Assignment I – Chinese character “detection”

Part 1: data preparation

The data used for the assignment is of two types: firstly, a folder, which contains 845 images of size 2048x2048 that had been used for training in the original task; and secondly, two files that contain the information of the images, ‘info.json’ and ‘train.jsonl’. The first file was used to get the name of the images that had been used to train in the task, while the second file contains information that will be used in the assignment.

Most of the images used for the initial task and saved in the information file were not in the folder. Comparing the names of the images in the folder and the names in the information file, there were 845 images that could be used for the assignment. These were separated into a train dataset, with 80% of the images, a validation dataset, containing 10% of the images, and a test dataset, which contains the reminding 10%.

The ‘train.jsonl’ contained information about the images. The images used in the challenge contain Chinese characters, which are delimited by a polygon surrounding each one of them. The information that was used for the assignment was the name of the image and the coordinates of the polygons. There were also some polygons that contained other characters, not Chinese. These non-Chinese characters were never used, although they were saved because the original idea was to do so.

Having gotten the names of the images and the coordinates that form the polygons which contain Chinese characters, the objective was to load the images and get tensors representing each of their pixels. Simultaneously, we needed to get what we wanted our models to output: the representations of the probability of the image to contain a Chinese character (which would be 1 if there is one and 0 if there is not).

The tensors were created with the function “imageAndTensor”. However, given the big size of the images (2048x2048), they could never load. For this reason, the images were resized into 200x200 pixels before they were made into a tensor. After having worked in the creation of the models and seeing that the first layer of both models was a convolution layer, this function also permuted the last dimension of the image to the first position.

The expected output for the images was created with the function “ImageGoldValues”. In this case, we kept the original image size and produced a one-dimensional tensor containing 0 or 1, depending on the pixel being in a polygon (and, therefore, containing a Chinese character). After trying to run this unsuccessfully for days, getting CUDA out-of-memory errors, a classmate shared her approach using Parallel (from joblib) to help in multiprocessing and do parallel computing. With this implementation, it would still take circa 20 minutes to load the data. In addition, connection to mltgpu kept getting closed by a remote host, so getting the data to being able to test the models was a big challenge.

Part 2: the models

Since this was the first time that I was creating a whole model by myself from scratch, as well as the first time working with images, I decided to look for simple architectures that I could understand how to implement. Therefore, I decided to recreate two older models: LeNet, from 1998, and Alexnet, from 2012.

The first model contains two convolutional layers and three fully connected linear layers. Each of these layers is followed by a rectified linear unit (ReLU), and there are two max pooling layers after the first two ReLUs.

Given that the images were reduced at the beginning, they needed to be upsampled back after passing through the model. For this reason, the train function did some reshaping, upsampling and permutation of the output of the model. The loss function used in train was binary cross entropy, since it is a binary classification class, and the optimizer was Adam. The results of the loss were appended and plotted in a graph to see how it changed and whether all models looked the same.

After testing the model with the validation data and getting very similar results, two changes were made to the first model, which led to slightly different models. Firstly, we changed the learning rate from 1e-5 to 0.0001 (m1_1). Secondly, the kernel size of the first convolution was changed from (3,3) to (5,5) (m1_2). The idea behind making the kernel size bigger was being able to generalise better from the images.

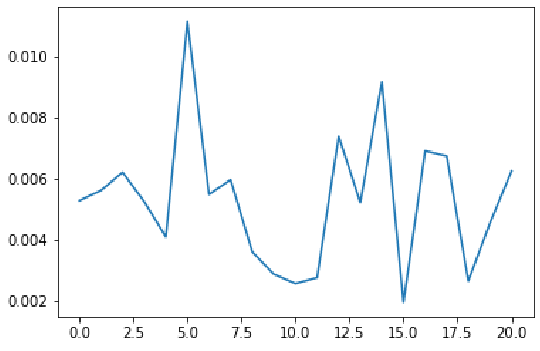
The second model is more complex than the first one and its variants. It has five convolutional layers and three fully connected layers. There is batch normalisation after all the convolutional layers and max pooling after the first two, and after the last convolutional layer. Then, there are three linear layers with dropout and ReLU. Finally, there is a sigmoid function.

After testing the model with the validation data, we could see that all the outputs of the tensor had a value of 0.5, which gave an accuracy of 0.0. The primary suspects were the dropout layers, perhaps given that the outputs of the model were particularly small, so they were commented out.

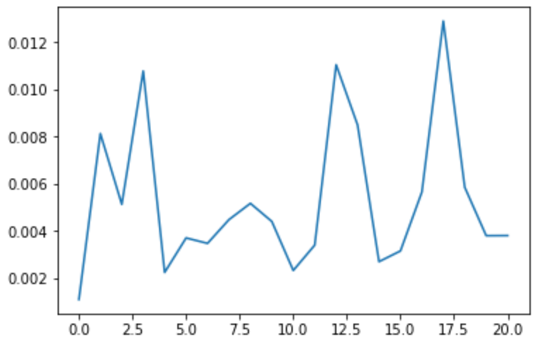
Part 3: testing and evaluation

The test function prints out the accuracy of the models, as well as the total error, computed with mean squared error, and its plot, after upsampling the images back to 2048x2048.

