CNNs for real time object detection on mobiles

Rajat Chand

University of Washington

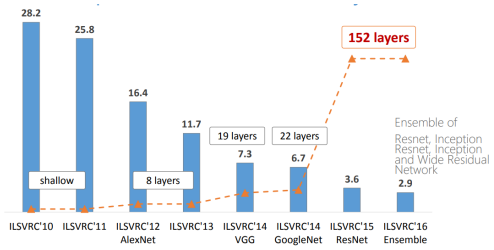
rajatc@uw.edu

Abstract

Object detection is a major challenge in computer vision, involving both object classification and object localization within a scene. The present architecture of many object detection algorithms are still far too heavy for mobiles. In this project, we try to access the performance of light weight neural network architectures for real time object detection on mobiles. We train one such architecture on an auonomous driving based 2D object detection dataset and examine if an object detection pipeline running entirely on smarphones be used for autonomous navigation.

1 Introduction

LeNet-5 [7] was the first convolutional neural network developed in 1998. It worked on 32×32 pixel grayscale input images to classify digits. However lack of computing power and large training dataset hindered the ability of these networks to process higher resolution images and solve complex problems. A renewed interest in CNNs was brought about by the AlexNet [6] that famously won the 2012 ImageNet LSVRC-2012 competition, pushing the classification error from 26.2 % in 2011 down to 15.3 %. And today, CNNs are everywhere. They are arguably the most popular deep learning architecture around and mostly responsible for recent surges in deep learning. Over the years CNNs and in general neural networks have become deeper and more complex. Figure 1(a) shows how we went from 8 layers to 153 layers in just 3 years. We have already surpassed human level recognition and the last year's ImageNet winning team had a top-5 error of 2.25%.



(a) ImageNet Classification Top-5 error (%)

(b) Le 2 Smartphone

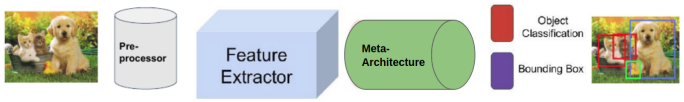
Figure 1

CNNs have been the go to model for solving problems in Computer Vision. One such classical problem is the simultaneous recognition and localization of objects in an image (or video), also termed as Object detection. Object detection has achieved significant progress in the last few years, using deep convolutional networks. [4]. While accurate, these approaches have been too

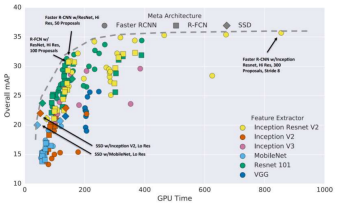
computationally intensive for embedded systems and, even with high-end hardware, too slow for real-time applications. In many real world applications such as robotics, self-driving car and augmented reality, the recognition tasks need to be carried out in a timely fashion on a computationally limited platform. This has led to recent interest in developing small and efficient neural networks such as Squeezenets, Mobilenets and Xnornet [3, 5, 9], just to name a few. Here, we try to fine-tune a light, pre-trained object detection architecture on the KITTI [2] dataset and extract conclusions on its feasibility for autonomous driving or other driving assistance systems. This architecture is based on the Mobilenet feature head and the SSD detector. [3, 8]. After finetuning, the performance of the detector on the validation set is evaluated according to the VOC evaluation metrics [1]. We then deploy it on an Android device and asses the trade-offs between real time detection accuracy and speed while driving across university district. We will use LeEco's Le 2 smartphone as a platform to run the inferences using the trained model. It has a Qualcomm Snapdragon 652 chipset with 8 CPUs, four Cortex-A72 clocked at 1.8 Ghz each and another four Cortex-A53 clocked at 1.4 GHz each. It also consists of an Adreno 510 GPU and has a 3 GB RAM.

