

\_\_\*\*\*\*\*\_ \_1\*\*1\*1\_

\_\_\_\_\_\*\*\_ \_\_\_\*\*\*1\_

In this case, the image is 5, but the best match template is a 6.

The '\*'s are foreground pixels. The '1's are the adjacent pixels that has been enabled, and the '\_' are the background pixels.

# Image Data

**Data Storage Format**

**image-data container**

The image data is stored in the "image-data" Azure Blob Storage container.

* Data under "original/" includes images drawn by the user and the MNIST dataset.

**binary format**

The images drawn by the user are binary, 16x16.

Each row is encoded as 2 byes, and each image is encoded as 32 bytes.

The storage order is row major. This means the image\_bytes[0] and image\_bytes[1] represent the first row of the image, then image\_bytes[2] and image\_bytes[3] represent the second row of the image, and so on.

**grayscale format**

The images from the MNIST dataset are grayscale, 28x28.

Each pixel takes up one byte, and each image takes up 784 byes.

The storage order is row major. This means the first row takes up image\_bytes[0..27].

**File Name format**

The file name format is "label\_dataNumber.bin".

Each "bin" file contains 32 images. This is hard coded into "CharRecognitionLib.ImageData.max\_images\_per\_file".

The original, unmodified data, is placed under the "/original" directory.

Examples:

|  |  |
| --- | --- |
| File Name | Meaning |
| original/0\_0.bin | Label "0", original data, images 0 ~ 31 |
| original/0\_32.bin | Label "0", original data, images 31 ~ 63 |
| original/0\_1056.bin | Label "0", original data, images 1056 ~ 1056 + 31 |

**index.tsv**

The "index.tsv" file contains information about each label.

Example:

|  |
| --- |
| Label Type Height Width Last File  A B 16 16 0  T B 16 16 0  0 G 28 28 6880  1 G 28 28 7872 |

Type "B" = black and white. Type "G" = grayscale.

The label "A" is of black and white, and the last file is "A\_0.bin". Having just one ".bin" file means label "A" has at most 32 files.

The label "0" is of grayscale, size 28 x 28, and its last file is "0\_6880.bin". So label "0" has between 6880 to (6880 + 31) images.

**Code**

**ImageData (CharRecognitionLib.ImageData)**

Manages images in the "image-data" container.

Can upload images into Azure Blob Storage, and download images from Azure Blob Storage.

**IndexEntry (CharRecognitionLib.ImageData.IndexEntry)**

In memory representation of a single row in the "index.tsv" file.

The entire index is represented as:

Dictionary<string, IndexEntry> index;

**ImageData\_Controller (WebUI.Controllers.ImageData\_Controller)**

APS.Net Core API endpoints for the "image-data" container.

Handles rest API calls for accessing the "image-data" container.

**AddData Razor Page (WebUI.Pages.AddData.cshtml)**

Web UI for user to add images to the "image-data" container.

**ViewData Razor Page (WebUI.Pages.ViewData.cshtml)**

Web UI for user to view images currently in the "image-data" container.

**upload\_mnist\_data() (CharRecognitionConsoleApp.Program.upload\_mnist\_data())**

Console program to upload MNIST data, from the hard drive to the "image-data" container.

# Image Processing

**Image Standardization**

The images added by the user, and the images from the MNIST dataset, are standardized into 8x8 black and white images, stored at the "image-data" blob container, with the "standard/" prefix.

**Code**

The image processing code is in "CharRecognitionLib/BlackAndWhite\_Image.cs"

The code does binarization, thinning, centering, up scaling, and down scaling.

**Binarization**

The code for binarization is "grayscale\_to\_black\_white()".

The brightest pixel in the image is found. For example let this be 210.

The dark lower third pixels are set to background value of 0.

The bright upper third pixels are set to foreground value of 255.

The middle third pixels are set to 255 if they are neighbor of a 255 pixel.