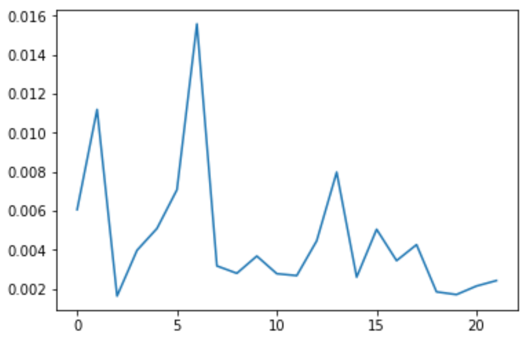
The accuracy for the first two versions of the first model, m1 and m1_1, are exactly the same: 28.97%. The mean squared error is almost the same (they differ on the 7th decimal), and the graphs seem to follow a similar pattern. However, the results do seem different in the validation pass of the testing function: m1 has a slightly smaller total mean square error and has less peaks, although they come to a higher value. This could suggest that using a smaller learning rate during training could yield slightly better results, slthough the test set did not verify it. The second version of the first model, m1_2, has a slightly lower accuracy: 26.93%. Its total mean squared error loss is similar to the ones of the previous first models, and the plot seems to have less peaks than m1_1. The accuracy of the second model is incredibly low: 2.92%. The total mean squared error is over 50 times higher than that of the previous models.



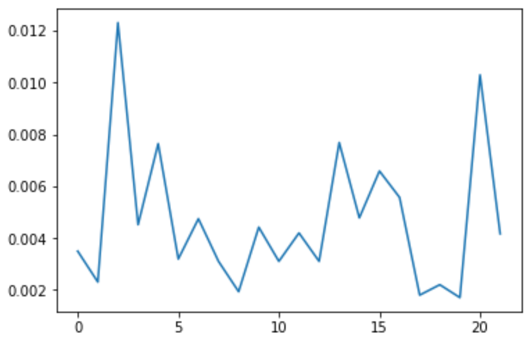
Graph 1 - mean squared error for m1(test set)



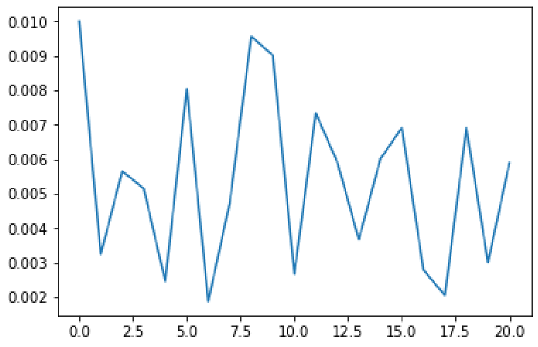
Graph 2 - mean squared error for m1_1 (test set)



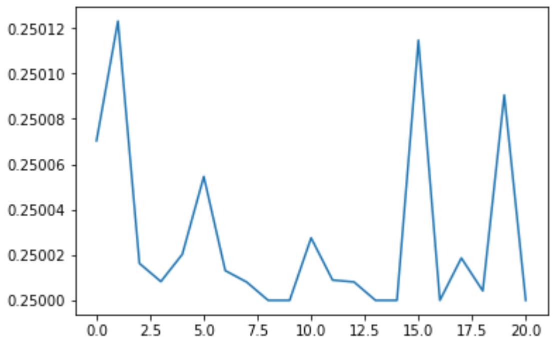
Graph 1 - mean squared error for m1(val set)



Graph 2 - mean squared error for m1_1 (val set)



Graph 3 - mean squared error for m1_2 (test set)



Graph 4 - mean squared error for m2 (test set)

Lastly, the function “showing_im” outputs the visual representation of the pixel probabilities superimposed on the original image. It applies the model to individual images and returns the output of the model. We can see that for the first model and its variants, the results are quite similar. There seems to be only a few areas where pixels are detected in the left side and centre of the image, which are marked in red in Image 1. We checked that the results of the model are not the same by adding the output of the function. This could indicate that the training images were all very similar, and simply given the location of Chinese characters might not be enough to train the models. The second model seems to output the detect either no pixels or just an area of them, which is not surprising given the accuracy of the model.

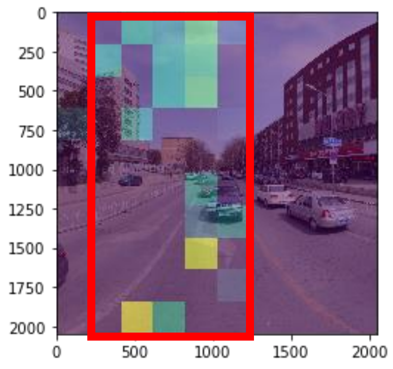


Image 1 – the representation of the output of m1 of a random image

References

https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d#e276

https://github.com/ChawDoe/LeNet5-MNIST-PyTorch

https://www.pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/