Jacob Coles

LT2326 Assignment 1

Chinese Character Detection

1. Introduction

Given the now 'old' task of recognising specific characters in text, there remains the issue of identifying exactly where these characters exist within an image. In this report we'll explore the task identifying where signage exists within an image by writing two models in PyTorch.

Part of the difficulty of this task is exacerbated by the lack of data, and due to the fact that we are working with Chinese characters, it will be incredibly difficult for a system to identify individual characters as a way of helping detecting signage. Thus, the eventual system must be able to identify "character-y-ness"; what the boundaries of a character in an image are, based on what a character/text 'should' generally look like, without necessarily identifying what the exact characters are.

2. Preprocessing

We are going to attempt to create a model which outputs a 'binary map' of where the signs should be within the images. In order to do this, we will need a 'gold-standard', in order to train the model what to look for. To create this, we will use the polygons (which are included with the image dataset), to create an image (one associated with every input image) with the same dimensions as the input images.

We can use the Polygon class from matplotlib.patches, which will allow us to use the .contains_points method in order to see whether or not a point lies within the polygon. We start by making an empty binary image, by instantiating an empty boolean numpy array with the dimensions of the desired image (2048*2048 in this case); we can then check all the points in the image as to whether or not they are within a polygon, and mark them with a 'True' (in a polygon) or 'False' (not in a polygon). As a speed-up, we only need to check the points within a range that the polygon could possibly exist, which is possible since we know the maximum and minimum thresholds (dimensions) of the corners of the polygons.

Now that we have all the data we need, we can create two associated lists; one of which contains the original images, and the other containing the 'gold-standard' images (which can be converted to pytorch tensors for ease of use). After this we can split into our train, test and validation sets, which I decided to set at 70%, 15% and 15% respectively. We can then use DataLoader() to create the batches for the images (for each of the sets). Note that the size of batches will be changed under testing because of eventual problems with memory on the GPUs.

3. Models

3.1 Research

A summary of the research found on these types of images, are that they almost always use a two-stage compression-decompression model. The first stage consists of a 'compression', in which 2D convolutions are used (often in combination with other steps) to reduce the image in dimensions, while increasing the number of features.

Auto-encoder-decoder systems [2] have existed for some time, and are very effective for compressing information in an image using convolutional networks, and have various architectures for decoding the information depending on the application.

Image segmentation is a specific problem under current investigation in research especially for its applications in the medical field, and there are a number of models, such as those presented in the paper "Machine learning techniques for biomedical image segmentation" [3].

A specific example of one similar to that presented in this paper is U-net [4] which compresses a representation of an image using several stages of convolutional layers with max-pooling (increasing the numbers of features with each convolution step). After generating a 'compressed' version of an image with many features, the system 'decompresses' the image by doing several stages of a transpose-convolution, (regular) convolution and upscaling, as well as concatenation with a downscaling of the images from the compression step.

Many of these models have a high degree of complexity, so in our models, we will attempt to boil down some of the more important architectural structures.

3.2 General Design Decisions and Testing Parameters

Selecting the loss function is largely dependent on the type of data we have. Cross-entropy works well for classification problems with a softmax, mean-squared works well for regression problems, and binary-cross-entropy works well for binary classification tasks. Given that our data is binary, binary-cross-entropy seems appropriate. Batch normalisation was used between the convolutional layers to help prevent gradients blowing up, and enable the better generalisation.

3.3 Model 1

This model is intended to be an extremely simple model that utilises convolutions for compression (torch.nn.Conv2d) with max-pooling (torch.nn.MaxPool2d) and a leaky-Relu (torch.nn.LeakyReLU). The system then passes the 'compressed' representation with 512 layers and a dimensionality of 32*32 into the up-sampling/decompression part of the model. This part of the model uses a combination of transpose-convolution (torch.nn.TransposeConv2d) and un-max-pooling (torch.nn.MaxUnpool2d(), which uses the optional output of max-pooling of the downsampling layers to 'reverse' it).

3.4 Model 2

The design of this model was originally loosely based on the previous model. The key concept was adding more generalising/reasoning-ability to the system by adding a couple of linear layers in the middle of the model between the encoder and decoder.