**Thinning**

The thinning algorithm is K3M.

It's based on the paper at "https://www.researchgate.net/publication/220273912". A copy of this paper is at "docs/K3M\_thinning\_algorithm.pdf"

The project does not use the final "1-pixel width" phase mentioned in the paper. The code exists as the "one\_pixel\_phase()". The current rule set, implemented as the look up table "A\_1pix", removes too many pixels. A final "1-pixel width" will probably improve this project's performance, but the rule set has to be tweaked.

**Azure function for image standardization**

The code is at "CharRecognitionFunctions/StandardizeImages.cs"

This function accepts a POST request with the format:

{  
 "label": "0",  
 "file\_numbers": "0,32,64",  
}

The above input will cause the function to standardize the following files:

/original/0\_0.bin  
/original/0\_32.bin  
/original/0\_64.bin

# ML UI

This section describes the events that take place on "WebUI/Pages/ML.cshtml"

Diagram

Description automatically generated with low confidence

|  |  |
| --- | --- |
| **Client Side**  WebUI/Pages/ML.cshtml | **Server Side**  WebUI/Controllers  CharRecognitionLib/RecognitionModel  CharRecognitionFunctions |
| User hits the "Test" button.  Triggers "start\_test()"  Issues a test request, which includes:  label - string, "All Labels" or individual label  data - string, "all" or "some"  start - string, applies if "data" is "some"  end - string, applies if "data" is "some" |  |
|  | ASP.Net's "ML\_Controller.TestData" receives the request  RecognitionModel's "Test\_Model\_Task" is started to execute the request. This task starts multiple threads. The testing task is distributed over multiple Azure function calls.  CharRecognitionFunctions's "TestModel" runs a portion of the test request.  Eventually, ASP.Net's "ML\_Controller.TestData" returns a response that includes:  success - "true" if task handled  correct - int, correctly classified  incorrect - int, incorrectly classified  unknown - int, unable to classify  misclassified - List<int>, image numbers that failed to be correctly classified |
| The client side creates a table that summarize the test results using "SummaryUI.init()". The table has one row for each table, and four columns: (label, correct, incorrect, unknown).  The testing can take a while if the Azure functions are slow to scale. The client side polls the server side for progress via "SummaryUI.update()". This displays the latest results in the table generated by "SummaryUI.init()".  Eventually the test is done, and the client side generates the UI that displays the misclassified images using "M\_ImagesUI.init()".  The "M" in "M\_ImagesUI" stands for "misclassified". |  |
| When multiple labels are being tested, the misclassified images UI has multiple labels to choose from. When the user chooses a new label, a new set of misclassified images is shown. This is handled by "M\_ImagesUI.label\_selection\_changed()"  When a new label is selected, the client requests the images from the server by issuing an "image-data/get\_images\_by\_number" request that includes:  prefix - always "standard/", because only the standardized images are being used for training  label: string  sortedImageNumbers: List<int>, these represent the misclassified images. They are assumed to be sorted to make it easier for the server to extract them from the "label\_number.bin" files. |  |
|  | The request for images is handled by "ImageData\_Controller.GetImagesByNumber()"  This request is further delegated to "ImageData.Get\_Label\_Bytes\_By\_Numbers()"  The "ImageData\_Controller.GetImagesByNumber()" returns  success - bool  base64\_Data - string, the image bytes encoded as base64 string  type - string, always "B", because the standardized images are always black and white  height, width - int |
| After receiving the misclassified images (base64 encoded binary data) from the server, the client side draws the various canvases. This is in "label\_selection\_changed()".  Drawing the images on the canvas is in "draw\_image\_black\_and\_white()". |  |
| The user can select images for training by clicking on them. This triggers the "toggle\_canvas()".  Selecting images for training does not update the user. The current implementation modifies the data structure stored at "M\_ImagesUI.data". |  |
| When the user clicks the "Retrain" button, at '$("retrain\_btn").onclick', the numbers representing the selected images will be send to the server. |  |

# Templates

This application uses template matching to recognize characters.

