Calvin Cramer

EEC 193 Lab 3

Jan 31, 2018

PHASE 1

Q1. How does your choice of loss affect the accuracy of your bounding box regressor. Specifically, what is the accuracy difference between L1 or L2 loss? Which works better and why? (To properly answer this question you will need to understand what is happening both conceptually as well have done some experimentation on your model).

The loss function does not affect the accuracy of the final model, but it does affect how quickly the model takes the train and converge. Here are the final train and test accuracies for each loss function:

Training accuracy L1: 89.7%

Testing accuracy L1: 78.1%

Training accuracy L2: 89.7%

Testing accuracy L2: 78.1%

I changed the accuracy threshold to 65%, but before with the original 60% the accuracies were very close to 100%, which didn't give much information. The L2 loss is preferable, since it's a devalues being farther away from the correct bounding box more heavily the farther away the prediction is, rather than L1 loss which devalues the error linearly. The result is that using L2 will converge the training faster than L1, but both loss functions will end up in the same trained state given enough time.

Q2. What is the purpose of IoU? Why is it so important in object detection?

The purpose of IoU is to measure how well bounding boxes are aligned. It is basically a measure of "how close are these two rectangles aligned". Without IoU, the obvious other metric to go by would be loss, but that does not tell any visual information, and also loss could be any number but that does not tell how well the bounding boxes are aligned. IoU is so important because it will tell us how well our model performs.

Q3. The current model in this phase can only do single object detection. How would you transform this model to handle multiple object detection? (There are multiple valid answers to this. You should compare the merits of each method.)

First of all, you need a dataset that has bounding box labels for all of the images.

Method 1: Add more output nodes to the final output layer, 4 nodes for each additional object you want to detect.

Method 2: Have a separate network trained for each different type of object. In phase 1, we are only tracking a single car, but we can run 1000 simultaneous models and track 1000 simultaneous objects, but only at most one of each type of object.

Method 3: If you only want to detect multiple cars then you can split the image into a grid, and run the single object detection network on each grid.

Method 1 would be the best of all three options, but it has a set number of objects it can detect. It has the advantage of the bounding boxes being able to overlap as well, whereas method 3 will have bounding boxes only within each grid cell. Method 3 has the huge disadvantage when the grid cells are too small, and an object spans multiple cells, that the bounding boxes, even if perfectly predicted, there will be multiple bounding boxes for each object, which is visually unappealing, and the network isn't actually finding a single box for a single object.

Method 2 is terrible in terms of efficiency, but method 1 and 3 should be about as fast as the original model in phase 1. In terms of changes to the dataset, method 1 and 2 would just require all bounding boxes for all important objects in the image, but method 3 will need the bounding boxes inside of every single grid cell, and even worse, there needs to be a different dataset for each grid size. Don't use method 3. Method 2 is too slow. Method 1 would work the best.

Method 4: Have a sliding method where there is a window that slides across the whole image, with a certain step size. At each step, the current model will predict the bounding box and object class of one object in that window.

The disadvantage of method 4 is speed, but it may work well predicting ever object in the image. The main limitation to method 4 is the choice of the window size, and step: if the window is too small, the model may fail to detect even a single object, or it may split a single object into multiple detections. Setting the window too large may hide the detection of an object in a window where there are multiple objects. This model is similar to method 3, but more general.

PHASE 2

I believe YOLOv3 is a better model than Mask-RCNN for the purpose of object detection for the self-driving car problem. My reasons are as follows:

- Inference time – YOLO runs at 45 fps to 150 fps for a fast version of the network, while Mask-RCNN runs around 5 fps.
- Training time – Mask-RCNN – 1 to 2 days on an 8 GPU machine, YOLO – 1 week on 1 GPU (assumed)
- False positive detection rate – "YOLO makes less than half the number of background errors compared to Fast R-CNN"

- Simplicity – YOLO is a single model that directly takes in the input image and outputs bounding boxes, classes, whereas Mask-RCNN has a mask branch, region proposals, selective search algorithm, and an image classification network.
- Weaknesses of YOLO – not good with small objects

While Mask-RCNN is generally a more accuracy model, and has the added benefit of providing an object mask, which represents the area of an object, Mask-RCNN is beaten by default for real-time object detection because of its inference speed. Mask-RCNN runs at 5 frames per second, which is too low for self-driving cars, and doesn't leave an room for other computation to be done on the system. Contrast the 5 fps with 150 fps for YOLO. YOLO is clearly the better option to do real time object detection just considering this metric.

The training time for Mask-RCNN was reported to be 1 to 2 days on an 8 GPU machine, whereas YOLO takes about a week on a (assumed to be) 1 GPU machine. While the YOLO paper did not explicitly say which machine they trained on, I am assuming it is the one they used for inference, which is a 1 GPU machine. Assuming this, YOLO will train in about a day on an 8 GPU machine, which is faster than Mask-RCNN.

The false positive rate is a very important metric for object detection networks. Think about how you would drive if every so often you hallucinated and thought there was a random object right in front of your car: you might start swerving to avoid it. For this reason, when the YOLO paper states that "YOLO makes less than half the number of background errors compared to Fast R-CNN", I am inclined to use YOLO, since false positives are so dangerous.

YOLO has the benefit of being simple and easily modifiable. Compared to Mask-RCNN, YOLO's architecture is simple to understand, and to see the reasoning behind each step. Mask-RCNN, comparatively, is built upon an image classifier, and uses that model to solve the harder problem of object detection. The multiple layers of Mask-RCNN leads to inefficiencies of data sharing between layers of the model, whereas YOLO treats the problem as a whole.

YOLO does state that because of its nature, choosing a set number of bounding boxes per grid, it is not adept at detecting small objects. Using YOLO for self-driving cars though, I argue that detecting small objects is that important, since cars, people, etc. are large. I would however use Mask-RCNN for a non-real time task, where accuracy and an object mask is needed.