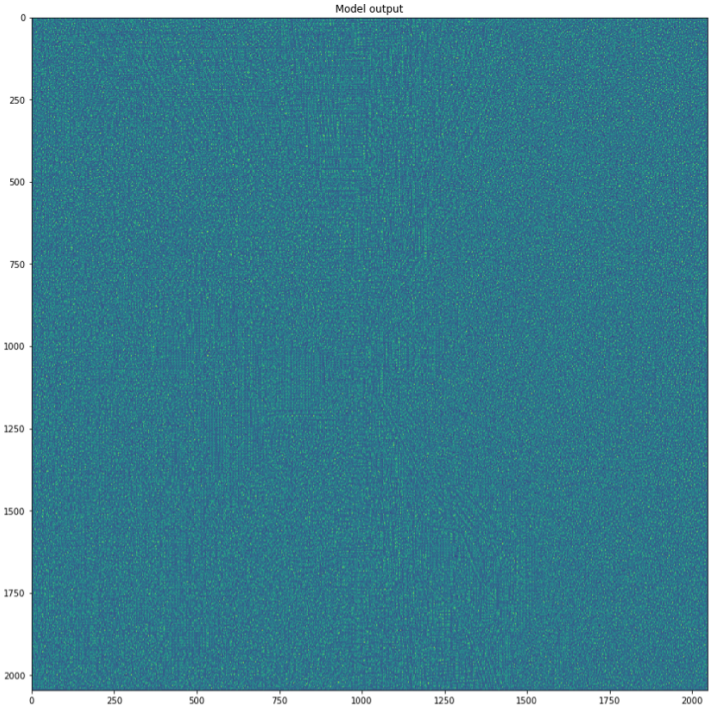
Convolutional layers compress the information, feeds it into a set of linear layers, and then upscales using transpose convolution. In order to coerce the data to fit into the linear layers, it was necessary to reshape/flatten the data and then reshape it into the form of an image on the other side.

4. Results

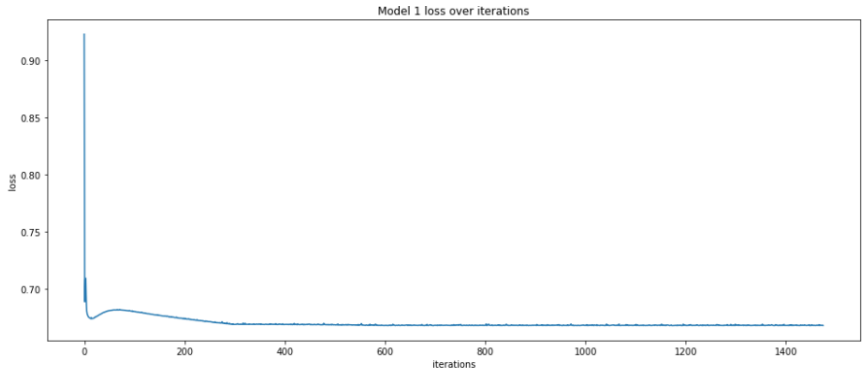
4.1 Model 1

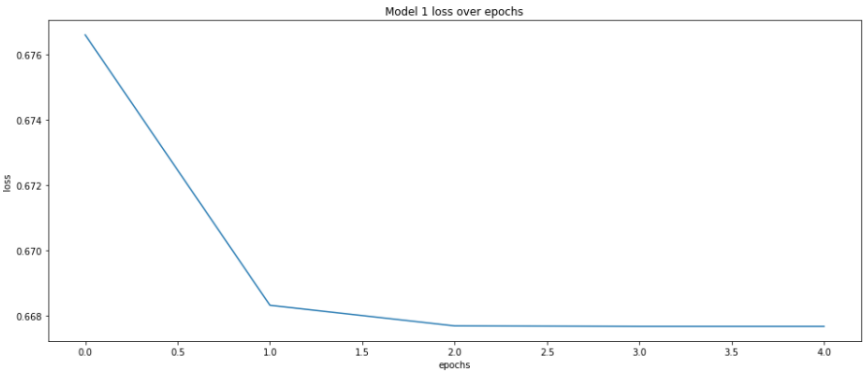
Model 1 achieved a mean-square error of 28.116407492603933 in the validation set, and 28.118389697549716 in the test set.

An example of an output is as follows:



We can see that it doesn't seem like the system is learning anything. This seems to be true as the loss doesn't seem to change as the system runs out of data after the first epoch.

