UNET Target Object Detection

Siddharth Diwan
Brown University

I. Project Summary

This document gives a (first-person) progress report of my starter project *Finding Blueno* that detects Blueno pasted on random background images. The code has been made generic such that one should be able to use any choice of target object and retrain the model.

The goal of this project is to reliably generate target masks (of Blueno) from images.

II. Dataset Generation

I aimed to paste foreground objects onto background objects. For this reason, my first step was to find a public repository of random background images. I chose the Stanford Background Dataset. My next step was to create the Blueno target object itself. For this, I took a generic Blueno picture and removed its background.

From the locally downloaded background images, I decided to create the images to classify by foregrounding Blueno on each of the background images using the PIL library. At initial stages of the project, I was only foregrounding Blueno at the center of each background with a fixed size. But, I soon incorporated the ability to use random scaling and vertical and horizontal translation coefficients to vary the distribution of Blueno on the background images.

I have made a conscious decision to ensure that Blueno is never cut from the borders of the image in the hopes of boosting model accuracy. This is by making the translation coefficients dependent on the scaling coefficient and the dimensions of the image. The disadvantage of this assumption is that such a practice is a dangerous assumption of what kinds of object detection tasks may be required by users. Moreover, it meant that I needed to resize each processed image before processing or generating its true mask.

For each of the processed images, I generated a ground true mask by superimposing Blueno on a white background, and converted the image to only store Luminance using a grayscale function against a target pixel threshold of 255. Making true masks would also help me avoid using a confusion matrix when determining the accuracy of the model in later stages of the project.

III. Model

I stumbled across the UNET model in one of my readings about image detection. I decided to implement it using the published architecture.

After analysing the model, I made a few implementation choices. For instance, I tried to take advantage of the repeated Double Convolutions by modularizing the forward convolutions into a separate class. Moreover, I tried to represent each of the up and down layers separately, although in retrospect, I could have potentially also made do with a ModuleList to reduce repeated layers. Finally, to tackle the cropped pasting of down convolution results on up convolution inputs, I cropped and stored the down convolution results in an array and concatenated them for the up convolution steps.

Overall, for my UNET implementation, I assumed input tensors of four dimensions representing the batch size, number of color channels and the image dimensions.

IV. Loaders

The resizing of images during data processing was driven not only by the need to ensure Blueno was always completely within the background, but also because the Stanford Dataset contained images of slightly varying dimensions. But, I realized that cropping the images during processing would be a very time-intensive way of normalizing the data. Predicting that the model would take a long amount of time to load if the dimensions are relatively large, I decided to transform each tensor generated from every processed image to give myself more control over the training and testing time.

Initially, I had attempted to convert the mask and image separately into a 3D array of pixel values using the Imageio library, and pass these to a TensorDataset and finally to a DataLoader. What I had neglected to see is that Pytorch's from_numpy method did not entertain object types. For this reason, I made use of the Albumentation Library. When creating augmentations using the albumentations, I tried to transform each image and associated mask separately, but this led to issues with the creation of the TensorDataset from the image and mask tensors. Thus, I settled on jointly augmenting each image and its mask together, and separating the augmentations to get the train pixel inputs and target mask outputs.

At this stage, when I attempted to train the model on the train data loaders, I realized that my model was expecting outputs of 1 channel, so I must transform my mask tensors to only store Luminance just as with dataset mask generation. To make the conversion of image to pixels consistent for images and their masks, I decided to use the PIL library and its convert method.

V. Training and Testing

I made use of the Adam optimizer and BCE with Logits Loss Function, a learning rate of 1e-5, and an epoch count of 3. There parameters were determined by trial and error over multiple different choices.

To get an accuracy measure, I took the prediction tensor generated by the model over the train dataset and passed it through a sigmoid function to get a resemblance to the true mask's tensor dimensions. Using this, I counted the number of matching pixels between the prediction and target mask tensors in proportion to the total number of pixels represented in the prediction tensor.

One design choice I adopted was to to accumulate accuracy instead of calculating it over a validation set. This was partly because I decided to move the model.train method from the accuracy calculation method to the top of each epoch calculation. Another reason for this is that I had ground true masks available for a training set, which I felt would be a more useful metric.

Initially, the Train/Test split of data occurred such that for every 9 images in the train set, 1 was added to the test set. I made use of the fact that the dataset was being fully generated to directly segment the train and test images and masks during dataset generation, instead of using a train test split module. This way, I was able to independently configure the train and test modules to link to independent train and test folder directories. Using a 9:1 ratio, however, gave me a test accuracy higher than the train accuracy. For this reason, from trial and error, I noticed that a 3:1 train:test proportion would serve me best. I also ended up adding Batch Normalization to the model, which also helped.

When running and training and testing, I noticed that it was getting increasingly frustrating to be unable to gauge the process of the training. So, I wrapped the train and test DataLoaders within tqdm loading bars to get a real-time estimate of the model's train and test states.

VI. Prediction

After training and testing the model, I made it such that the model can be saved to a target directory for mask prediction ability on a user's choice of images.

