Object Detection: FasterRCNN and SSD for Custom Object Detection

David Hughes
Cal Poly Pomona
3801 W Temple Ave, Pomona, CA 91768
dmhughes@cpp.edu

Abstract

Object Detection has gone through a renaissance in recent years with modern architectures like FasterRCNN and SSD. Models created from these methods have greatly increased accuracy and speed over past models with tradeoffs between popular state-of-the-art methods in terms of performance and precision. The goal of this project is both to learn about the details of FasterRCNN and SSD and to investigate the Object Detection API for running pre-trained and custom-trained models. Additionally, we are setting up a knowledge background for future work in automatic custom object detection that solves one of the challenges in machine learning: gathering a sufficient data-set. Finally, an implementation of code to run these object detectors will be used with toy applications of COCO data-set labels and custom playing card labels.

1. Introduction

Object Detection is the problem of detecting objects of a specific class from an image. This type of problem is a computer vision and image processing task. Furthermore, some examples applications of this problem are face detection and pedestrian detection in autonomous cars.

The problem that we faced is to learn the details of modern architectures that solve the Object Detection task in different ways. The deep learning methods that we reviewed in this project are FasterRCNN (2015) and SSD (2016).

Before getting to the architectures themselves, it is important to have some background on keywords and ideas in Object Detection. First is the proposal. A proposal is a rectangular region that might contain an object; other names include bounding box proposal, region of interest, region proposal, and box proposal. There are two different storage representations that are commonly used for proposals: 1. two corners (x0, y0, x1, y1) 2. center location (x, y, w, h). Second, it is important to have an evaluation metric. One such metric is Intersection over Union (IoU). This requires two sets of bounding boxes (for example: ground truth and predicted proposals). Afterwards, you calculate the division of the area of overlap over the area of union. This metric is the IoU evaluation metric that is commonly used in Object Detection as well as other areas of application like Information Retrieval.

$$IoU = \frac{OverlapArea}{UnionArea} \tag{1}$$

Third, remember that "recall" is the measurement of ratio of the correct objects in an image, and "precision" is the measurement of the ratio of correct predictions.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{2}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{3}$$

Using these two evaluation metrics together, we can calculate an even better metric. This is known as mean Average Precision, and it is commonly used in Object Detection to compare the success of certain models over a whole data-set. The Average Precision (AP) is the average of maximum precision at 11 recall levels. The mean Average Precision (mAP) is simply the average over all of the classes. When looking at the results of certain models, researchers and evaluators will use this common notation: mAP@0.5 ; this refers to the usage of mean Average Precision at 0.5 Intersection over Union.

$$AP = \frac{1}{11}\Sigma AP_r = \frac{1}{11}\Sigma p_{interp}(r) \ , \tag{4}$$
$$where\ p_{interp}(r) = max_{\tilde{r}>=r}p(\tilde{r})$$

Finally, there are some common techniques that help us refine results. First is Non Maximum Suppression (NMS). NMS is the process of merging overlapping proposals. If you have two proposals, you first compare them using IoU. If the measured IoU is greater than some IoU threshold then remove the proposal of lower confidence. This technique is

used frequently and at different levels in both of the architectures discussed in this paper. Second is Bounding Box Regression. The goal of this regression is to infer a bounding box that better fits the object inside. In order to do this, you train the regressor to look at the input region box and predict the offset to the ground truth box. This technique is often found at the end of the model and is applied before presenting the final box prediction to the user. Third and final is Negative Mining. Generally a large amount of proposals will have low IoU, which leads to imbalance in training data. This imbalance leans towards making negative predictions. The solution is to re-balance the training data to a 3:1 ratio (negative:positive). These techniques improve the quality of results provided by our models.

1.1. Review of Literature

The beginning of the short history (in terms of time) that will be reviewed started with AlexNet in 2012. The development of this Convolutional Neural Network (CNN) architecture that won ILSVRC 2012 combined the techniques of data augmentation, ReLU, dropout, and GPU usage. This started the modern era of computer vision and led directly to the development of current Object Detection models.

Following AlexNet, Ross Girshick combined the CNN feature extractor with the heuristic region proposal network [2]. This was known at the Region-Based Convolutional Neural Network (RCNN) and was the first step in modern Object Detection. The heuristic chosen was selective search, which computes hierarchical groupings based on similarity criteria (such as color, texture, brightness, size, shape, etc.).

