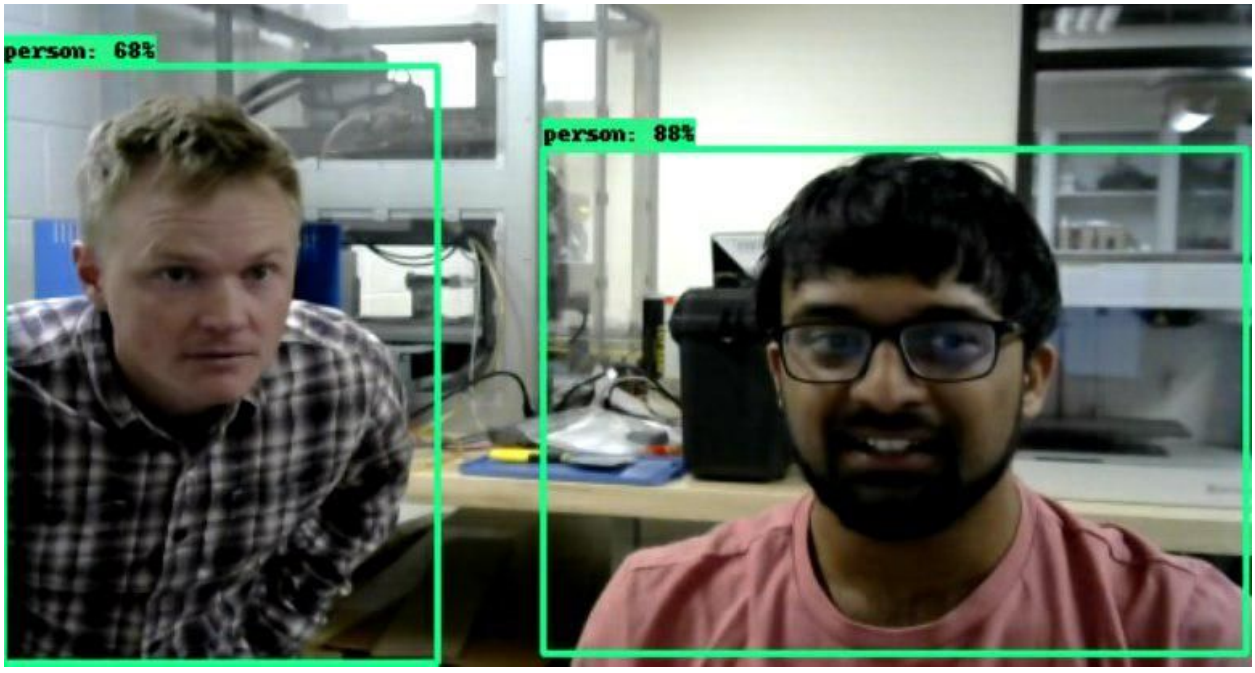
CSCI 5900 Independent Study

FINAL REPORT by Pranav Kumar Sivakumar

Multi-Object Detection and Tracking on Nvidia Jetson TX2 in Subterranea



Introduction

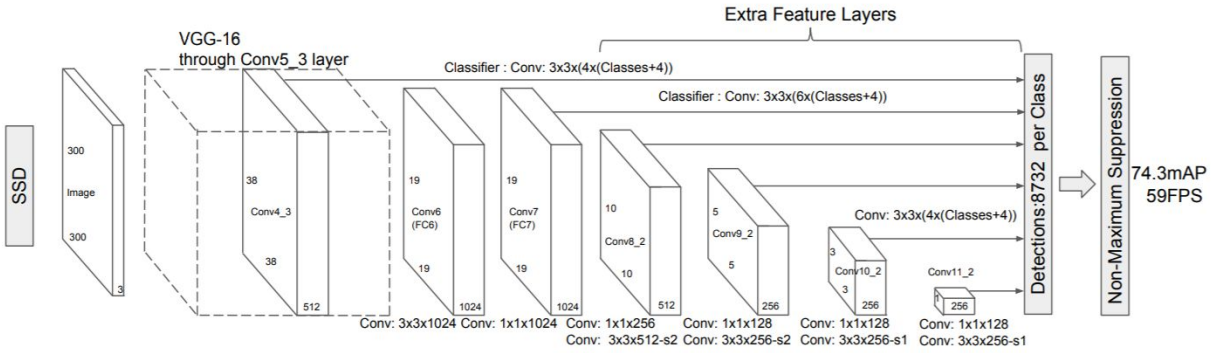
It's really not a good idea to compare the results of different object detectors directly from their papers to get a clear idea of which one is the best among them as there are many factors that affect them like different base feature extractors (e.g., VGG, MobileNet, ResNet, Inception), different default image resolutions, as well as different hardware and software platforms. In reality, we have to mainly decide on a proper trade-off between speed and accuracy based on our requirements. This project is mainly about developing a real-time object detection framework in an embedded AI computing device known as Nvidia Jetson TX2. The following details describe the parts of the model I chose and also explains the reason for choosing it and later talks about the experimental settings and results obtained.

