**Weekly report**

Realised that when saving our images we were consuming massive amounts of memory, to the point where we were getting memory errors in Python.

We resolved this by debugging our code bit by bit (pun intended), eventually getting our data down to the appropriate size. Without compression, with the addition of white space to matrices to make the images all the same (max) size, we generate about 100mb of data for 30mb of images.

A large part of this was rewriting much of our code so that we handle the information in array-form rather than converting it to an image, handling the information there and then converting it back.

Handling information in array form also gives us explicit manipulation over exactly what happens to our information, rather than using something such as PIL.

Next we needed to implement a process to scale our images. We first upscaled the image size to an integer multiple of the desired downscaled dimension. Then we added white space so that the upscaled image would be a square. Finally we downscaled the image to a desired size. This is encapsulated in a function which can be completed for every image, and is described in the figure below.

C:\Users\Sebastian\AppData\Local\Microsoft\Windows\INetCache\Content.Word\origImage.jpgC:\Users\Sebastian\AppData\Local\Microsoft\Windows\INetCache\Content.Word\upscaledImage.jpgC:\Users\Sebastian\AppData\Local\Microsoft\Windows\INetCache\Content.Word\upscaledImage.jpg

Figure 1: a) [76x56] Original image from .gnt file. b) [80x58] upscaled image so that one dimension (height) is an integer multiple of the desired reduced dimensions (40x40). c) [80x80] White space added along smaller dimension so image is square. d) [40x40] image reduced by averaging blocks to fit desired dimensions.

Now we have our data in usable form. Characters are labelled by numbers and all images are stored as 1D vector arrays. Storing each image as a 40x40 pixel image reduces the data file saved to 12mb.

Using the MNIST tutorial on the Tensorflow website, we have implemented a CNN for handwritten number recognition using the MNIST database. The network has two convolution layers each followed by max pooling. These are then followed by two fully connected layers. Training time is lengthy although this will be reduced using transfer learning.

Implemented a 1-hot vector approach for our Chinese characters so we can use the same CNN method as in the MNIST tutorial. Unfortunately the 1-hot vector file is huge (~1gb) since it is a (10x3755,3755) array, and mostly full of zeroes. We are trying to find a way around this by generating 1-hot vectors on the fly from a list.

The table below shows how Chinese characters correspond to a ord() value which we have to convert to a corresponding value in the range {1,3755} otherwise our hot ones vectors will be ~40,800 long (the max ord() value).

Figured out the issue with decoding, after reading through the python codec library we found out what codec we needed to use and are now able to decode all files, accessing a total of 3926 characters. This includes 171 Latin and alphanumerical characters, including some strange ones such as ‘km’ and ‘cc’.

|  |  |  |
| --- | --- | --- |
| Chinese characters | ‘ord()’ value | Corresponding list value for hot ones vector |
| 角 | 35282 | 989 |
| 饺 | 39290 | 3052 |
| 缴 | 32564 | 3663 |
| 绞 | 32478 | 3602 |
| 剿 | 21119 | 616 |
| 教 | 25945 | 1949 |
| 酵 | 37237 | 1651 |
| 轿 | 36735 | 2102 |
| 较 | 36739 | 1653 |
| 叫 | 21483 | 2268 |

**Action points for the next week**

1. Implement the CNN from the MNIST tutorial on our own data.

2. Take in all the data we have from the -c files so that we can start using this data for training

3. Read about the mathematics behind machine learning.

4. Research different thresholding techniques and their effect on optical character recognition. Do we need to binarize our data? (Apart from for reducing memory usage)

5. Research different machine learning methods used in optical character recognition