Object Detection: FasterRCNN and SSD for Custom Object Detection

David Hughes ~ dmhughes@cpp.edu
Dr. Hao Ji ~ hji@cpp.edu
Department of Computer Science, Cal Poly Pomona

Abstract

In this poster, we will present the topic of Object Detection along with FasterRCNN and SSD. 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 trade-offs 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.

Introduction

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 data-set, 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.

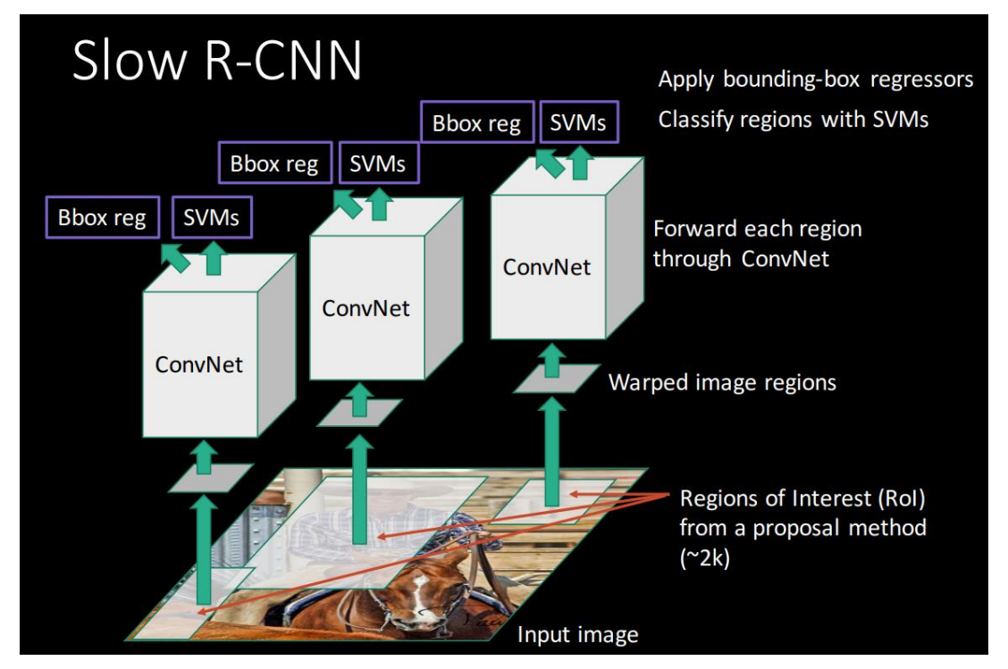
Slow R-CNN
Apply bounding-box regressors
Classify regions with SVMs
Bbox reg
SVMs
ConvNet
Forward each region through ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

Fig. 1: RCNN

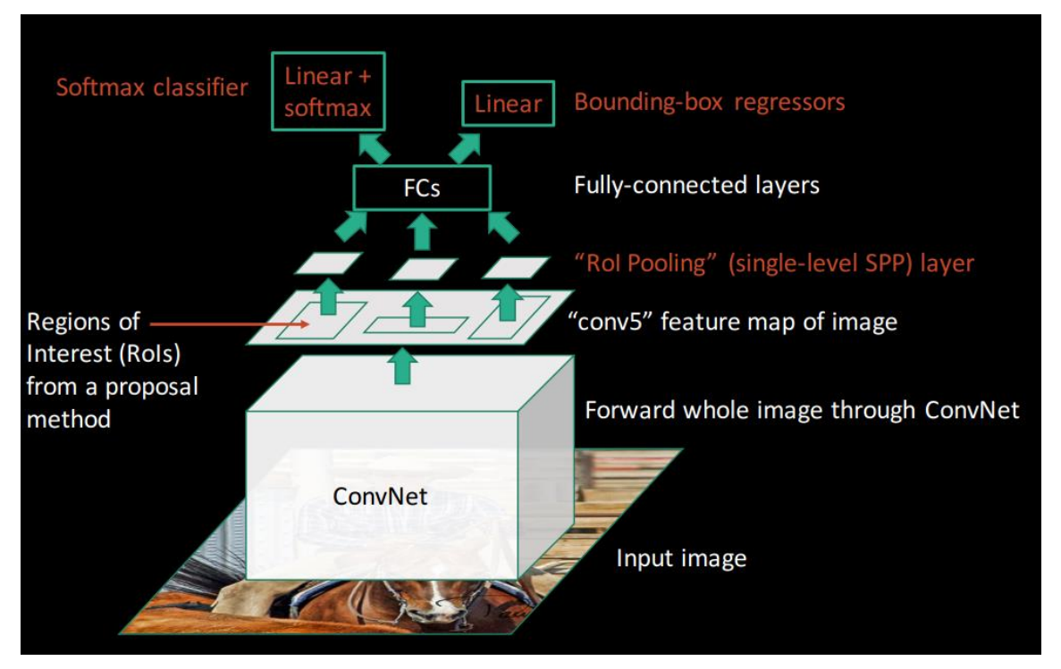
Softmax classifier
Linear + softmax
Linear
Bounding-box regressors
FCs
Fully-connected layers
"RoI Pooling" (single-level SPP) layer
Regions of Interest (RoIs) from a proposal method
"conv5" feature map of image
Forward whole image through ConvNet
ConvNet
Input image

Fig. 2: FastRCNN

Following AlexNet in 2012, Ross Girshick combined the CNN feature extractor with the heuristic region proposal network 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.). 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.

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

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. 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." Therefore, they created a unified network with shared feature between the tasks that "waives nearly all computational burdens of SS at test-time". 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.

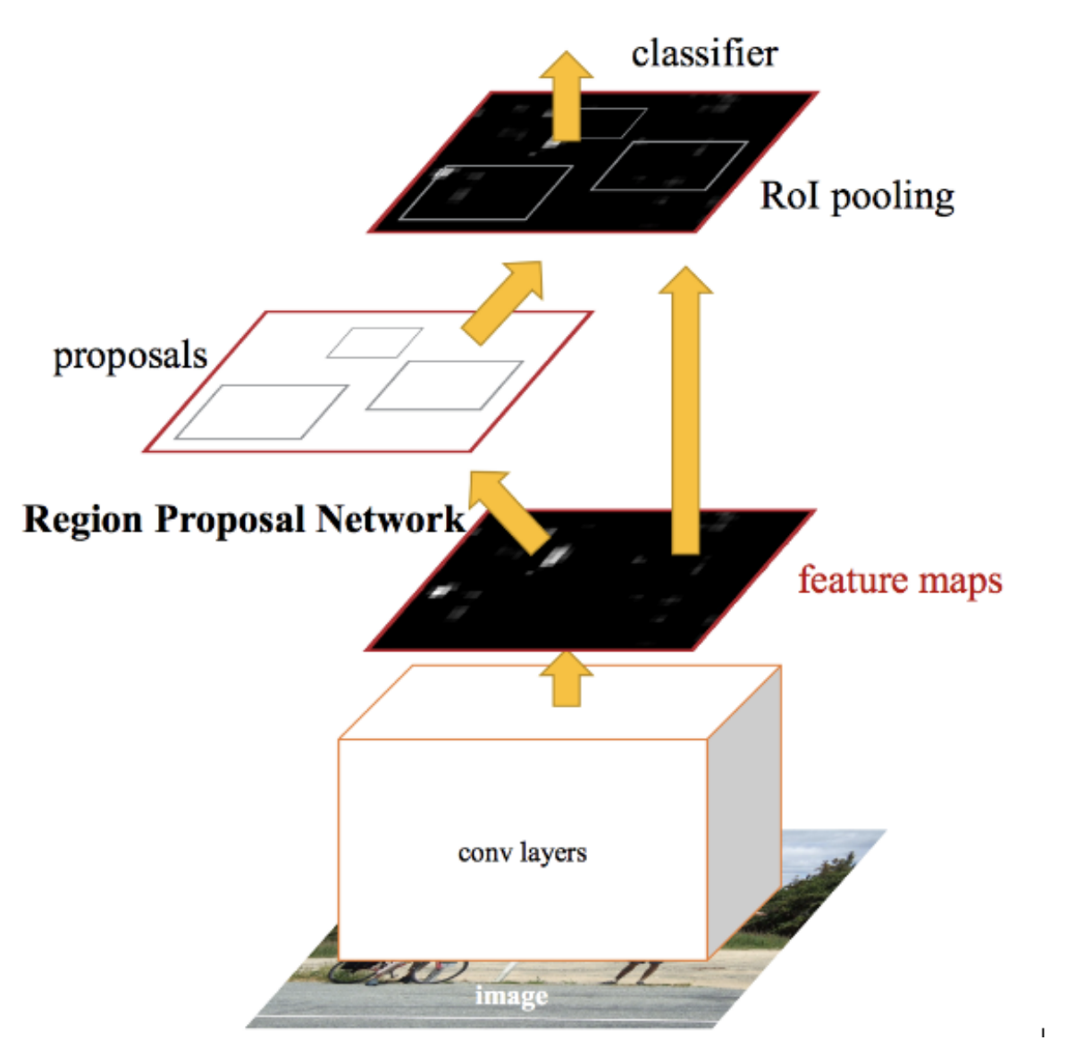

Fig. 3: FasterRCNN

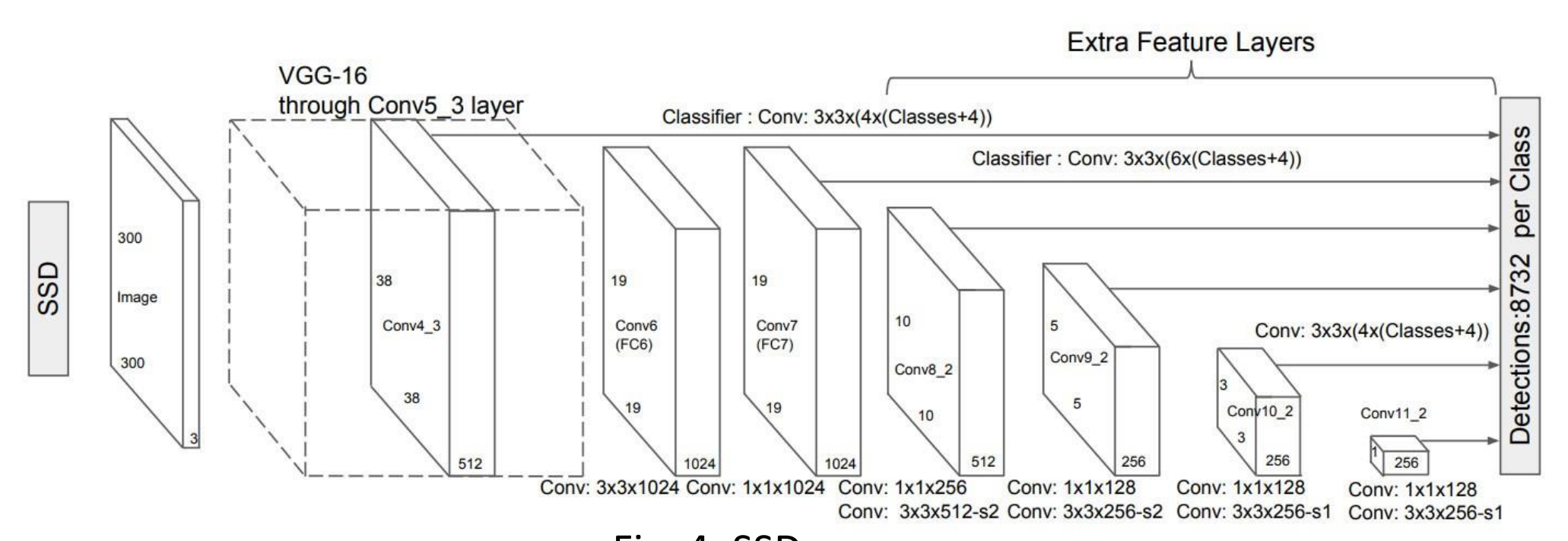

Fig. 4: SSD

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". 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. 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.

Method

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.. 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 Objects 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.

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. 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 (code here) and custom-trained (code here).

Data

The source of data for custom detector training was EdjeElectronics tutorial on the Detection API here. The process of creating a dataset for custom training consists of gather images, drawing boxes and labeling classes by hand using LabelImg (outputs XML files), and creating a label map. The test images for the pre-trained model were any images that contain COCO label map objects.