SSD with MobileNet

Single Shot Detector[1] (SSD) is very popular for having a good number of frames per second (FPS) using lower resolution images at an admissible cost of accuracy. SSD is one of a few methods that pose detection as a regression problem. The other popular one is called YOLO(You Only Look Once). Before we compare them, let's get to know each of them.

For YOLO[11][9] framework, detection is a straightforward regression which takes an input image and learns the class possibilities along with the bounding box coordinates. YOLO divides every image into a grid of S x S and every grid predicts N bounding boxes and confidence values for each. The confidence reflects the precision of the bounding box and whether the bounding box in point of fact contains an object in spite of the defined class. YOLO also forecasts the classification score for every box for each class. You can merge both the classes to work out the chance of every class being in attendance in a predicted box. So, a total of SxSxN boxes is forecasted. On the other hand, most of these boxes have lower confidence scores and if we set a threshold (say, 30% confidence), we can get rid of most of them.

Whereas, SSD[1][9] runs a convolutional network on the input image only once and calculates a feature map. Later, we run a small 3×3 sized convolutional kernel on this feature map to predict the bounding boxes and classification probability. SSD also uses anchor boxes at various aspect ratio and learns the off-set rather than learning the box. In order to handle the scale, SSD predicts bounding boxes after multiple convolutional layers. Since each convolutional layer operates at a different scale, it is able to detect objects of various scales.



The above figure is an example of an SSD with VGG-16 as the feature extracter. The SSD model adds several feature layers to the end of a base network, which predicts the offsets to default boxes of different scales and aspect ratios and their associated confidences.

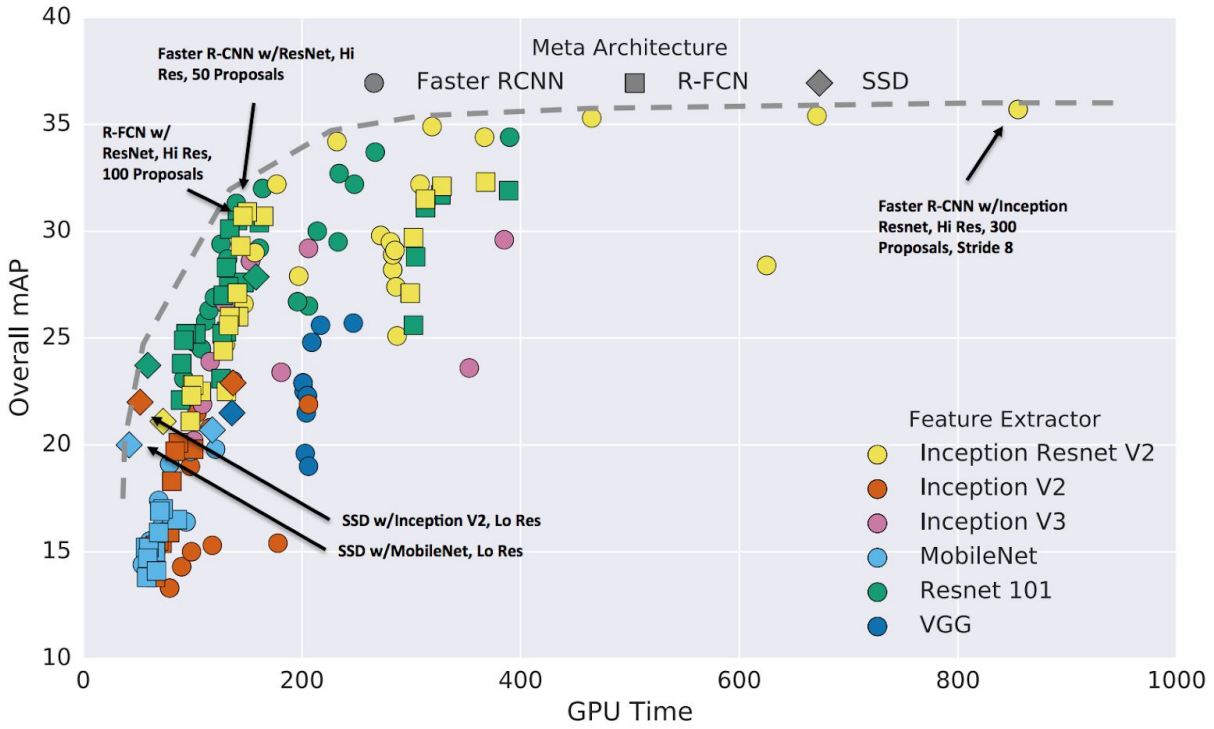
The main difference between these two models is that YOLO makes predictions for only a single feature map while SSD combines predictions across multiple feature maps at different sizes. Also, SSD is a better option than YOLO as we are able to run it on a video and the accuracy trade-off is very modest. While dealing with large sizes, SSD seems to perform well, but when we look at the accuracies when the object size is small, the performance dips a bit. Even though YOLO is faster it loses on in its scores comparatively.