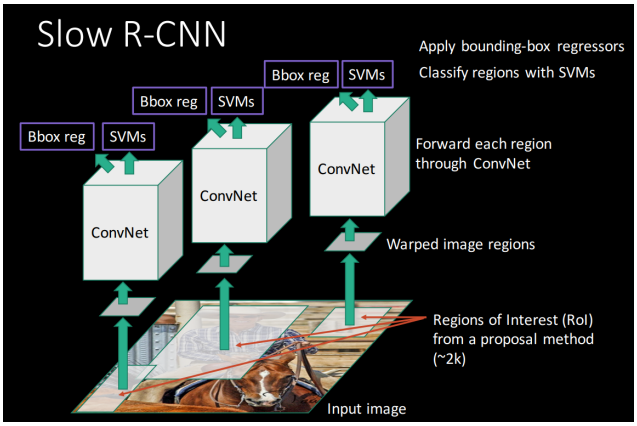

Figure 1. RCNN [6]

The problem with RCNN is that it uses a CNN to extract features for each of the 2000 regions generated by selective search. Also, it uses three models together which makes end-to-end training hard. As a result, it is too slow for a large data-set. The test time per image is 50 seconds with 66.0% mAP on VOC 2007 [3].

FastRCNN was also developed by Ross Girshick; it uses a single model to extract regions, predict classes, and return proposals [1]. The trick is to run the CNN only once per image.

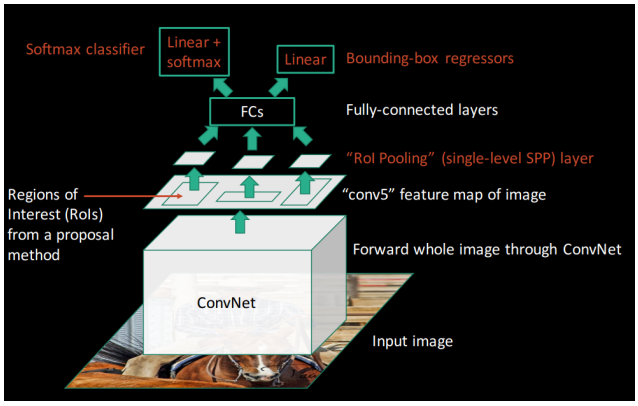

Figure 2. FastRCNN [6]

The major addition in this version is the Region of Interest (RoI) pooling. It is a differentiable layer that is an improvement over an idea developed in Spatial Pyramid Pooling Net (SPPnet). It allows the whole pipeline to be differentiable and trainable, providing the end-to-end pipeline that we are looking for. However, the major bottleneck is that this model version still uses the region proposal heuristic: selective search. Now the model has a test time of 2 seconds per image with a 66.9% mAP on VOC 2007, but selective search is still slowing everything down [3].

FasterRCNN is the evolution of FastRCNN. FasterRCNN introduces a Region Proposal Network (RPN) that shares features and computation time with the detection network in order to fix the bottleneck of generating proposals [6]. Thus the cost of generating proposals consumes similar time to the detection network itself. The authors of FasterRCNN noticed that the convolutional (conv) feature maps used by region-based detectors, like Fast R-CNN, can also be used for generating region proposals [6]. Therefore, they created a unified network with shared feature between the tasks that waives nearly all computational burdens of SS at test-time [6].

One problem with FasterRCNN is that it still requires many passes through one image to extract all of the objects. However, this model has a test time of 0.2 seconds with a 66.9% mAP on VOC 2007 [3].

The current debate in Object Detection is direct classification versus refined classification. Direct Classification is where you regress a prior box and classify the object directly from the same input region. It is faster but less accurate, since the features used to classify are not extracted exactly. Some models in this category are YOLO and SSD. Refined Classification regresses the prior box for a refined bounding box, then pools features of the refined box to form