2 Object Detection Pipeline

As shown in Figure 2(a) all object detection algorithms consist of mainly three moving parts, the pre-processor, the feature extractor and the the proposal generator (or meta architecture). The pre-processor takes in the raw images and performs image resizing, image pixel normalization and data augmentation operations, such as flips and random rotations, on it and makes it suitable to be fed into the feature extractor. The pre-processed image is then passed through the feature extractor, which is an off the shelf image classification network (VGG, Inception, Resnet, Inception Resnet, Googlenet or Mobilenet) with its final few or all fully connected layers removed. It output a feature map, a mapping from high-dimensional, low level pixel space to low-dimensional high level feature space. This feature map is further passed through the detector Meta-architecture (or the meta architecture) to get bounding boxes and class probability for the image. The three main meta-architectures are the Single Shot Detection, R-FCN and the R-CNN. The meta architecture of R-FCN and Faster R-CNN consist of two different networks - a proposal generator to output bounding boxes and objectiveness and a box classifier to output class probability. Single Shot Detector modifies the proposal generator to directly output class probability instead of objectivetiness and is thus generally faster. YOLO and SSD Multibox detector(also known as SSD) are examples of single shot detection [10, 8].



(a) Accuracy vs GPU Time for different extactor and meta-architecture combinations [4]



(b) Accuracy vs GPU time

Figure 2

A study at Google Research by Hunag et al[4] tried different combinations of feature extractor and meta-architectures and plotted the overall mAP value with GPU time, as can be seen in Figure 2(b). In general, Faster R-CNN is more accurate while R-FCN and SSD are faster. Faster R-CNN using Inception Resnet with 300 proposals gives the highest accuracy at 1 FPS. SSD on MobileNet has the highest mAP within the fastest models. This graph also helps us to locate some sweet spots with a good return in speed and cost tradeoff. R-FCN models using Residual Network strikes a good balance between accuracy and speed while Faster R-CNN with Resnet can attain similar performance if we restrict the number of proposals to 50.

3 How Good is your Detection algorithm ?

There are different metrics that exist for evaluating your algorithm. The three important ones are

- **Mean Average Precision** This involves thresholding based on intersection of union score and then averaging over all class predictions. Higher the mAP the more accurate will be your detections.

- **Inference time** This is the total time required to pre-process the image, pass it through the detector network and get a prediciton.

- **Memory usage** This refers to the model's cpu/gpu usage during prediction. Depending on the task and the resources available one should try to achieve a sweet-spot between the three.

Let us understand mAP in more detail as it would be used in further analysis.In order to calculate Mean Average Precision (mAP) in the context of Object Detection one must compute the Average Precision (AP) for each class, and then compute the mean across all classes. The key here is to compute the AP for each class, in general for computing Precision (P) and Recall (R) one must define what are: True Positives (TP), False Positives (FP), True Negative (TN) and False Negative (FN). In the setting of Object Detection of the Pascal VOC Challenge, these are the following:

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN} \tag{1}$$

- **TP:** Those Bounding Boxes (BB) for which the intersection of union with ground truth boxes (GT) is more than 0.5

- **FP:** BB for which the IoU with GT is below 0.5 and also the BB that has an IoU with a GT that has already been detected.

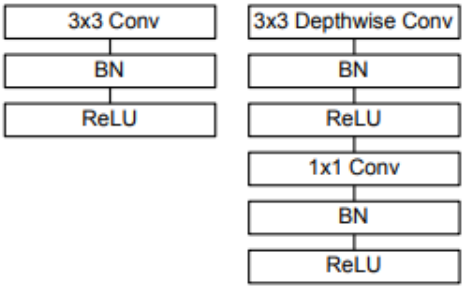
- **TN:** there are no true negative, the image is expected to contain at least one object

- **FN:** Those images where the method failed to produce a BB

Now once we have the precision recall values, we calculate AP (average precision) as average of 11 values of Precision at the points where Recall = $\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$, i.e. $AP = \frac{1}{11} \sum_{R \in \{0, 0.1, ..., 1\}} P(R)$. Once we have the Average precision for each class, the mean precision is the weighted average of Average precision over all the classes.