**1 Pixel Matching**

An image matches the template if the pixels are at most 1 pixel off.

For example, the template for a "1":

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

will match with:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

The blue pixels are just one pixel away from the template pixels.

One pixel matching is how the model generalizes.

**Score Based Matching**

Pixels that match the template exactly is 0, as in the distance from the template is zero.

Pixels that are one pixel off will count as a score of 1, as in the distance from the template is one.

Continuing the previous example:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | 1 |
|  |  | 0 |  |
|  |  | 0 |  |
|  | 1 |  |  |

The total difference score is 2.

Any pixel more than 1 pixel away from the template will count as a score of 255, and the matching algorithm instantly terminates.

**Two Way Matching**

The template matching score is computed two ways.

As an example consider two images:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Image 1   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  | |  |  |  |  | |  |  |  |  | |  |  |  |  | | Image 2   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  | |  |  |  |  | |  |  |  |  | |  |  |  |  | |

Projecting image 1 onto image 2 gives a perfect score of 0, because every pixel of image 1 is exactly on image 2.

Yet projecting image 2 onto image 1 gives a score of 2:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 1 |  |
|  |  | 0 |  |
|  |  | 1 |  |
|  |  |  |  |

because two pixels of image 2 are not exactly on image 1. It's still considered a match, but the final difference score is 2. This score of 2 will rank below other better matches.

**Training**

The core pixels that are part of the template is assigned a score of 0. Training is about enabling the pixels that are adjacent to the core template pixels. Blindly enabling all pixels next to template pixels will not work.

For example, suppose a template for 3:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

If 1 pixel neighbors are all enabled, then every single pixel in the template is enabled. Anything will match this template 3.

In particular, the number 9 will match this template 3 extremely well:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Only two pixels are not on the "3" template, for a total difference score of 2.

Relying on scoring alone is not sufficient to solve this incorrect match.

During training, the software first enables all pixels adjacent to the template:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Then the training algorithm sees the 9, and will disable this match scenario by turning the two mismatching pixels in the 9 to 255.

The post training template for the "3" becomes:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

The background pixels in the "Template" class are marked as 255. When attempting to match the "3" and the "9", the software will see the 255 pixels and will flag this pair as a mismatch. It will not attempt to compute the total difference score.

**Image Processing**

The templates are sized 8x8.

The input data on the "Demo" page is 16x16, so there's image thinning, followed by down sampling.

**Foreground Pixel List**

To make template matching faster, the Template class tracks which pixels are the foreground pixels:

public class Template

{

// Format is value 0 for foreground, adjacent to foreground

// pixels are 1, and all other pixels are 255

byte[,] image;

// Coordinates of the foreground pixels

byte[] foreground\_rows;

byte[] foreground\_cols;

Many operations will just go through the list of pixels, rather than go through all 8x8 pixels.

**Template Index**

The template index (class TemplateIndex) is a lookup table to speed up template matching.

For example:

|  |  |
| --- | --- |
| Coordinate | Template Candidates |
| (2, 3) | #1, 100, 150 |
| (4, 2) | #10, 100, 110 |

This means the templates #1, #100, #150 all either have a foreground pixel at exactly (2, 3), or have a foreground pixel adjacent to (2, 3).

Suppose an image has a foreground pixel at both (2, 3) and (4, 2), the only template that satisfy both pixel requirements, is template #100.

Furthermore, the list of coordinates are sorted, with two considerations in mind.

First, coordinates with the fewest templates are listed first. The goal is to narrow down the list of template candidates as quickly as possible.

Second, it's best to utilize coordinates from different quadrants. By jumping to different quadrants, the hope is that the "Template Candidates" in the table will change rapidly, which will reduce the number of template candidates as quickly as possible.