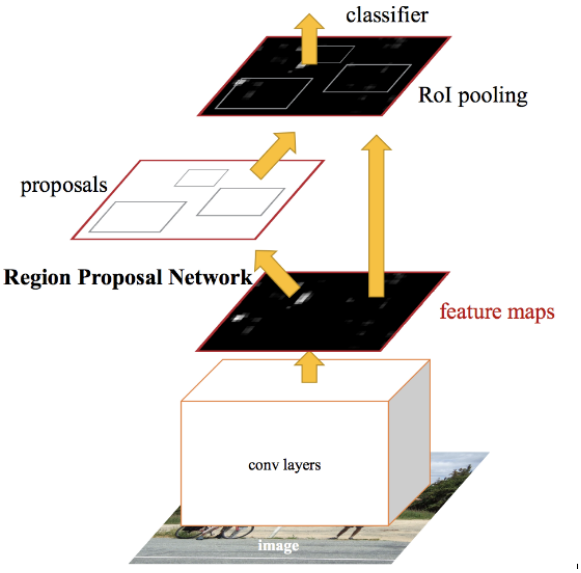

Figure 3. FasterRCNN [6]

a common feature volume. Afterwards, the model classifies the object by this feature volume. It is slower byt more accurate. Some models in this category are FasterRCNN and MaskRCNN.

Single Shot MultiBox Detector (SSD) is a method that breaks away from the standard approach in FasterRCNN of hypothesiz[ing] bounding boxes, resample[ing] pixels or features for each box, and apply[ing] a high-quality classifier [4]. SSD predicts category scores and box offsets for a fixed set of bounding boxes, which, along with various other improvements, allowed the method to produce high detection accuracy within a single-shot system.

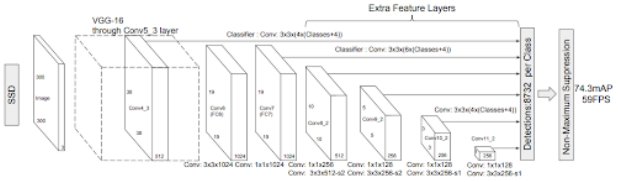

Figure 4. SSD [4]

SSD predicts category scores and box offsets for a fixed set of default bounding boxes using small convolution filters applied to feature maps. Running the multibox at different convolutional layer levels increases the chance of finding an object whether it is small or large. Additionally, the authors highly recommend some of the previously mentioned techniques of improving results such as data augmentation, NMS, and Negative Mining.

Based on the COCO mAP evaluation metric, the differences between the two algorithms become clear. For example, with the same COCO trained model and the same inception_v2 model: fasterRCNN has a speed of 58 ms and a COCO mAP of 28 and SSD has a speed of 42 ms and a COCO mAP of 24 [5]. It can be seen that with the modern solutions, there is a tradeoff between precision and time. FasterRCNN is slower but more accurate, while SSD is faster but less accurate. Therefore, the application of Object Detection must be considered when choosing a popular, pre-trained model.

2. Problem Description

Object detection is a task that can be easily performed by humans. We look out of a car and easily identify various signs, pedestrians, road condition, etc. However, as a task for a computer, it requires computer vision and image processing to discover the semantic meaning behind the rectangle of pixels that is a typical image. Object detection was once performed as a segmentation task, where images were partitioned by pixels in hopes of revealing the objects within the image. This method is insufficient for correctly identifying objects. The strategy shifted to finding probable segmentations; algorithms declare that within a certain bounding box, there might be an object. These probable segmentations are called object proposals or proposals. The way these proposals are generated are the source of the problem.

Our problem is to study FasterRCNN and SSD for Custom Object Detection. The first task is to learn the history and implementation of these modern models. Next, the second task is to apply these models over a custom dataset, which involves training the model according to Tensorflow's Object Detection API. Finally, there is future work in creating an end-to-end Custom Object Detection pipeline that can handle new classes provided by a user.

3. Methodology

The main methodology of this assignment is to make use of Tensorflow and Tensorflow's Object Detection API. The API allows a user to easily construct, train, and run various object detection models.

First, we used the code located in the object detection tutorial Jupyter Notebook in the API. This provides us with the necessary code for running a pre-trained model on the API. Next, we combined the code that installs the Object Detection API on Google Colab with the object detection tutorial code. The result is the assignment that we will provide to students, but also one method that will be used to gather results. Basically, with this code, we can import various pre-trained models from the Tensorflow Model Zoo in order to try different architectures with different CNN feature extractor back-ends.

Second, we combined code from a tutorial titled "TensorFlow Object Detection API Tutorial Train Multiple Ob-

jects Windows 10" by EdjeElectronics with Google Colab code that will set up with the API. This allows us to custom train a object detection model using our own data-set. The catch is that you have to either find a already labeled data-set or label your own images by hand using a software like LabelImg. This code is also available to students as a more challenging and time-consuming assignment.