Region-based detectors like Faster R-CNN[12] (that uses RPNs to create boundary boxes and classify objects) demonstrate small accuracy advantage if real-time speed is not needed (it runs only at 7 FPS). But in our case, the real-time constraint has to be satisfied, so let's adhere to SSDs. Another important factor to consider is that the model should run on Nvidia Jetson TX2 which relatively has a lower configuration compared to general-use PCs. So it's fair to use a light-weight model with real-time performance with reasonably reduced accuracy from the state-of-the-art. (Here, the Accuracy is measured as the mean average precision mAP: the precision of the predictions.)

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	$\sim 1000 \times 600$
Fast YOLO	52.7	155	1	98	448×448
YOLO (VGG16)	66.4	21	1	98	448×448
SSD300	74.3	46	1	8732	300×300
SSD512	76.8	19	1	24564	512×512
SSD300	74.3	59	8	8732	300×300
SSD512	76.8	22	8	24564	512×512

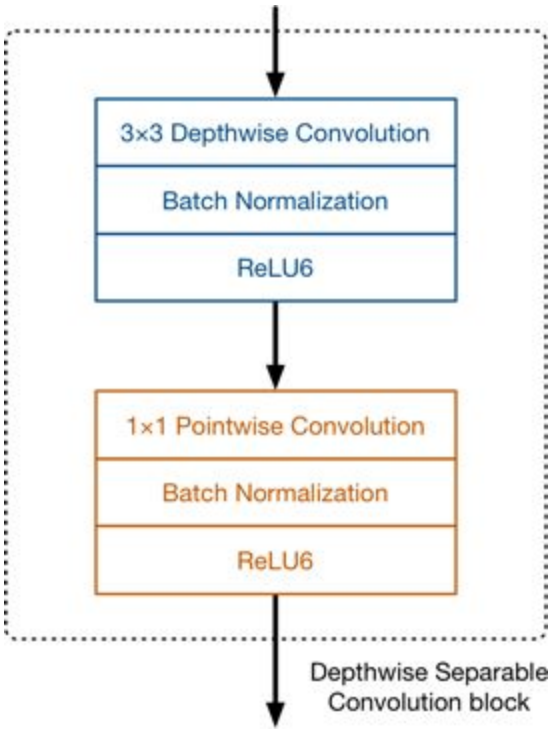
The above table displays the results on Pascal VOC2007 test that compares SSD, Faster R-CNN, and YOLO. SSD300 is the only real-time detection method that can achieve above 70% mAP. By using a larger input image, SSD512 outperforms all methods on accuracy while maintaining a close to real-time speed. These results were computed on Nvidia Titan X which has approximately 10 times higher GFlops-FP32 than Nvidia Jetson TX2. Therefore,

using a faster base network could even further improve the speed, which can possibly make this SSD300 model real-time as well in Nvidia Jetson TX2.



From the above graph[2], SSD w/ MobileNet (Lo-Res) is the fastest having a good trade-off with the Overall mAP. Here MobileNet[3] is a feature extractor/base network which has been shown to achieve VGG-16 level accuracy on Imagenet with only 1/30 of the computational cost and model size. MobileNet consists of an efficient architecture introduced by Google mainly designed for efficient inference in many portable vision applications found in mobile phones.

The main idea behind MobileNet[3][10] is that convolutional layers can be replaced by so-called depthwise separable convolutions (refer below figure) which are faster than them. Its job is split into two subtasks: first, there is a depthwise convolution layer that filters the input, followed by a 1×1 (or pointwise) convolution layer that combines these filtered values to create new features. The following figure is an illustration of a depthwise separable convolution block. It essentially helps in building lightweight deep neural networks that do approximately the same thing as convolutions.