**Binary Storage of Templates**

The core content of a template object:

public class Template

{

// Format is value 0 for foreground, adjacent to foreground

// pixels are 1, and all other pixels are 255

byte[,] image;

// Coordinates of the foreground pixels

byte[] foreground\_rows;

byte[] foreground\_cols;

Binary layout:

|  |  |  |
| --- | --- | --- |
| Field Name | Length in Bytes |  |
| foreground\_rows  Length int | 4 | Number of foreground points. |
| **foreground\_rows**  **byte[]** | num\_foreground\_points | The "row" part of the (row, col) coordinate of the foreground points. |
| foreground\_cols  Length int | 4 | Number of foreground points. |
| **foreground\_cols**  **byte[]** | num\_foreground\_points | The "col" part of the (row, col) coordinate of the foreground points. |
| height int | 4 | Image height |
| width int | 4 | Image width |
| **image byte[,]** | height \* width |  |

The colored blocks are encoded / decoded in a single function call, supported by the "BytesWriter" and "BytesReader" classes.

For example, the "BytesWriter.Add\_2D\_Bytes(image)" will encode the height, then the width, and finally the content of "image".

**TemplateCollection**

Holds an array of template objects.

Tracks the labels for the templates.

Can merge with other "TemplateCollection" objects.

**TemplateCollection Label Management**

The "TemplateCollection" class stores a table for label management. For example:

|  |  |
| --- | --- |
| Label | Index |
| "0" | 0 |
| "1" | 20 |

In this case, the templates from index 0 to 19 belong to the label "0".

The "BinarySearch()" is used to map from "index" to label. So it's important that the table is sorted by "Index".

**TemplateCollection Binary Representation**

|  |  |  |
| --- | --- | --- |
| Field Name | Length in Bytes |  |
| Height | 4 | Height of each template in the collection |
| Width | 4 | Width of each template in the collection |
| indices Count | 4 | The length of the "indices" list |
| indices List<int> | indices.Count \* 4 | The content of the "indices" list |
| labels Count | 4 | The length of the "labels" list |
| labels[0] Length | 2 | The length (in bytes) of the "labels[0]" string |
| labels[0] string |  | The content of the "labels[0]" string |
| labels[1] Length | 2 | ... |
| labels[1] string |  | ... |
| ... |  |  |
| templates Count | 4 | The number of elements in the "templates" list |
| templates[0] Size | 4 | The number of bytes taken up by templates[0] |
| templates[0] |  | The binary content of templates[0] |
| templates[1] Size | 4 |  |
| templates[1] |  |  |
| ... |  |  |

The colored blocks are encoded in a single function call.

For example, the "BytesWriter.Add\_Int\_List(indices)" will encode the length of the list, and then individual elements of the list. Also, the "BytesWriter.Add\_String(string)" will encode both the length of the string, and the string itself.

**Binary Storage of TemplateIndex**

The "template index" is composed of multiple "template index entry" objects:

Each template index entry's binary representation is an integer array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| row | col | first template ID | ... | last template ID |

The "template index" in binary is an integer array. Each template index entry is preceded by its size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| height | width |  |  |  |
| size of  first template index entry | first template index entry | size of  second template index entry | second template index entry | ... |

Functions:

TemplateIndexEntry.To\_int\_array()

TemplateIndexEntry.From\_int\_array()

TemplateIndex.ToBytes()

TemplateIndex(byte[] bytes)

# Model Building

This section picks up where the "ML UI" section left off. The model building starts when the user click the "Retrain" button.

**Clicking the Retrain Button**

Clicking the "Retrain" button submits new image numbers, which will be included in the training data.

The retrain button handler is at: WebUI.Pages.ML.cshtml

retrain\_ui.start\_retrain()

**ASP.Net Handlers**

The retrain request is handled on the server side by

WebUI.Controllers.MLControllers:

StartRetrain()

GetTrainingProgress()

monitor\_training\_end()

The "StartRetrain()" starts the training in a background thread. The code doing the actual work is:

CharRecognitionLib.Training\_Task

The webpage will periodically ask for progress, which is handled by the "GetTrainingProgress()".