4 The Mobilenet-SSD Architecture

The early network layers of the Mobilenet-SSD architecture is based on the mobilenet architecture [3] which is a lightweight convolutional neural network which uses depthwise seperable convoltions to reduce the number of parameters and increase speed. Depthwise seperable convolution feactorize a standard convolution into a depthwise convoltuion and a 1×1 convolution called a pointwise convolution greatly reducing computation complexity. Figure 4 shows the SSD based detector with VGG-16 base network, a Mobilenet-SSD is similar to this architecture where the VGG-16 base network is replaced by Mobilenet. The SSD meta-arachitecture adds convolutional feature layers to the end of the truncated base network. These layers progressively decrease in size allowing for predictions at multiple scale. We apply on top of each convolutional feature map (output of SSD's convolutional feature layer) a set of filters that predict detections for different aspect ratios and class categories



(a) Depthwise Seperable Convolution

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

(b) Mobilenet Architecture

Figure 3

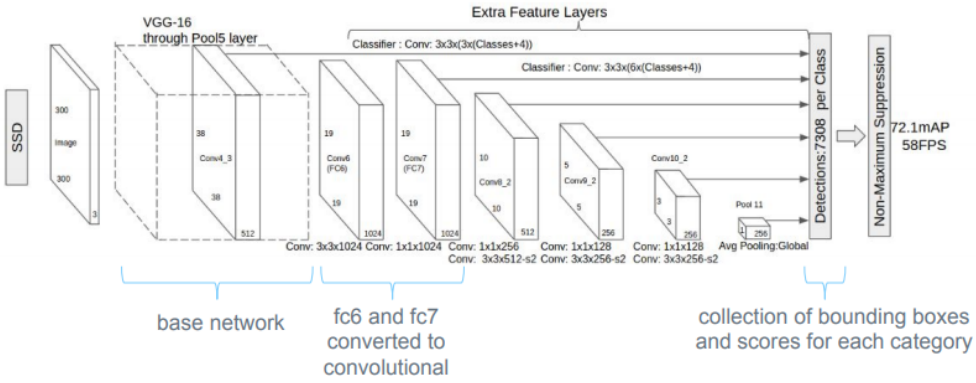


Figure 4: Single Shot Multibox detector (SSD)

5 KITTI Dataset

KITTI Vision Benchmark Suite datasets [2] are captured from driving sessions around the city of Karlsruhe (Germany), in rural areas and on highways. It consist of 7481 training images, each with a resolution of 1240×375 pixels. Since the testing images in this dataset aren't labelled 80% of the training images are used as training data and the rest 20% is used as validation data. The dataset consist of 11 annotated classes out of which the main classes present are: cars, with 28.742 samples, followed by pedestrians, with 4.487 samples and cyclists, with 1.627 samples. We use 5 classes for finetuning the Mobilenet-SSD model - car, pedestrian, cyclist, truck and van.

Figure 5: Images from the KITTI dataset

6 Transfer Learning

We finetune the MobileNet SSD Model pre-trained for the coco dataset (called `ssd_mobilenet_v1_coco` and downloaded from Tensorflow detection model zoo). Horizontal flipping which involves mirroring the image and random cropping operations are applied to the input training images to augment the dataset. The entire network is trained using a batch-size of 32 images and a momentum based RMSprop optimizer. The learning rate of the optimizer is initially set to 0.004 and allowed to exponentially decay with a decay factor of 0.95 and with 800720 as the decay steps. The other hyper-parameter values used are, momentum optimizer value=0.9, decay=0.9 and epsilon=1.0 .

6.1 Loss function

The overall loss function as proposed in [8] is calculated as the weighted sum of localization loss and classification loss (also known as confidence loss). Localization loss is a measure of how close are the bounding boxes the model created to the ground truth boxes. The classification(or confidence loss) is a measure of how accurately does the model predict the class of each object. Here, we use use a weighted sigmoid classification loss and a weighted smooth L1 localization loss. These two losses are equally weighted when calculating the final total loss. Below we have summarized the nomencalure for defining losses

$\boldsymbol{L}$: Total loss, $\boldsymbol{L_{conf}}$: Confidence (or Classification) loss, $\boldsymbol{L_{loc}}$: Localization loss, $\boldsymbol{l}$: predicted box, $\boldsymbol{g}$: ground truth box, $\boldsymbol{d}$: default box, $\boldsymbol{N}$: number of default boxes, **(cx, cy)**:center of the default bounding box, $\boldsymbol{w}$: width of the bounding box, $\boldsymbol{h}$: height of the bounding box.