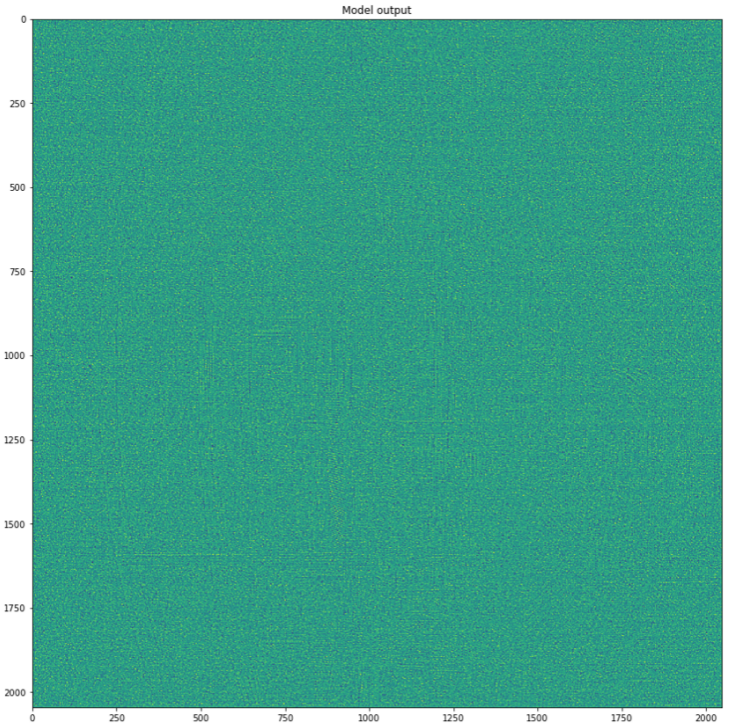




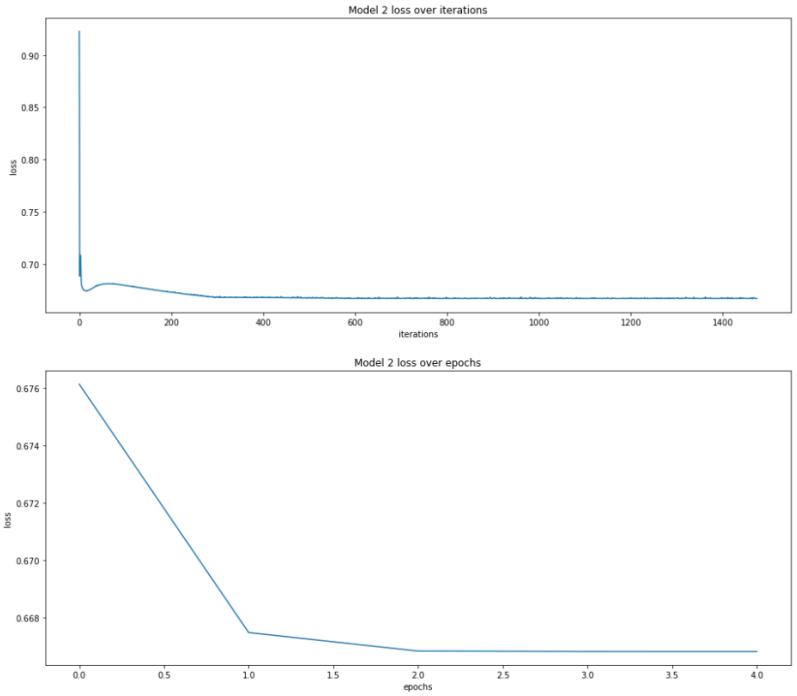
4.2 Model 2

Model 2 achieved a mean-square error of 29.402526120896386 in the validation set, and 29.402192702332542 in the test set.

An example of an output from the test set is as follows:



This result is similar to the results for model 1:



5. Discussion:

Testing was done to establish an appropriate number of convolution/ transpose-convolutional layers to be used in the model. Having an increased number of layers on the encoding and decoding step past 4 layers seemed to decrease apparent performance. Another issue was that of dimensionality of the compressions and a lot of tuning was done to attempt to improve the situation, however to no avail.