Third, we set up a GPU computer with the necessary environment for Object Detection. This includes CUDA, CUDNN, Tensorflow-GPU, and the Object Detection API. It seems straight-forward at first, but version numbers and compatibility issues are major challenge. The GPU computer provides us with faster training than is available on the Google Colab environment. Also, using scripts provided by EdjeElectronics, we can easily use the desktop to run the Object Detectors on single images, videos, or webcams.

Finally, we ran the SSD and FasterRCNN models for object detection that were either pre-trained and custom-trained. While leaning of the code written by Tensorflow in their Object Detection API, we can set-up and run an object detector with these modern architectures.

4. Results

The results contained in this section are gathered from the Google Colaboratory Jupyter Notebook run-time environment; Google Colab offers access to a Linux back-end with a K80 GPU. The code is available at Github links in the Conclusion section, and it represents assignments that are available to other students. The code in the Github is the same code that is used to gather the results.

4.1. Pre-Trained Models

The first set of results was gathered using the same code that students' will run in the assignment. This allows us to easily run the options available in Tensorflow's Detection Model Zoo where they provide users with a collection of pre-trained detection models. We used the COCO-trained models with several test images that correspond to classes contained in the COCO protobuf "pbtxt" file.

The results match up with the theory that SSD is faster and less accurate, while FasterRCNN is slower and more accurate. SSD tends to be less confident about the proposals and sometimes miss a cut-off object, while FasterRCNN performs with better precision and recall at the cost of significant time.

4.2. Custom-Trained Models

The second set of results was gathered using custom model training. The steps to take include labeling the images by hand and running the training using the "model_train.py" file offered by Tensorflow's Object Detection API. The data-set for the test image in this paper is the

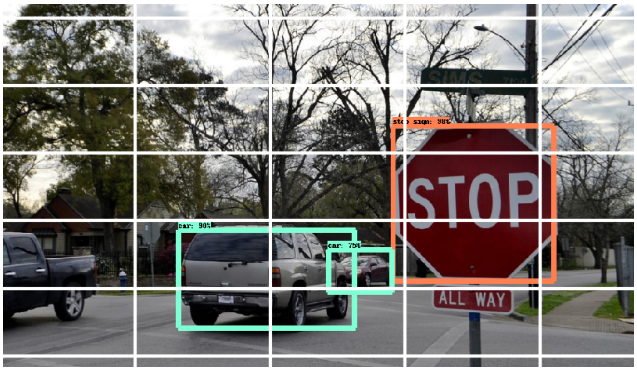

Figure 5. SSD Test Run on Image 2

playing card data-set provided by EdjeElectronics (link in conclusion). The training and validation loss were gathered using logs and visualized with Tensorboard. Note that the plotted loss is the combination of classification loss and regression loss.

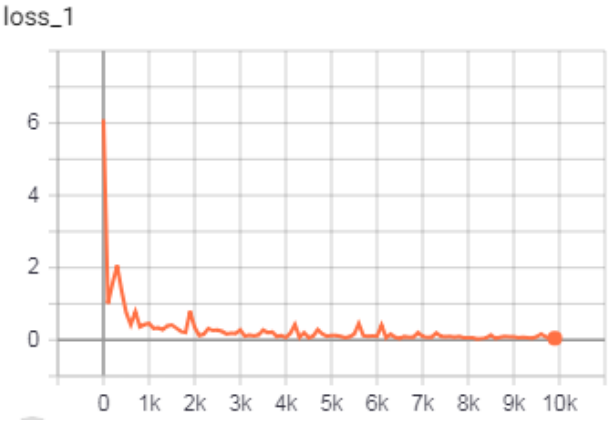

Figure 6. Training Loss of Custom Trained FasterRCNN

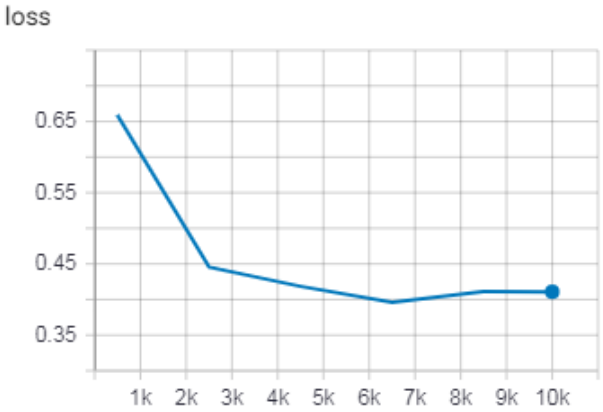

Figure 7. Validation Loss of Custom Trained FasterRCNN

The results in Table 1 show that we are able to train and run the custom object detector using the Tensorflow Object