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + L_{loc}(x, l, g)) \tag{2}$$

6.1.1 Localization loss

The localization loss is calculated only on positive boxes (ones with a matched ground truth). It calculates the difference between the correct and predicted offsets to center point coordinates, and the correct and predicted scales to the widths and heights. And smooth the absolute differences.

$$L_{loc}(x, l, g) = \sum_{i \in Pos} \sum_{m \in \{cx, cy, w, h\}} x_{i,j}^{k} \text{smooth}_{L_1}(l_i^m - \hat{g}_j^m) \tag{3}$$

$$\hat{g}_j^{cx} = \frac{(g_j^{cx} - d_i^{cx})}{d_i^w}, \hat{g}_j^{cy} = \frac{(g_j^{cy} - d_i^{cy})}{d_i^h}, \hat{g}_j^w = \log(\frac{g_j}{d_i^w}), \hat{g}_j^h = \log(\frac{g_j^h}{d_i^h}) \tag{4}$$

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2, & \text{if } |x| < 1 \\ |x| - 0.5, & \text{otherwise} \end{cases}$$

where, $x_{i,j}^p = \{1, 0\}$ is the indicator function for matching the $i-$th default box to the $j-$th ground truth box of category p

6.1.2 Classification loss(or Confidence loss)

The confidence loss is the softmax loss over multiple classes confidences (c).

$$L_{conf}(x, c) = -\sum_{i \in Pos}^{N} x_{i,j}^p log(\hat{c}_i^p) - \sum_{i \in Neg} log(\hat{c}_i^0) \tag{5}$$

where $\hat{c}_i^p = \frac{e^{c_i^p}}{\sum_p e^{c_i^p}}$

6.2 Training

For training an object detection pipeline we need both positive and negative examples.Here, Online Hard example algorithm [11] is used for training. It reduces computation by running hard example mining hand to hand with the regular optimization cycle. In general, to pick a subset of negatives, we first train our network for couple of iterations, then it is run all along our negative instances before we pick the ones with the greater loss values. However, it is very computationally toilsome since we have possibly millions of images to process, and sub-optimal for the optimization process since we freeze our network while picking the hard instances. That is, you assume here all hard negatives you pick are useful for all the next iterations until the next selection. Which is an imperfect assumption especially for large datasets. Online means in this regard. OHEM solves these two aforementioned problems by performing hard example selection batch-wise. Given a batch sized k, it performs regular forward propagation and computes per instance losses. Then, it finds $m < k$ hard examples in the batch with high loss values and it only back-propagates the loss computed over the selected instances.

7 Results and Discussions

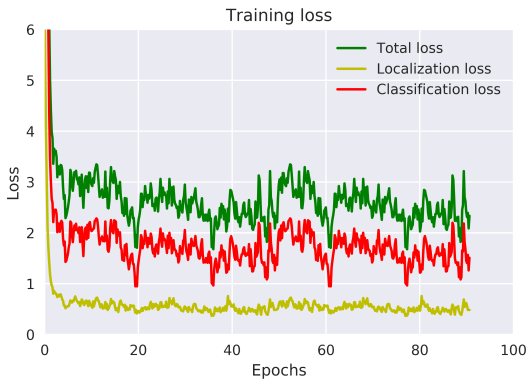


Figure 6: 2-D scatterplot of the Student Database

Category	Average Precision
Car	0.76
Pedestrian	0.72
Cyclist	0.38
Truck	0.51
Van	0.35
Overall	0.544

Table 1: Average Precision on the Validation set

The training was done on UW's Hyak cluster using a P100 GPU. Figure 6 shows how the three losses, total loss, classification loss and localization loss evolves with the number of epochs. These values have been exponentially smoothed out with a smoothing factor of 0.6. The trained model is then evaluated on the validation set we had set aside consisting of 20 % of KITTI training images. mAP scores are calculated on this validation set using VOC Evaluation criteria we have previously explained in section 3. As it can be seen from the mAP scores in Table 1 or from the example predictions on validation set images in Figure 7, the network is way better in detecting cars and pedestrians compared to other classes. The reason might be that the number of instances of car and pedestrian class is the maximum in the dataset , much higher than the other classes, with cyclist class coming third. Thus the network has enough examples to learn the corresponding features of the car and pedestrian really well. Also, we changed our image resolution from 1240×375 to 300×300 before feeding it to the Mobilenet feature extractor (This is because the Mobilenet we used was designed to work on 300×300 images). This naive trick is probably not the best way to go about resizing the images and what we could do in the future is maintain the aspect ratio while resizing and then pad the edges to get a 300×300 image.

Figure 7: Example predictions from validation set

7.1 Android Implementation and Benchmarking

Leeco's Le 2 model was used to test the android implementation of the trained model. We make use of the "Tensorflow Android Inference Interface" which provides the ability to load a tensorflow graph, set up the inputs and run the model to calculate particular outputs. First, the trained model is freezed into a protobuf file. This protobuf file is further optimized for inferece-only purposes by removing parts that are only needed during training. To get the entire model working on our android device we need the tensorflow libraries for Android (Tensorflow Android Inference Interface), create an Android App that has access to live camera stream and configure it to use these libraries, and then invoke the tensorflow model inside the app. After getting the entire set-up working we drove around University district to test how to asses its performance. Figure 8 and 9 consist of three different scenarios, one in each row - "stopping at an intersection", "following a cyclist", "taking a trun", "driving past a truck". The images in each row have a 1 second time difference with it's neighbouring frame. As it can be clearly seen that the setup isn't exactly real time and for most moving objects the bounding boxes seem to follow the object with a delay of about 1-2 seconds even when the car is stationary. The entire video can be accessed here.

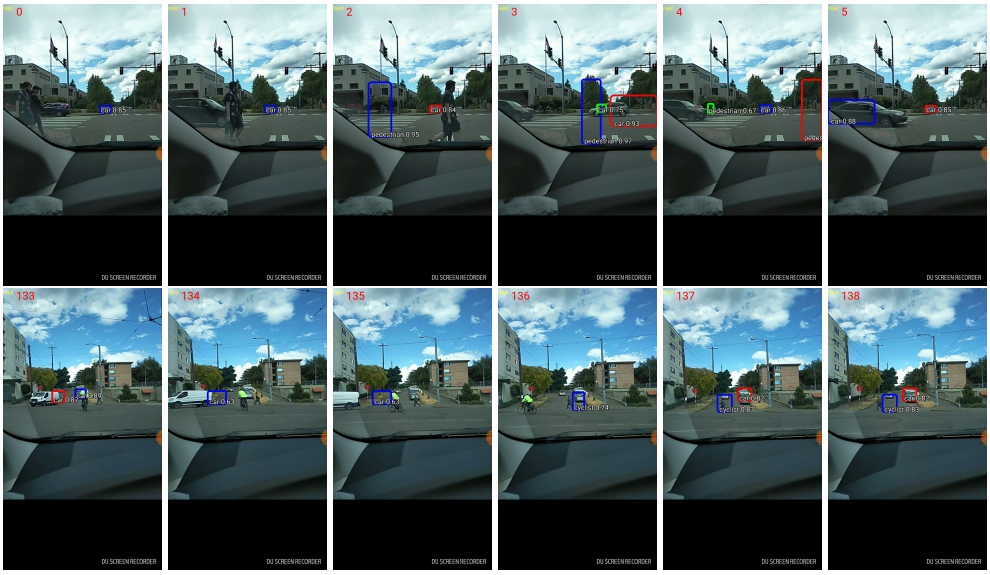


Figure 8: (a) Stopping at an intersection (b) Following a cyclist

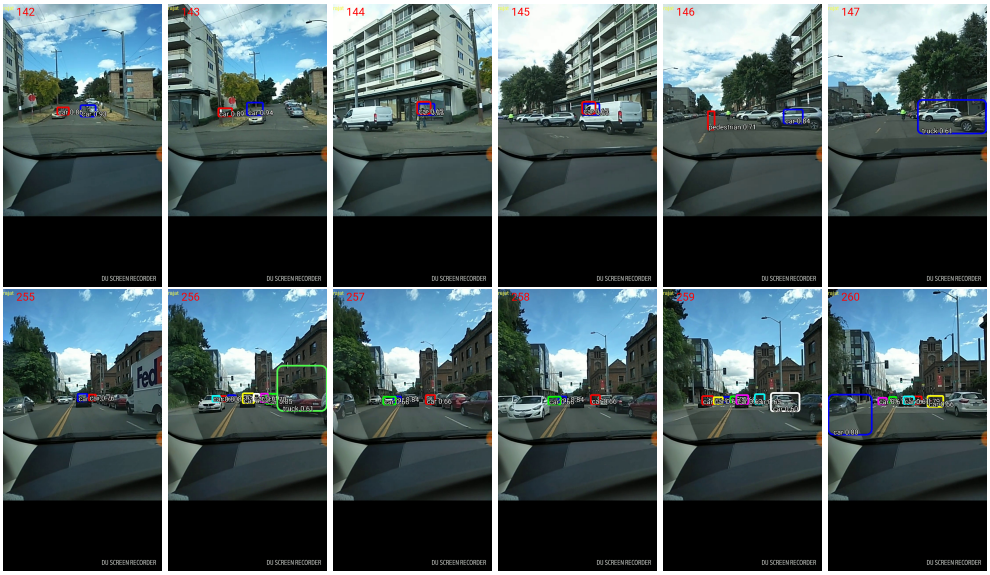


Figure 9: (a) Taking a turn (b) Driving past a truck

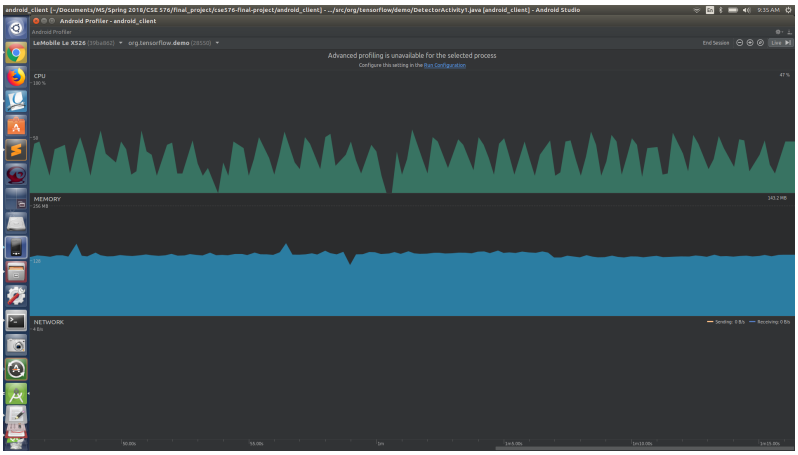


Figure 10: CPU and Memory usage while running the app