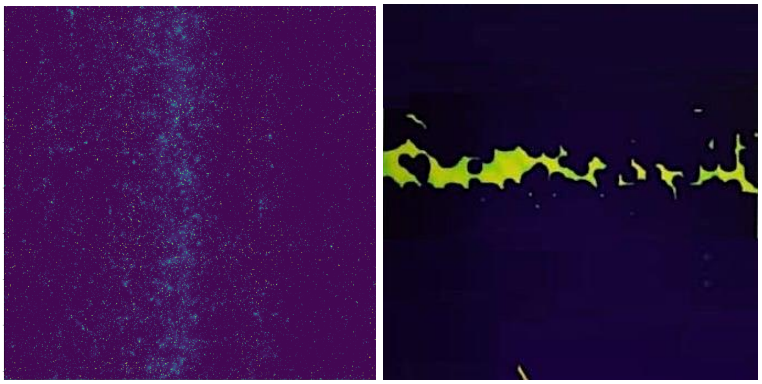
The system's results varied based on the hyperparameters chosen. In most cases the loss converged on a particular value after a single epoch and barely shifted from there and there was a difficulty in improving this. This is perhaps this is due to the fact that there wasn't enough data to learn any trends. In the best of cases, that is even when the loss was a low value, the model wasn't learning any 'useful' information; only a black screen.

A major oversight was the effect of dropout on the performance of the models. At one point, often with the models was doing very little but outputting a blank screen. The effect of having low or no dropout layers was that no training was happening. The loss was changing by no more than 0.01. Through experimentation it was learned that lower loss could be achieved by having a very low dropout on the transpose-convolution layers (tested as p=0.02), except with a moderate (p=0.3) dropout on the (regular) convolution layers (in the decoder, wedged between transpose-convolution layers) would result in an average loss of ~0.08. For comparison, when the reverse of these values values was tested, it resulted in an average loss of ~0.31, and when both were set to p=0.3, the average loss was 0.22.

Unfortunately, due to some modified hyperparameters, and the model wasn't able to reproduce these results at a later stage. However, despite these models achieving quite low-loss, the output was simply a blank screen (all zero outputs), so it was decided that some more remodelling could be done hopefully to improve on this and get some valuable output.

After further research, it was revealed that perhaps dropout wasn't the most effective method to avoid a model over-fitting. Though changing dropout was effective, it caused the model to have a very unstable change in loss which makes optimising for a model extremely difficult. Dropout generally works well for linear layers by restricting some of the parameters being trained all at the same time, however this is not the case (necessarily) for convolutional layers. As an alternative, a normalisation function was used. This also had the benefit of preventing gradients blowing-up, which happened on occasion.

A final thought; the times that the model seemed to output something "reasonable", it seemed to be an anomaly. This was especially true early in designing the system when it was simple with few layers in it. In some cases, the model was overfitting such that regardless of the image chosen it would always show the same output, wherein on other occasions it was a 'speckled' mess of pixels that appeared uncorrelated to the gold-standard. It was assumed that the model wasn't actually learning anything, and that in many cases where it seemed to be producing an 'interesting' output, that there was an issue in the design. An example of an output here shows that it was picking up that there were more things happening across the middle horizontal section of the image (I discovered at this point that the image was accidentally rotated sideways from mixing the x-y dimensions). The other example simply shows splotches, in a similar part of the image, but every image looked exactly the same and nothing was being learned. In both of these cases the decoder part of the model was extremely short and there probably weren't enough parameters and information being passed through the system to produce an appropriate output.



6. Conclusion:

There were many parameters that were tuned, and many variations tested, however with such a small dataset, achieving reproducible and consistent results was extremely difficult. Despite this, we were able to produce a system which was able to process images and produce an output with a reasonable loss value, even if in this case the output leaned towards producing false-negatives rather than false positives.

References:

[1] Srivastava N., Hinton G., Krizhevsky A., Sutskever I., Salakhutdinov R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In: Journal of Machine Learning Research

[2] Abdelaal A. (2020). Autoencoders for Image Reconstruction in Python and Keras. Accessed 18/10/2021. Available at: https://stackabuse.com/autoencoders-for-image-reconstruction-in-python-and-keras/

[3] Seo H., Khuzani M.B., Vasudevan V., Huang C., Ren H., Xiao R., Jia X., Xing L. (2020) Machine learning techniques for biomedical image segmentation: An overview of technical aspects and introduction to state-of-art applications. In: Medical Physics

[3] Arora A. (2020). U-Net: A PyTorch Implementation in 60 lines of Code. Accessed 18/10/2021. Available at: https://amaarora.github.io/2020/09/13/unet.html

[4] Park S., Kwak N. (2017). Analysis on the Dropout Effect in Convolutional Neural Networks. In: Lai SH., Lepetit V., Nishino K., Sato Y. (eds) Computer Vision – ACCV 2016. ACCV 2016. Lecture Notes in Computer Science, vol 10112. Springer, Cham.

[5] SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation