House Number Detection Model

By Andrew Henkel

Abstract

I trained a Faster R-CNN model in PyTorch for the purpose of reading in images containing house/building numbers and then outputting the digits it detects. This would be beneficial to a companies like Google which need to associate street-level photos with the addresses of the buildings in those photos. My model was trained for several epochs on Stanford's Street View House Numbers dataset (provided by Hugging Face) using the SGD optimizer and the StepLR learning rate scheduler. After training, I used my model to detect the digits in several photos and then assemble those digits into full building numbers. While the model usually detected the digits correctly, there were a few exceptions, and I provided several explanations as to how these issues can be addressed.

Full Paper

The goal of my final project was to build a tool for identifying house and building numbers. This could be useful when a company like Google sends cars down the streets of cities to take photos for its Street View. There needs to be some way to associate a 360 degree photo with an address, and being able to determine the numbers of the buildings in the photos is the first step towards that. My hypothesis is that I can build or fine-tune a convolutional neural network to detect the digits printed on the sides of buildings and then use the digits it detects to form whole numbers.

Figure 1: An example of how building numbers and addresses appear in Street View in Google Maps.

For this project, I chose to use a Faster R-CNN model with a ResNet-50-FPN backbone. A R-Convolutional Neural Network attempts to address the problem of a variable number of objects existing in a photo by dividing it up

into two thousand regions and feeding it into a 4096-dimensional feature vector that a CNN extracts features from (Gandhi). Fast R-CNN and Faster R-CNN differ in that a feature map is generated first, and then regions are identified and transformed so that they can be fed into a fully connected layer, thus allowing the whole model to train and predict much more quickly (Gandhi). The ResNet-50-FPN backbone is comprised of 50 layers that ensure that it can learn much more than other image recognition models (Gandhi). I chose this model not only because it seems to be a decent choice for image detection, but also because PyTorch contains a built-in function for obtaining this model (*Fasterrcnn_resnet50_fpn — Torchvision 0.18 Documentation*). (I could have also used a version of this model that was pretrained on the COCO dataset, which is what was done in the tutorial I was following to help me with part of my code (*TorchVision Object Detection Finetuning Tutorial — PyTorch Tutorials 2.3.0+cu121 Documentation*). However, that dataset contains primarily images of ordinary objects, and since I was focused on digits, I opted not to use this pretrained version.)

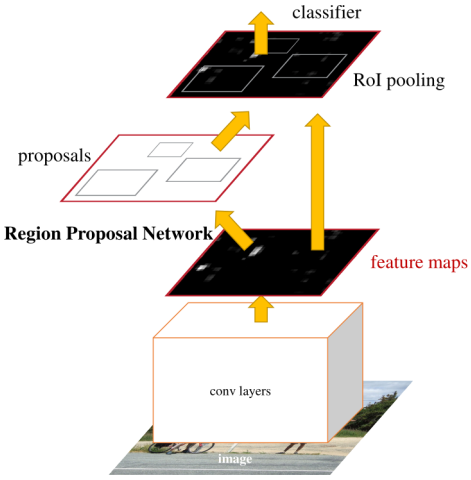

Figure 2: This image was taken from Faster R-CNN: Towards Real-Time Object *(Ren et al.). It shows how the Region Proposal Network provides the attention mechanism for the network.*

Before I even implemented my model, I first had to obtain a dataset that I would use for training. I chose Stanford's Street View House Numbers dataset for this (*The Street View House Numbers (SVHN) Dataset*). The dataset comes in two formats. The first format is for object detection and contains images with bounding boxes and labels for the digits detected in the bounding boxes. The second is for detecting single digits and only contains the images and labels, similar to the MNIST dataset. Since I want to detect multiple digits in a photo, I need to use the first format. Unfortunately, PyTorch only contain built-in support for the second, not the first. I could have manually downloaded and imported the dataset from the Stanford website, but that would have been very difficult, especially since the dataset was (if I understand correctly) originally intended to be used in MATLAB. Fortunately,

someone had already transferred the SVHN dataset to Hugging Face, so I could easily import it using Hugging Face's libraries (*Svhn · Datasets at Hugging Face*).

Since I chose to build upon an existing PyTorch model with its own idiosyncrasies, I had to adapt the SVHN dataset so that the training data could be fed in. The original dataset contained two features: images and targets (which are dictionaries containing the labels and bounding boxes). I had to pass the training data into a CustomDataset class, where each image, list of labels, and list of bounding boxes would be converted into tensors running on the GPU. In the process, I had to change all instances of 0 in the labels to 10 because PyTorch's fasterrcnn_resnet50_fpn model considers 0 to represent the background class during object detection. I also had to modify the bounding boxes since they came in the format of [starting X value, starting Y value, width, height], and PyTorch's fasterrcnn_resnet50_fpn model only accepts bounding boxes in the format of [starting X value, starting Y value, ending X value, ending Y value]. I passed the custom dataset to a DataLoader that would be used as an iterator, serving tuple pairs of images and targets in batches of six. This was the highest batch size I could use without causing the T4 GPU to run out of memory.

After this, I finally constructed the model that I would use for my project. First, I used PyTorch's built-in functionality to download a version of a fasterrcnn_resnet50_fpn model and move it to the GPU. Then, I replaced the box indicator in the model's region of interest with a FastRCNNPredictor object intended to detect eleven different classes of objects – ten for the digits and one for the background. For the training process, I also created an instance of the SGD optimizer with a learning rate of 0.005 and an instance of the StepLR learning rate scheduler with a step size of 10 and gamma of 0.5. The SGD optimizer is good for object detection, and the StepLR learning rate scheduler multiplies the learning rate by the gamma value throughout training to speed up convergence.

After the model, optimizer, and learning rate schedulers were set up, I trained the model for four epochs. (I wanted to train for more epochs, but each epoch had taken more than two hours.) Table 1 shows the training loss of the model after each epoch.

Table 1: Table showing the training loss decrease over time.

Epoch	Loss
1	1638.4
2	1389.5
3	1332.74
4	1295.83

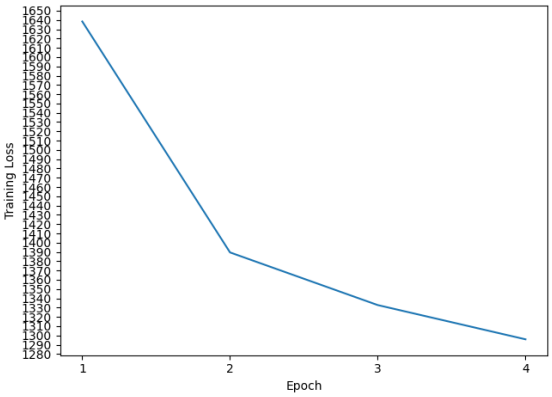

Figure 3: Graph showing the training loss decrease over time.

After training, I wrote code that fed an image from the test dataset into the model after putting it in evaluation ("eval") mode. (This test image had a building number of 649.) The model then outputted the bounding boxes and labels it detected from the image along with the scores indicating how confident it is. I removed everything with a confidence score less than 0.8. I also applied the torchvision.ops.nms function with a threshold of 0.4 to remove overlapping bounding boxes until that no redundant ones remain. (It accomplishes this by calculating the intersection-over-union score between two bounding boxes to determine how much they overlap, and if the IoU score is higher than a specific threshold, then only the bounding box with the highest score will be kept (*Nms — Torchvision Main Documentation*).) The remaining labels were [9, 4, 6]. Unfortunately, these digits are out of order. To address this, I sorted the labels based on where their bounding boxes appear horizontally. (For example, the label associated with the leftmost bounding box will be considered the first digit.) The end result was [6, 4, 9], which is correct.