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 method section, and it represents assignments that are available to other students.

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

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. 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.

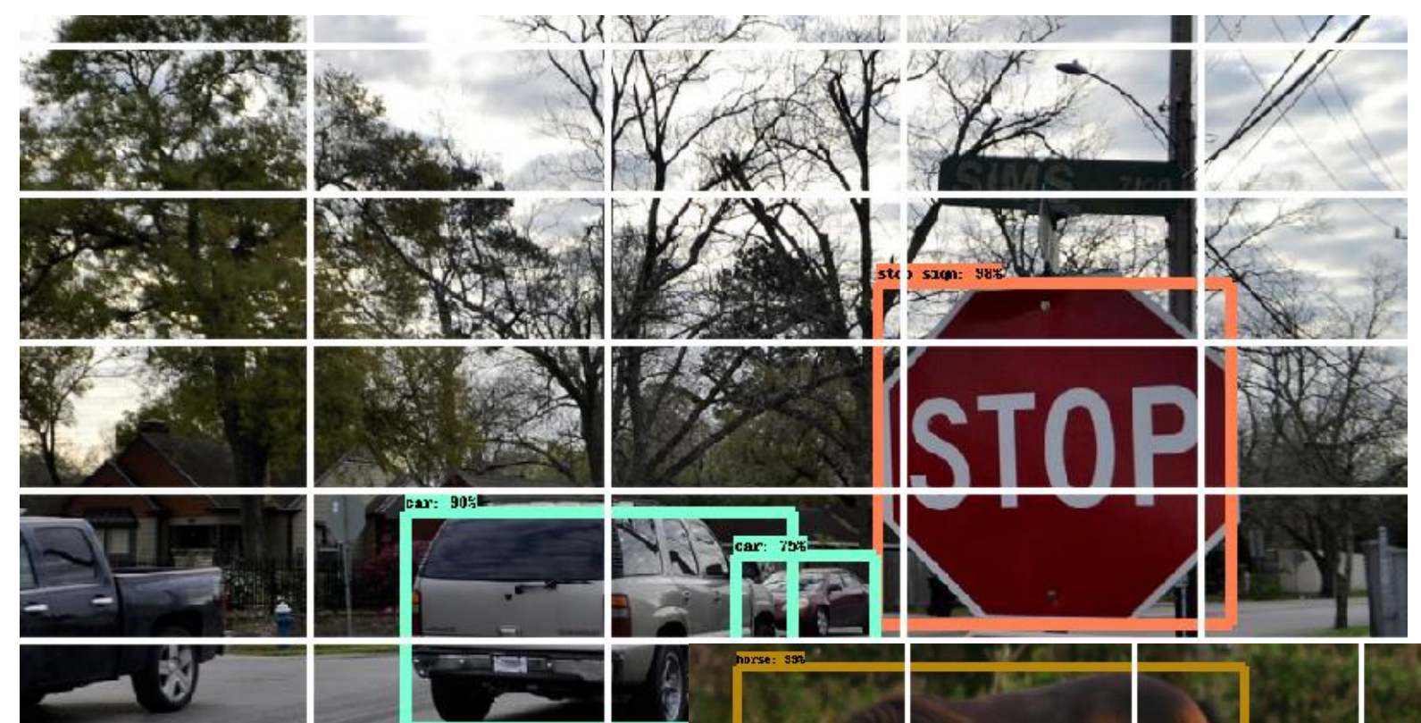

Fig. 5: Test Image 2 ran with SSD and MobileNetV1

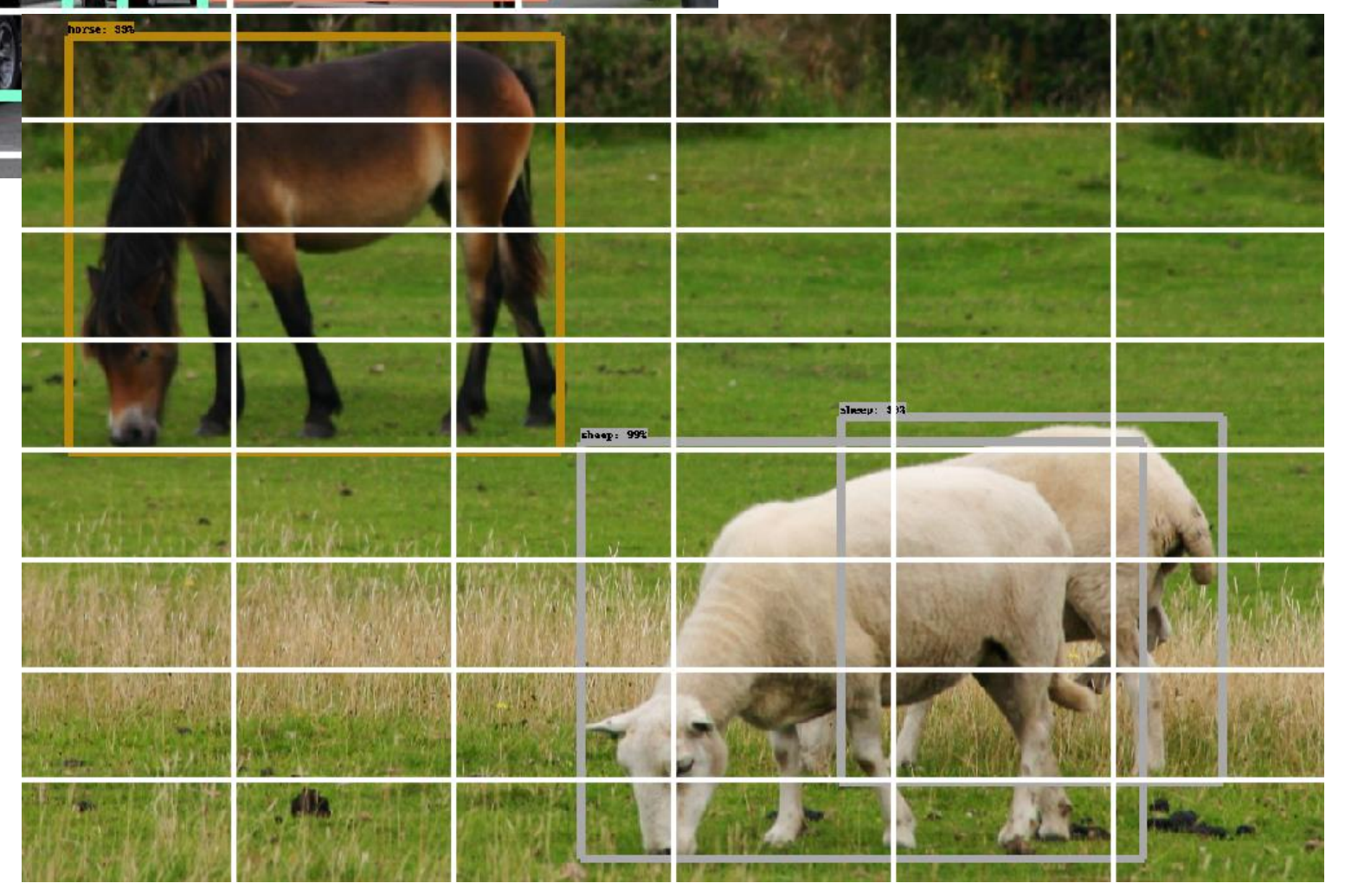

Fig. 6: Test Image 1 ran with FasterRCNN and resnet101

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 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.

