LT2326

Assignment 1 Report

Calvin Kullvén-Liao

**Introduction**

In this task, two CNN models were trained using a smaller portion of the dataset from https://ctwdataset.github.io/.

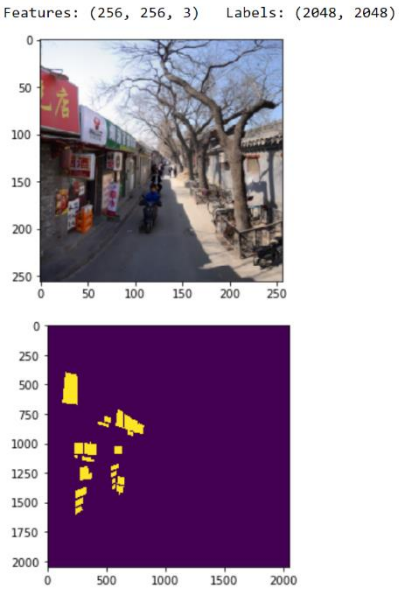
The following sections details the data pre-processing for supervised training, the models architecture, and the evaluation of the results/outputs of the trained models.

The main Python codes for pre-processing, model design and training, and evaluation are in the Assignment1.ipynb notebook. The two saved models cnn_model.pt and conv2d_model.pt are in the same GitHub directory. Note that if using test images as input, the images should be converted to a PyTorch tensor of shape (N, H, W, 3).

The input/output of the two models also slightly differs: cnn_model.pt takes resized images (N, 256, 256, 3) and outputs a prediction (N, 2048, 2048, 1); while conv2d_model.pt should be able to take images of unspecified size (N, H, W, 3) and outputs *two* predictions, both in the shape (N, 2048, 2048, 1).

**Data and Pre-Processing**

The data were processing using the 1000 jpg images on mltgpu server in /scratch/lt2326-h21/a1/images/ and the relevant jsonl files in /scratch/lt2326-h21/a1/ which contains information of an image's Chinese character bounding polygons.




The 1000 images-and-polygons-information were split to train and test by 85:15.

The images is processing into a list of (features, labels) where features in a rescaled image 3D array, and labels is a 2048 by 2048 binary mask array representing whether a pixel is inside or outside the polygons (created using rasterio.features and shapely.geometry modules).

Originally I did not want to downscale the images as it would be like training a model to look at thumbnails and detect which parts contains Chinese characters. However, the 2048*2048 image arrays keep overloading the GPU memory, so I used skimage.transform to resize the images to 256*256.

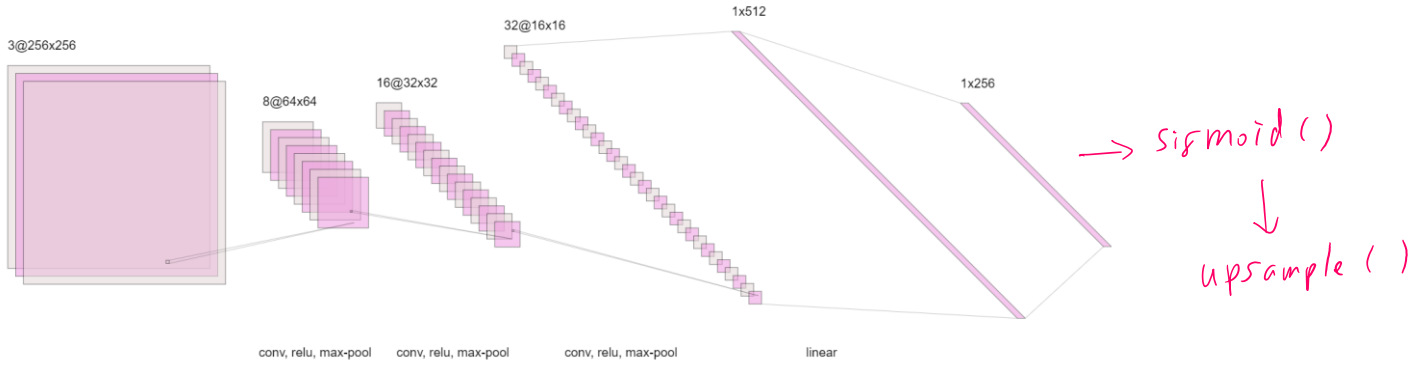
**Model Architectures**

2 CNN model classes were defined:

(1) LeNet

The architecture is inspired by https://d2l.ai/chapter_convolutional-neural-networks/lenet.html except that I changed the avg-pool to max-pool as the former performed poorly in my original testing.

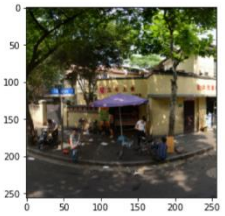
The model consists of multiple (conv2d, relu, maxpool2d) layers that increases the number of channels while downsizing the images. The downsized images then go through two linear function and a sigmoid function, then upsampled back to 2048*2048.