We used Android profiler to check the mobile's CPU and memory usage while the app was running. Figure 10 shows how the CPU usage bumps up every time the input data travels through the neural network for inference. It seems to be operating at 6-7 FPS which is four times less than the real time speed of 24 FPS. However we are are running this on the mobile's CPU and there is a scope for improvement in the future by exploting the device's GPU. Also the app just seems to use only 50% of the cpu and we could further improve performance by incorporating parallelization in our android java code. Moreover there are other mobile devices around with faster cpus and more RAM which could help increase the app's FPS and hence performance.

8 Conclusion

In this project we used transfer learning to finetune a COCO pre-trained Mobilenet-SSD detector to adapt to the KITTI dataset. The trained model was evaluated according to VOC's evaluation criteria and then exported to run on Android mobile device using Tensorflow Android Interface Inference libraries. This project indicates the possibility of using a smartphone locally and independtly for object detection in autonomous driving. It outlines the necessary steps and concepts required all the way from training an object detection model to deploying it on mobile for autonomous navigation.

The opportunities and challeneges mentioned here can serve as a steeping stone to future mobile based object detection applications.

The project's Bitbucket repository can be found here.

References

[1] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[2] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3354–3361. IEEE, 2012.

[3] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[4] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *IEEE CVPR*, 2017.

[5] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, page 20, 2015.

[8] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[9] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[11] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. Training region-based object detectors with online hard example mining. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 761–769, 2016.