The "monitor\_training\_end()" monitor will poll the "Training\_Task" for competition. When training is completed, the recognition model needs to be updated by being reloaded from storage.

**CharRecognitionLib.Training\_Task**

This code manages the thread that is doing the training.

* bool Done - to indicate that training is done
* string[] ReadMessages() - to retrieve messages generated during training

This code also oversees the training process:

train\_model()  
 |🡪 add\_new\_data()  
 |🡪 train\_model\_using\_serverless()  
 |🡪 merge\_template\_collections()

**CharRecognitionLib.Training\_Task.add\_new\_data()**

The ML UI allows the user to select images for training.

Suppose the user chooses label "5", images #3, 15, 25, for training. The numbers {3, 15, 25} will be written to the "char-recognition" container, at:

candidate\_numbers/5.bin

The code managing this is "TemplateCandidates".

public class TemplateCandidates

{

public readonly string Label;

HashSet<int> image\_id\_set = new HashSet<int>();

A "template candidates" object holds the label and a set of image IDs that are being used for training. In the preceding example, the label is "5", and "image\_id\_set" will be {3, 15, 25}.

Training any single label actually involves all the data. For example, to train "5", the templates for "5" needs to be able to reject the training data for all other characters. "5" is visually close to "6", so training "5" requires rejecting "6".

The code will read the various "candidate\_numbers/xxx.bin" files to see which images are being used for training. Then all these training images are packaged together into a single file, located in the "char-recognition" container, at:

template\_collections/all\_candidates.bin

**CharRecognitionLib.Training\_Task.train\_model\_using\_serverless()**

Training is distributed over Azure functions, one function call per label.

The "train\_model\_using\_serverless()" makes the Azure function calls.

**CharRecognitionFunctions.UpdateTemplateCollection.Run()**

This is the Azure function code that handles the training request.

The "Training" class is the code responsible for training.

CharRecognitionLib.Training.Train\_One\_Label()

In this application, training means:

* From a set of candidates, select a subset to be used as templates. For example, if the candidates are images #100, 150, 170, and it turns out that #150 and #170 are close, then only images #100 and #150 will be selected as templates for character recognition.
* For all the pixels that are adjacent to the template's pixels, enable only a subset. As an example, previously in the "Templates" section, there's the example of a hypothetical "3" template:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Originally the orange pixels enabled, because they differ from the template foreground by just one pixel. However, after discovering that "3" contradicts with another "9" image, those orange pixels are disabled.

**CharRecognitionLib.Training\_Task.merge\_template\_collections()**

Each Azure function trains just one label. The result of training label "5" would be stored in the "char-recognition" container, at:

template\_collections/5.bin

Another Azure function, working on label "6", will produce its output at

template\_collections/6.bin

The "merge\_template\_collections()" function merges all these outputs together, into a single template collection object, at:

template\_collections/all\_labels.bin

In addition, an index will be build and stored in the "char-recognition" container at:

template\_indices/all\_labels.bin

**WebUI.Controllers.ML\_Controller.monitor\_training\_end()**

This class is polling the training task ever second. When training is done, the ASP.Net refreshes the recognition model via:

WebUI.App.Refresh\_Recognition\_Model()

public static void Refresh\_Recognition\_Model()

{

var ti = Util.Download\_From\_Storage("char-recognition",

"template\_indices/all\_labels.bin");

var tc = Util.Download\_From\_Storage("char-recognition",

"template\_collections/all\_labels.bin");

recognition\_model = new RecognitionModel(ti, tc);

}

A recognition model is composed of two things:

* A collection of templates, stored at "template\_collections/all\_labels.bin"
* An index to search those templates, stored at "template\_collections/all\_labels.bin"