Chinese character detection report

Anna Katarina Page

October 2021

1 Introduction

In this project I use images from a database of pictures of Chinese characters occurring 'in the wild' (i.e. on road signs, buildings, etc.) alongside with annotated versions of the images where the bounding boxes for each character are indicated to attempt to build two machine learning models with different architectures that will be able to 'learn' where the bounding boxes are in the images. That is, for each pixel, the model will generate a probability of being in a bounding box and will use that probability to make a prediction.

The models are both built using PyTorch and are executed on the MLTGPU. To run the model using ones own test images, run through the `simplified_duplicate_for_test_images` Jupyter notebook included in the Github repository to process your own files into the right format and run evaluation on them.

2 Data Preparation

2.1 Data 'pre-' pre-processing

I first access the data from Scratch and generate a list of all of all available files which are in the `images` directory. I then check that all images have the same height and width in pixels and that the `train` JSON file and check that the length is the same as `info.json` and then I filter down the entries in `traindata` to just those which are actually in `images`, leaving 845 images total.

2.2 Extracting polygons from all images

In this section I take an `image` entry from the `train.jsonl` file and process the JSON contents. The `file_name` is used to load the image itself into memory before converting to a numpy array. Additionally, the `annotations` data is used to create a matrix of 0s and 1s indicating whether or not a specific pixel is part of a bounding box.

Processing all images individually takes far too long, so the `joblob` library is used to run the processing in parallel threads for each subset of the data. This reduces the overall runtime of the processing from around 34 minutes to less than 500 seconds.

Each processed image returns a tuple, so the `joblob` library returns a list of tuples, where the first element is the input data (`x`) and the second element is the output data (`y`). Before splitting the data into training, testing, and validation sets, I first convert the list of tuples into a list of lists. Here, the first list is all the input data points, while the second list is all the target data points. I then use the `train_test_split` function from Sklearn to split our data. To create the training, validation, and test sets, I call this function twice. Once to split the data 80:20 and again to split that 80 into 75:25. This leaves us with a training set comprising 60% of the data, as well as a validation and test set comprised of 20% each.

3 The models

At this point the data has been processed, but we need an easy way to interface with the PyTorch models which will be defined later in this section. To achieve this, we implement a simple Dataset class which gives us the input and output data as vectors. These are passed into a DataLoader to batch and shuffle them for input into a model. The batch size can be specified at this point. In order for one of the models to run without exceeding the available memory on the GPU, I had to set the batch size unusuall low. The GPU device is also specified here.

3.1 Model 1

The first model I build is called `NoPoolsAllowed`. In this model I wanted to experiment with using strides rather than pooling in order to downsample the images. It is a very primitive model architecture. It contains two convolutional layers, the first with a stride of 3 and the second with a stride of 2 to downsample the image. Each is followed by the application of a sigmoid function. These are followed by an upsampling layer and another convolutional layer at a middle size, and finally, a last upsampling layer to return the image to its original size and a final sigmoid function. The forward layer begins by permuting the image into the right form, and then runs through each of the above layers in turn.

3.2 Model 2

In this model I was inspired by the dreadful performance of the previous model and my desire to get something resembling actual predictions to attempt to implement a simplified version of the U-net architecture [1]. It is aptly named `UnspiredNet`. The architecture for this model consists of two encoder blocks, followed by a bottleneck block, followed by two decoder blocks, and ending with a final convolutional layer and the application of a sigmoid function.

The encoding blocks shrink the dimensions of the images while increasing the channel size. Each contains two convolutional layers followed by batch normalisation and the latter by a relu function. The each end with maxpooling. Due to GPU limitations, the channels are only increased to 16 here.

The bottleneck block takes the pooled output of the previous encoding block and runs it through two convolutional layers applying batch normalisation after each. A pooling layer is then applied to this output.

The decoder blocks then reduce the channels again while returning the image to its original size using `ConvTranspose2d`. In each decoder block the transpose layers are followed by convolutional layers which in turn are followed by batch normalisation. As above, the final of these is followed by the application of a relu function.