At first, when I attempted to convert each of the images within a target directory to its pixel tensor representation, I realized that I was not providing the bath size. For this reason, I mirrored the process of DataLoading each image's pixels, this time creating a Tensor Dataset with two duplicate tensors, the second of which is a dummy, and passed this to a test_loader. The dummy was required because I did not expect users to provide known true masks for their test images. Finally, in a similar method to converting the prediction tensor into a Luminance converted image pixel representation for accuracy, I used the pixel representation to convert each image tensor back into an image.

One important mistake I had initially made is that I was shuffling even the test DataLoader, which was resulting in original images and their predictions being mismatched upon saving the processed masks. By setting the shuffle to False, I was able to alleviate this issue.

VII. Results

Accuracy	
Train	Test
85.12%	84.67%

The predicted masks are most accurate when Blueno is large (1, 2).

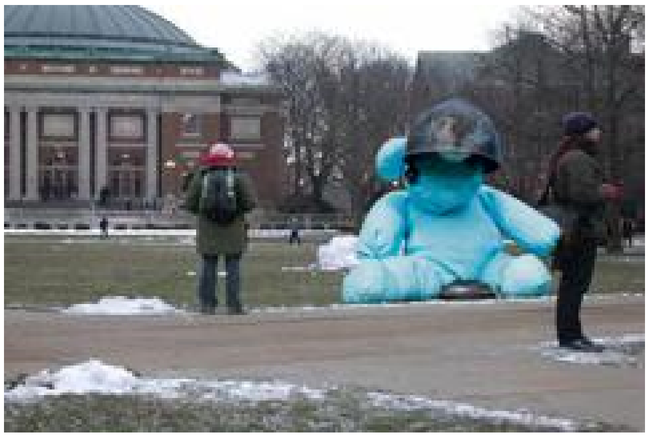

Fig. 1. Processed Image: Large Blueno on Random Background.

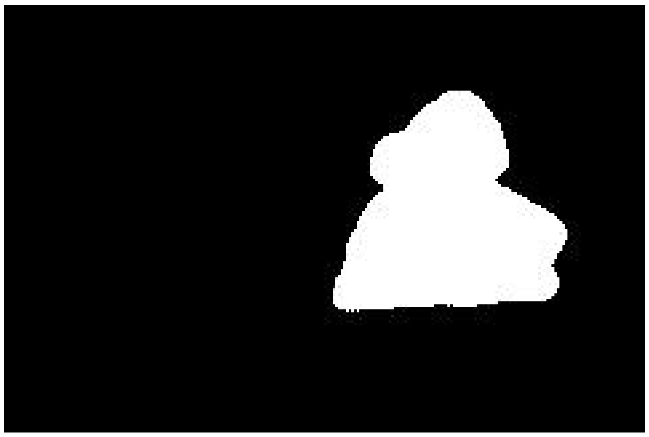

Fig. 2. Prediction Mask: Large Blueno on Random Background.

The model also performs well for smaller Blueno when the background is distinct from blue (3, 4).

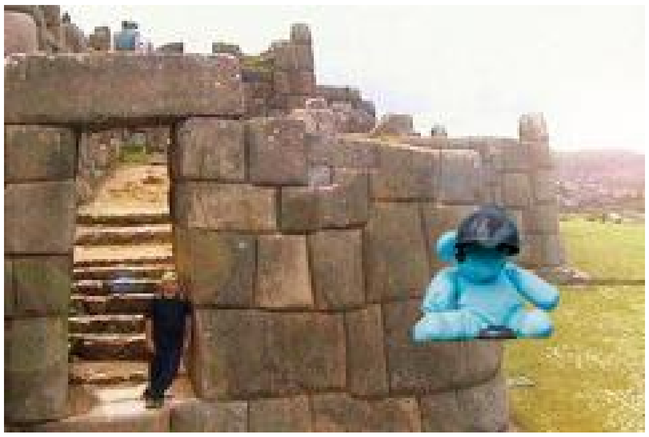

Fig. 3. Processed Image: Small Blueno on Distinct Background.

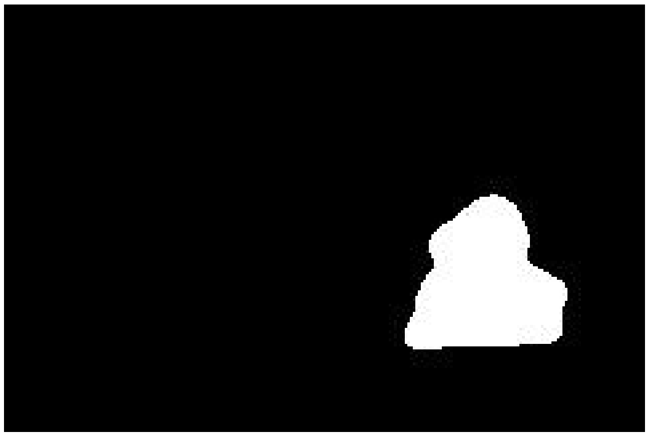

Fig. 4. Prediction Mask: Small Blueno on Distinct Background.