Pretrained COCO Model Results			
Test #	Model	CNN Feature Extractor	Result (seconds)
Image1	SSD	MobileNetV1	3.3052144050598145
Image2	SSD	MobileNetV1	3.1751675605773926
Image1	FasterRCNN	Resnet101	9.989279747009277
Image2	FasterRCNN	Resnet101	10.854303359985352

Table 1. Results of Running Pretrained Models from the Detection Model Zoo

Detection API.

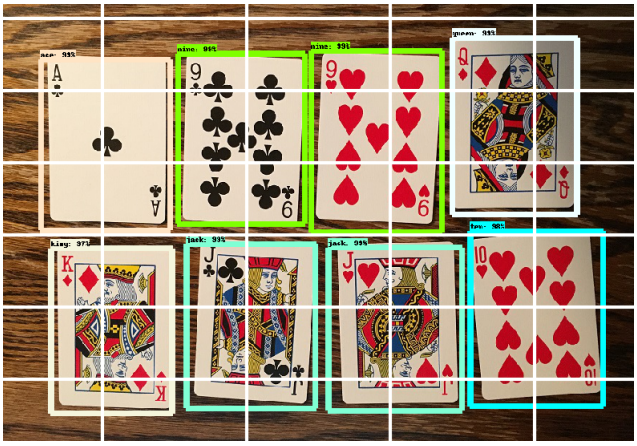

Figure 8. Custom Object Detector on FasterRCNN

The main challenge in this custom object detection is creating a sufficient data-set. It takes a lot of grunt work to label images by hand using a labeling software. This is a challenge that we wish to tackle in future work through automatic custom object detection.

5. Conclusions and Future Work

The main contributions of this project are mainly to create the desired deliverables: term paper, presentation, assignment, etc. In particular, we created multiple Jupyter Notebook for Google Colaboratory that allows you to easily run a pre-trained model for object detection (the assignment) and another notebook that allows you to create a custom object detector given that you label your own data-set (additional code). We also went through the set-up process of the Object Detection API manually on a GPU computer.

We learned a variety of lessons on this project. First, we learned that Deep Learning is a rapidly overturned field, where new models are replaced with a state-of-the-art model every year. Relative to the history of the world or even the history of computers, the history of modern object detection models has rapidly advanced in just seven years. Second, we learned how difficult it is to gather data for Machine Learning. It is something that everyone is aware of in this field but having to draw boxes and label images by hand really draws your attention to the issue. Third, we learned how convenient of a resource Tensorflow can be, but also how rigid the environment can be as well. The Tensorflow Object Detection API is a powerful tool that lets you easily run modern object detection architecture models. However, it is difficult to work in the environment as a beginner, where you have to be aware of unique file formats and random ways of printing the output you want (set a verbosity flag in a specific file).

For future work, like many other machine-learning and deep-learning tasks, the models are limited by the available training dataset. The Object Detection model can only identify objects that it was trained for; if a new object is added to the model then the model must be retrained with a sufficient set of images and associated image labels (rectangular bounding boxes in the XML file format). The process to create these labels is often done by hand, which makes it easy to see the tedious nature of this process. The goal is to create a pipeline of tasks that takes a desired object, generates data, and trains a model for Custom Object Detection.

One source of data is a Object Detection tutorial provided at this link: TensorFlow-Object-Detection-API-Tutorial-Train-Multiple-Objects-Windows-10 ; this provides the images for the playing card image-set. Pre-trained models are available at Tensorflow Detection Model Zoo. Code that runs the pre-trained model is available at my Github link here; this is also a simple assignment that helps to run the Object Detection API on Google Colab. Code that runs the custom trained model is available at my Github link here

References

[1] R. Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[3] F.-F. Li, A. Karpathy, and J. Johnson. Lecture 8: Spatial localization and detection, Feb 2016.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[5] pkulzc, V. Rathod, and N. Wu. Tensorflow detection model zoo, Dec 2018.

[6] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.