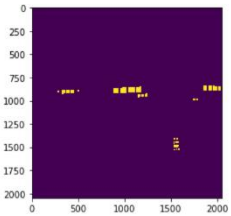


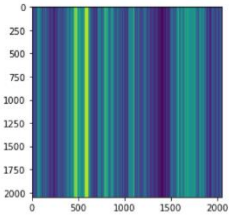

(2) Conv2d layers from Resnet18

This model has the middle layers similar to the ones mentioned on https://ravivaishnav20.medium.com/visualizing-feature-maps-using-pytorch-12a48cd1e573 with the 17 convolutional layers with different parameters from ResNet18, in the hope that these deeper layers would be able to extract the features, ie, the Chinese characters. The full ResNet with BasicBlock seemed complex so I did not attempt to replicate the whole architecture. Instead I only added the abovementioned layers consisting of (conv2d, batchnorm2d, relu), and optional maxpool and dropout in between. Then layer 18 is an AdpativeAvgPool function that shrinks the tensor to (num_channels, 1, 1), in this model being (512, 1, 1). The final two layers then linearise and take the sigmoid, and upsample the 1D tensor to 2048*2048.

**Results and Evaluation**



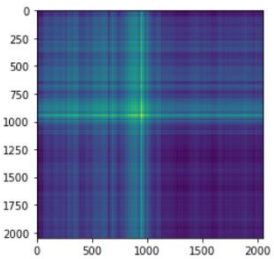


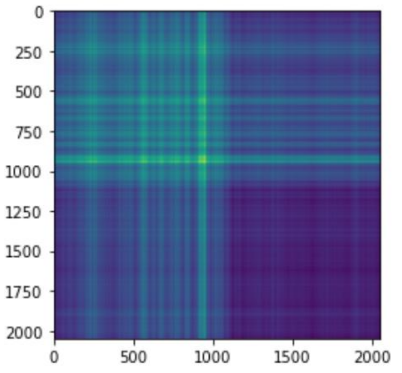


The output of the first LeNet model looks like a barcode, possibly because it was upsampled from the 1D linearisation to 2D. Upon inspection, the distributions, ie, the brighter parts, does correspond to the distribution of polygon areas. I originally upsampled the vertically, ie, (1,1, hiddien_size) to (1,2048,2048). Later I upsampled instead from (1, hiddien_size, 1) and the output (horizontal lines) seems to be better aligned with the actual polygon distribution, probably because the characters/polygons mostly appear horizontally in the images.

Therefore, I let model2 output both vertical and horizontal lines to see if they can be combined. (The loss during the training is simply the sum of two losses.) I later tried both adding and multiplying the two outputs, and multiplication seems to work better as the pixels with lower sigmoid get darker when multiplied.

```
plt.imshow(xyp_list[0][2]+xyp_list[0][3])
<matplotlib.image.AxesImage at 0x7f73ec488b20>
```



To evaluate the outputs, I used nn.MSELoss to compute the mean squared error, and the results look similar:

```
eval_model(test_xy, model=cnn_model, threshold=0.50303566)
```

```
Mean squared error: 3.767077684402466; avg 0.2511385122934977
Accuracy: 0.8716362118721008
```

```
eval_model(test_xy, model=conv2d_model, threshold=0.25389633)
```

```
Mean squared error: 3.8045142590999603; avg 0.2536342839399974
Accuracy: 0.6061683297157288
```

In terms of computing the accuracies, it is a bit complicated. First, I decided on the output heatmap that is to be compared to the y-labels mask: for model1, it is the original output, and for model2, I chose the product of its two outputs.

Then, using a threshold, the heatmap 2D tensors are turned into a 2D binary, for example, if threshold is 0.5, then

[… 0.05, 0.60, 0.32, 0.54 …] becomes [… 0, 1, 0, 1 …].

This binary tensor is then compared to the y-label mask which itself is also binary, to compute the number of exact matches. Accuracy is then num_matches / 2048*2048.

The problem is that it is hard to specify a single threshold or a range of thresholds, since the heatmap / binary heatmap are essentially upsample 1D values instead of a true 2D mask. Therefore the accuracy does not really reflect the 2D distribution of polygons.

**Discussion / Conclusion**

The 1D output of the models is not ideal for the purpose of this task, which is to get a 2D map of each pixel's probability of being in a polygon. I suspect that the linearisation/dense layers compressed the image to 1D and made it impossible to retain the 2D information. It may be useful for classification, etc, but not for the purpose of this task. Maybe a different model architecture is required to preserved the 2D information.

**References**

For design of LeNet https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

For design of Conv2d layers https://ravivaishnav20.medium.com/visualizing-feature-maps-using-pytorch-12a48cd1e573

Tool used to draw the diagram of model architecture http://alexlenail.me/NN-SVG/LeNet.html