After doing this, I copied my code into a separate function that performs similar operations: it reads an image from the filesystem, removes its alpha channel, passes it to the model, filters out the resulting entries that overlap or lack a high enough score, and prints out its findings. (The minimum score was originally 0.8, but I soon changed this to 0.7, and I'll explain why in a moment.) In addition to this, it

also displays the same image, this time with the bounding boxes over each detected digit. When doing all this, I made sure that all instances of 10 in the resulting labels will be converted to 0.

The model accurately detected the digits in most of the images I've tested it on, thus confirming my hypothesis. (Most were the ones I used for testing were from the "test" part of the SVHN dataset, but a few were from other websites, and I included them in my notebook and in Figures 4 and 5.) However, there are a few cases where its performance is subpar. A good example of this is when I used my model to detect digits in the "test2.jpg" file, and the smaller 1 wasn't detected. There could be several reasons behind this. Even though the training process took several hours (and a lot of frustration with Google Colab frequently disconnecting my session), perhaps it should have gone on for more epochs. Alternatively, perhaps I should have chosen a different optimizer or a different learning rate scheduler. Even though the training loss rate was worse when I tried the Adam optimizer instead of SGD, I might have had more success with a different optimizer. Perhaps I should have kept the SGD + StepLR combination but spent more time fiddling with their parameters. Finally, perhaps I should have used a different type of model instead of a Faster R-CNN one with a ResNet-50-FPN backbone. As mentioned before, I had chosen this model because PyTorch has built-in support for it. However, it might have been better to chose different

Figure 4: The "test.jpg" image. The 5 and 6 were detected correctly.

Figure 5: The "test2.jpg" image. The first 1, 7, and 8 were detected correctly, but the second 1 wasn't included in the final output.

model. (Although, to be fair, Faster R-CNN likely performs better than some other models like YOLO, which apparently “struggles with small objects within the image” (Gandhi)).

When considering these possibilities for how my model can be improved, one thing that needs to be taken into account is that certain digits might actually get detected, but their confidence scores are simply too low. For example, when I had initially tested my model on “test2.jpg”, the first 1 wasn’t detected because it has a score of 0.753, and I only kept digits with scores of at least 0.8. This was fixed after I changed the threshold to 0.7, but the second 1 still wasn’t detected. In fact, it was only detected after I lowered the threshold to 0.3, but that introduces the risk of including false-positives in the final output. In this scenario and in others, there must be an ideal middle point where the threshold is large enough to avoid false-positives but low enough to avoid false-negatives.

To conclude, this project has been very educational in terms of teaching me how certain kinds of convolutional neural networks work and how to go about training a model for object detection. The end result of all this was that I was able to produce a model that does a decent job at detecting digits in a photo. I’ve also come up with a few ways in which the model can be improved or why an entirely different type of model might be necessary for this purpose. Even though these considerations would have to be taken into account if a company or organization like Google was to actually use a model for detecting building numbers, my current model seems sufficient at the moment.

References and Publications Used

Fasterrcnn_resnet50_fpn — Torchvision 0.18 Documentation. https://pytorch.org/vision/stable/models/generated/torchvision.models.detection.fasterrcnn_resnet50_fpn.html. Accessed 9 May 2024.

Gandhi, Rohith. “R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms.” *Medium*, 9 July 2018, https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

Nms — Torchvision Main Documentation. https://pytorch.org/vision/main/generated/torchvision.ops.nms.html. Accessed 9 May 2024.

Ren, Shaoqing, et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. arXiv:1506.01497, arXiv, 6 Jan. 2016. *arXiv.org*, https://doi.org/10.48550/arXiv.1506.01497.

Svhn · Datasets at Hugging Face. 19 Feb. 2023, https://huggingface.co/datasets/svhn.

The Street View House Numbers (SVHN) Dataset. http://ufldl.stanford.edu/housenumbers/. Accessed 6 May 2024.

TorchVision Object Detection Finetuning Tutorial — PyTorch Tutorials 2.3.0+cu121 Documentation. https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html. Accessed 6 May 2024.