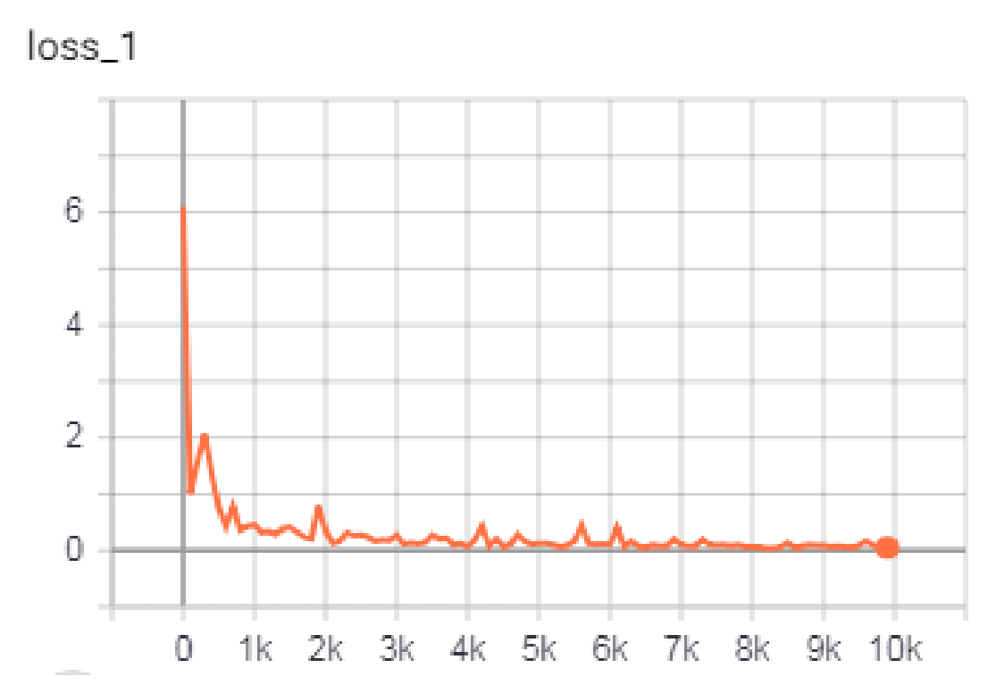

Fig. 7: Custom Training Loss FasterRCNN

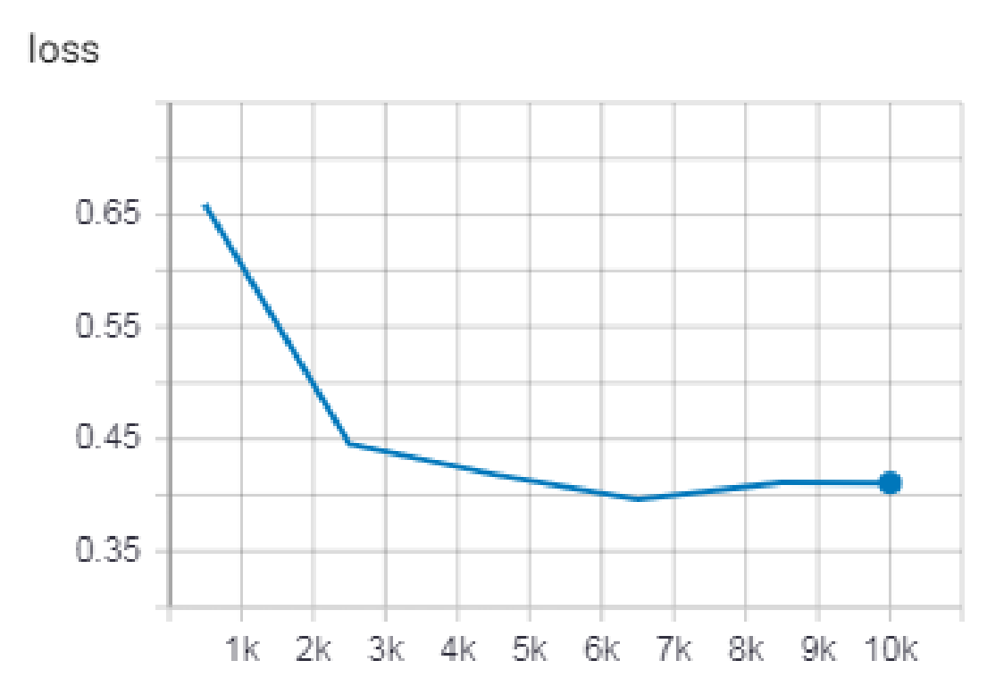

Fig. 8: Custom Evaluation Loss FasterRCNN

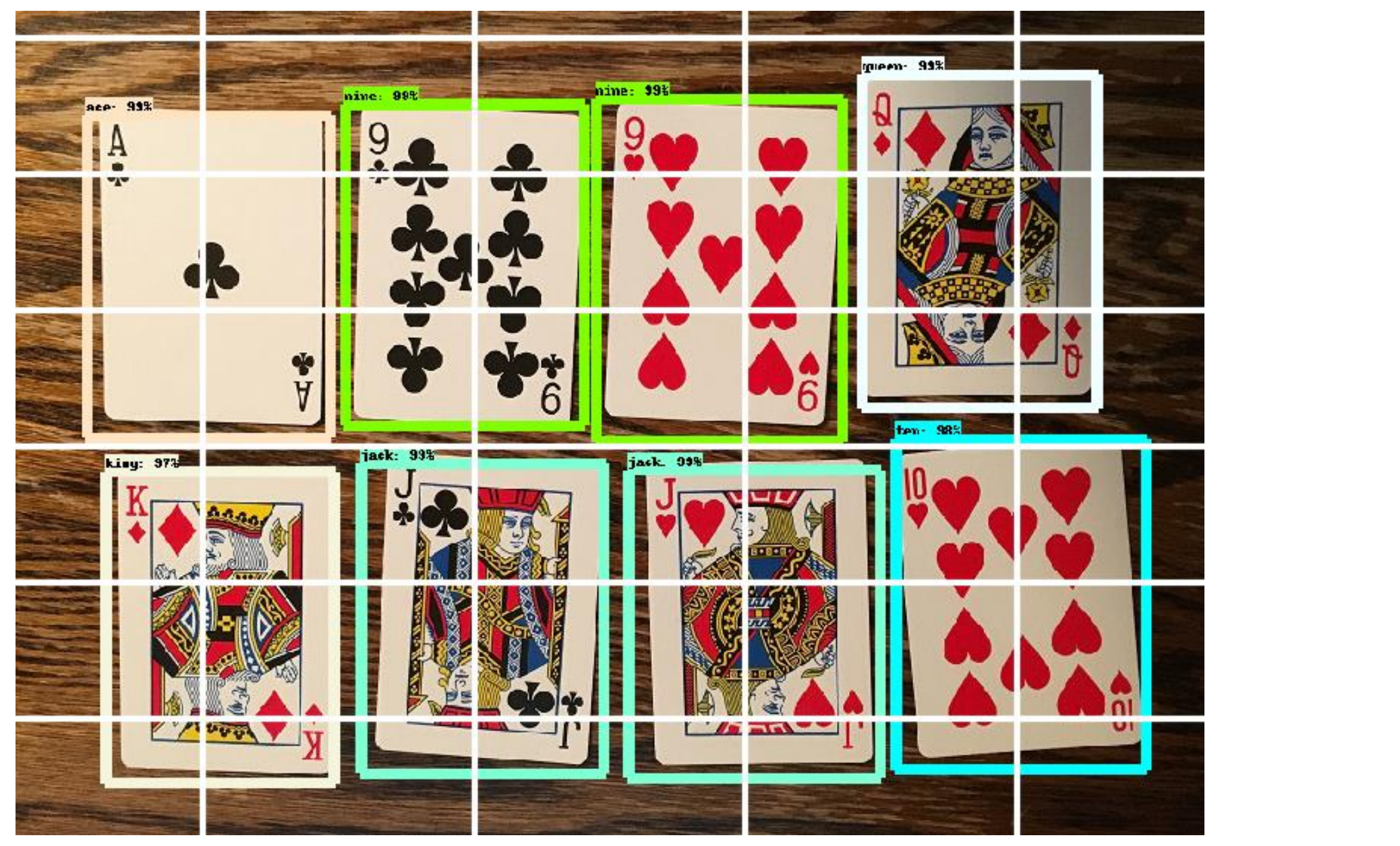

Fig. 9: Test Image with FasterRCNN for custom object detection

For the first set of results, the table shows that we are able to run the object detector using the Tensorflow Object Detection API with FasterRCNN and SSD. For the second results, the loss plots and the properly labeled test image demonstrate that were able to run the Object Detection API for training and custom object detection.

Conclusion

The main point of this work was to learn about FasterRCNN and SSD from a history, architecture, and implementation standpoint. This can be seen in the research that went into the introduction and the results section.

Furthermore, I learned a variety of lessons on this project. First, I 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, I 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.

References

[1] 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

[2] 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.

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

Acknowledgments

The main acknowledgments are to Tensorflow's Object Detection API for its Juptyer notebook tutorial code here and to EdjeElectronics for his custom object detection tutorial here