The model's perfomance, however, begins waning if the background has a blue hue (5, 6).

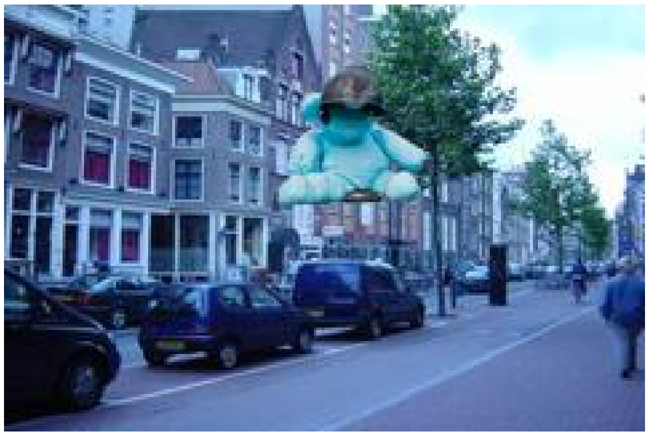

Fig. 5. Processed Image: Small Blueno on Blue Hue Background.

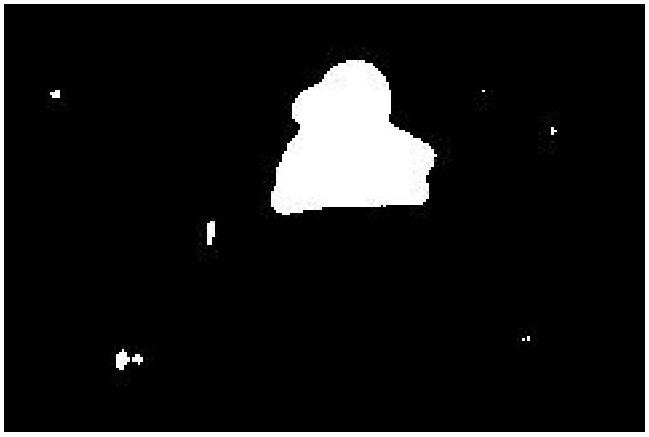

Fig. 6. Prediction Mask: Small Blueno on Blue Hue Background.

Given that scaling down Blueno makes its dimensions rounded given the loss of edge pixel data, the underperformance of the model for smaller Blueno in some contexts can be justified. For instance, the performance issues are representative when the image contains a small Blueno and other blue objects (7, 8).

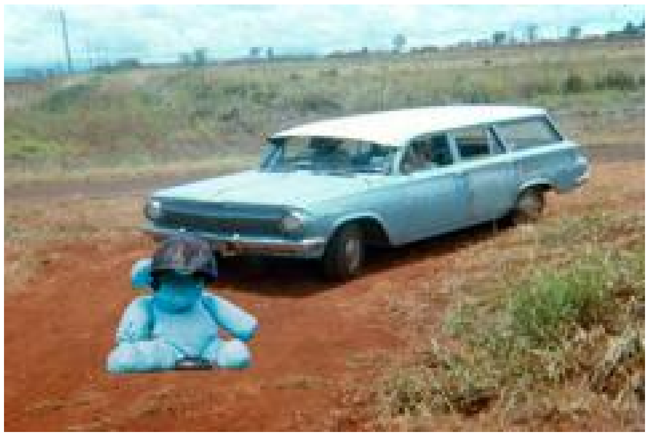

Fig. 7. Processed Image: Blueno with Surrounding Blue Object.

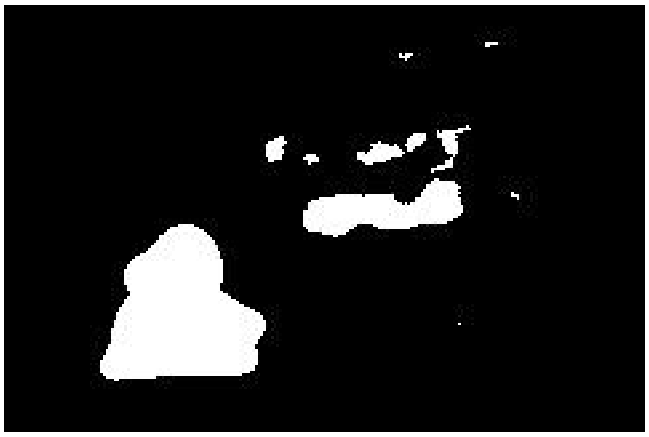

Fig. 8. Prediction Mask: Blueno with Surrounding Blue Object.

VIII. Conclusion

Whereas the model achieves decent results for high contrast backgrounds and larger images of Blueno, it often underperforms for smaller renditions of Blueno. Considering that data is lost when scaling down images, this is to be expected. However, the small dataset size also contributes to the underperformance.

As an improvement, I could attempt changing the model to use Convolution Transpose 2D Layers instead of regular Convolution Layers, and I could attempt to choose a more sharply defined target object that does not turn rounded or blob-esque when scaled down.