After these blocks, there is one final convolutional layer which reduces the channels to one (so that a prediction can be made) and a sigmoid is applied to the output.

In the forward function for this model, the first step is permuting the input into the right shape. The input is then put through the encoder layers and the bottleneck layer, and the outputs are saved as variables so that some of them can be used for concatenation. In the first decoder layer, the first `ConvTranspose2d` layer is applied to the concatenation of the output of the bottleneck and the output of encoder block 2 (the output before maxpooling is applied). The rest of the layers are then sequentially applied. The second decoder block is run through similarly, but here the first `ConvTranspose2d` in the block is applied to the concatenation of the output of decoder block 1 and the output of encoder block 1 (the output before maxpooling is applied). Finally, a last convolutional layer is applied and the forward function returns the sigmoid function run on the output.

3.3 Training

Before training the model, I define a helper function that will run the training and validation for us. The function takes a name, a model class, a device, an optimizer, a learning rate and the number of epochs and does the training loop for us, before running through the validation data. For both datasets, the function stores the average loss per epoch and, along with the model parameters, saves these to a unique name. In the training of model 1 I used 32 as batch size. I was forced to reduce this to 2 for the second model due to memory constraints and to reduce the epochs to 5, which still took about 3 hours to complete. Being a simpler architecture, model one runs through the training significantly faster. The training function saves the model parameters and a path to the history file so that these can be accessed later.

4 Testing and evaluation

To test and evaluate the models' performance I define four functions. The first, `get_evaluation_variables`, takes takes the path to the model parameters and a GPU device and runs the test images from the dataloader forward through the model and calculates the loss. To turn the raw output of the model into predictions (that is, 0 for 'not in a bounding box' and 1 otherwise), the function calls the helper function `get_predictions` on each batch and extends the list of `test_predictions` with the output. The helper function applied a threshold and returns a 1 for items in the input over that threshold and 0 otherwise. The `get_evaluation_variables` returns the test_images, test_preds, test_truth, avg_test_loss, and the raw_preds.

Outputs from the above can then be turned into numpy arrays and fed into the function `get_evaluation_metrics` which takes the predictions, the ground truth and the average test loss and returns a DataFrame containing the loss, accuracy, recall, precision F1-score, and mean squared error for the test test.

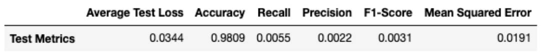
The function `display_training_validation_loss` can be run on the history file and is used to display the loss for the training sets and validation sets over time during training.

Finally, in order to visualise the models performance `visualise_probability_map` takes the images, preds, ground truth and optionally an index and a save path and displays the original image with the predicted bounding box pixels in green overlaid on it as well as the image with the actual bounding boxes overlayed on it. If no index is provided, an index is chosen at random. If a save path is specified the image is saved.

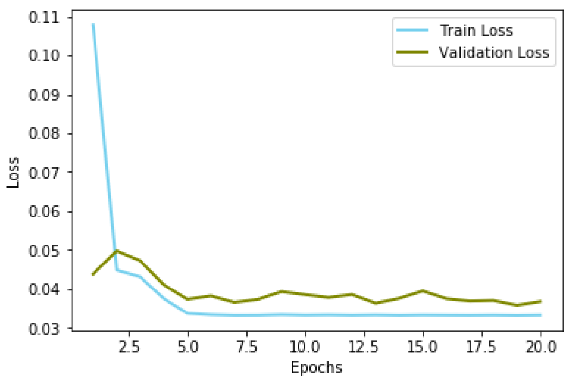
5 Results and Discussion

5.1 NoPoolsAllowed

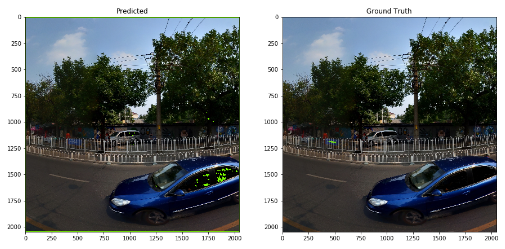
The first model I attempted did not perform well. The accuracy is high but all of the other metrics are very very low. Additionally, the loss does not improve much over time after a big jump after the first epoch. These facts can be seen below:



	Average Test Loss	Accuracy	Recall	Precision	F1-Score	Mean Squared Error
Test Metrics	0.0344	0.9809	0.0055	0.0022	0.0031	0.0191



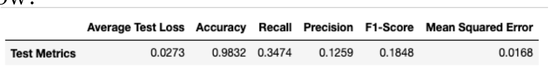
The accuracy is only high because it overwhelming 'learned' that almost all pixels are 0s. However, to even get it to predict any pixels as within the bounding box, the threshold must be set extremely low. For some reason, the model appears to predict that the 20 or so pixels along the top and bottom of the images is most likely to be a bounding box and predicts this for every image. Very few of the predictions have pixels guessed elsewhere. One example is the below:



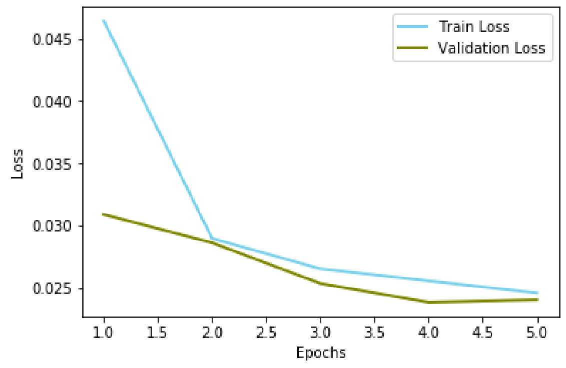
I ran though the training several times with different hyper-parameters but these did not have an effect on the performance of the model. Given the model's extremely simple architecture and the small amount of training data, this result is not particularly surprising.

5.2 UnspiredNet

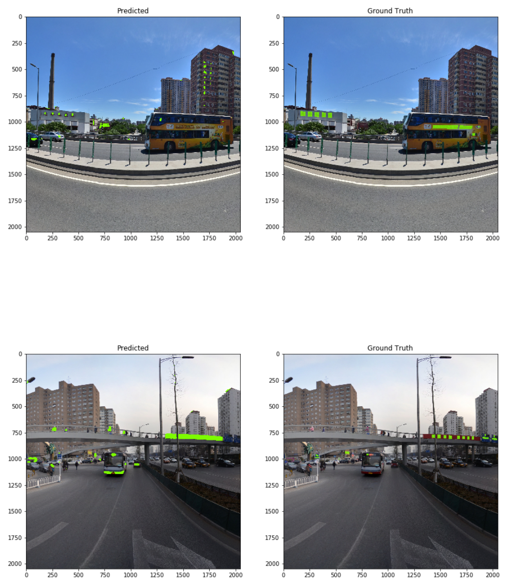
This models holds more promise. Although the recall, precision, accuracy, F-1 score and mean squared error are all still low, they are not nearly as low as in the previous model. These, along with the training loss over time are represented below:



	Average Test Loss	Accuracy	Recall	Precision	F1-Score	Mean Squared Error
Test Metrics	0.0273	0.9832	0.3474	0.1259	0.1848	0.0168



When I initially built this model, I had the channels set to a higher number which might have improved the models performance, but I was forced to lower this number on account of memory constraints on the GPU. Additionally, I was only able to train the model over 5 epochs. I would have liked to be able to see the performance over more epochs but as each epoch was taking about 45 minutes and the MLTGPU kept closing the connection, I limited it to just 5 epochs here. With those changes made and with more training data used, I think that this the success rate of this model may improve further. I would also like to have been able to test the model with other hyper-parameters changed (such as the learning rate), but as above, with the time each epoch was taking, this was not possible here. The below are example of an image where the model correctly predicts the locations of a number of the bounding boxes in the image.



In both of these, some of the bounding box predictions are accurate. Other predictions, although they are inaccurate, subjectively seem to be fairly reasonable guesses. The incorrect predictions generally take the shape of bounding boxes and appear in the right sorts of places in the image, for example. along building edges and across the front of vehicles.

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.