The architecture of MobileNet consists of a regular 3×3 convolution as the very first layer, followed by 13 times the above building block with no pooling layers between them. They are followed by batch normalization and the activation function used in this architecture is ReLU6. Because of these changes, MobileNet has to do about 9 times less work than comparable neural nets with the same accuracy.

Model summary	minival mAP	test-dev mAP
(Fastest) SSD w/MobileNet (Low Resolution)	19.3	18.8
(Fastest) SSD w/Inception V2 (Low Resolution)	22	21.6
(Sweet Spot) Faster R-CNN w/Resnet 101, 100 Proposals	32	31.9
(Sweet Spot) R-FCN w/Resnet 101, 300 Proposals	30.4	30.3
(Most Accurate) Faster R-CNN w/Inception Resnet V2, 300 Proposals	35.7	35.6

Test-dev performance of the "critical" points along our optimality frontier.

From the above table [2], we see that SSD models with Inception v2 and Mobilenet feature extractors are most accurate of the fastest models. we can identify that Faster R-CNN[12] is the most accurate but it is very slow compared to SSD. Note that if we ignore postprocessing costs, Mobilenet is roughly twice as fast as Inception v2 while being slightly worse in accuracy.

Based on stats in Tensorflow model zoo, Mobilenet v2 is faster on mobile devices than Mobilenet v1 but is slightly slower on desktop GPU so we are keeping v1 as our model.

Other lessons learned are, SSD performs worse for small objects and takes less significant advantage of better feature extractors compared to Faster R-CNN and R-FCN.

Framework Resolution	Model	mAP	Billion Mult-Adds	Million Parameters
SSD 300	deeplab-VGG	21.1%	34.9	33.1
	Inception V2	22.0%	3.8	13.7
	MobileNet	19.3%	1.2	6.8
Faster-RCNN 300	VGG	22.9%	64.3	138.5
	Inception V2	15.4%	118.2	13.3
	MobileNet	16.4%	25.2	6.1

For both SSD and Faster RCNN[12], MobileNet achieves comparable results to other networks with only a fraction of computational complexity and model size (refer above table).

Experiments and Results (Repository)

- The Nvidia Jetson TX2 was flashed with Ubuntu 16.04 and it's board packages along with several modules like OpenCV, TensorFlow, CUDA toolkit, cuDNN, and many other Computer Vision based dependencies.
- The model that I've been using was the frozen graph of the SSD with MobileNet v1 trained on COCO dataset from TensorFlow.
- This project is mainly based on Tensorflow's Object Detection API[4]
- The following work has been done in this project (the Jupyter notebooks in the repository are very readable with many comments and references)
    - Receive the video frames using OpenCV and add them to an input queue. Later threads running the detection mechanism work on them concurrently and add bounding boxes for the detected objects in frames and add them to an output queue. This queue is used by a visualization module for our viewing it. (For later version I removed the outer queues)

- Later added KCF tracking[5] alongside the detections (for example, 10 frames can be tracked between every detection).
- Have added two types of compilations of KCF tracker that have different dependencies, so it can be used based on our convenience in different platforms.
- Non-maxima suppression is the main bottleneck for this model. Likewise, Tensorflow object detection API provides a method to export the model with a reduced threshold for non-maxima suppression. Used it to do the same and replaced the original model.
- Added split_model speed hack which splits the process into GPU and CPU sessions[6] working in a single thread that does visualization towards the end with the frames that come from either of the CPU or GPU worker. The code additions work only for this model but result in a great performance increase. Also added multithreading for supporting split sessions.[7][8]
- Added options for the model to grow GPU memory allocation in tensorflow to support the model.
- Conditionals are added that manages the options for different types of Visualizations.
- Separated the functionalities for fps computations, session handlings and OpenCV related methods as external helper functions/classes.
- Made it minimal enough by removing unnecessary dependencies from Tensorflow object detection utility helpers.

- Current rate: 30 to 32 FPS (without visualizations improves the speed).

References

1. SSD: Single Shot MultiBox Detector
2. Speed/accuracy trade-offs for modern convolutional object detectors
3. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications
4. https://github.com/tensorflow/models/tree/master/research/object_detection
5. High-Speed Tracking with Kernelized Correlation Filters
6. https://github.com/tensorflow/models/issues/3270

7. https://github.com/naisy/realtime_object_detection/blob/master/About_Split-Model.md
8. https://github.com/gustavz/realtime_object_detection/tree/master
9. https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/
10. https://machinethink.net/blog/mobilenet-v2/
11. You Only Look Once: Unified, Real-Time Object Detection
12. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks