CS341 Final Report: Towards Real-time Detection and Camera Triggering

Yundong Zhang yundong@stanford.edu
Haomin Peng haomin@stanford.edu
Pan Hu panhu@stanford.edu

Abstract

In this project, we aim at deploying a real-time object detection system that operates at high FPS on resource-constrained device such as Raspberry Pi and mobile phones. Although many systems have proved their success since the era of machine learning and neural network, most of the evaluations are done with high-end CPU or GPU. Nevertheless, real-world applications such as surveillance pose strict constraints on the resources of the device: low-power, small form factor, relatively good accuracy and fast speed, making it hard to find a good trade-off when designing the system. We present an object detection pipeline which is capable of working smoothly under the situation of traffic surveillance on Raspberry Pi 3 with only 1GB RAM and 1.2GHz ARM CPU that costs merely $35.99.

1. Introduction

Monitoring cameras are used almost everywhere, and are producing immense video stream everyday. Unfortunately, only a small portion of these data are interesting to the users. Let's take traffic surveillance as example: only the part of videos that actually has moving human or vehicles are useful, therefore recording everything all day long would inevitably become a waste of storage, and would make going through video a miserable work for searching and tracking. To address this problem, object detection system could be applied as a trigger to tell cameras when to record. However, most of the state-of-the-art algorithm have only been tested on GPU with strong computation ability, and are not likely to achieve the same speed and accuracy on less powerful devices with micro-processors only. In fact, the need for object detection algorithm to be applied on resource-constrained device is highly in demand and not limited to monitor camera (e.g. phones, sports watch). This serves as the primary motivation for our team to find a different detection system that is capable of working under embedded devices.

Problem statement. Our goal is to design a system that runs real-time object detection on Raspberry Pi 3. Formally, given a continuous camera live stream (with a resolution of 640*480), we want to successfully 'recognize' (draw bounding box and label the object) the moving object in a small amount of time. We choose Pi 3 as our platform because it is a standard representative of embedded device, and has been used widely as primary platform for devising low-cost system.

The main challenge of this project lies in three aspects: acceleration, compression and accurateness, where accuracy would be the trade-off for the first two aspects. We evaluate the models based on three metrics: mAP metric, detection speed (fps) and model size. As there is really no published work on the similar scope (stream that starts from a static frame), the final benchmark is under-defined at this stage. But we would like to have a prototype that can successfully perform real-time detection in about 5-10fps on Pi, with decent accuracy.

We make a number of contributions in this report, including:

- Test the performance of state of the art YOLO system and its quantized version on Raspberry Pi device and found that their speed is not eligible for real time use.
- Proposed a new object detection system with region proposal based on temporal information and reaches nearly 20X speed up and 15X less storage compared to YOLO.
- Combined SSD system and MobileNet to propose Mobile-Det, a detector version of MobileNet classifier and preliminarily tested its performance, provide a baseline for future improvement.

The paper is organized as following: we first introduce related works that attempts to solve similar problem, then explain our technical approach including network architecture and detection pipeline. After that we present experimental results and conclusion/discussion.

2. Related works

In this section we introduce some related works in object detection system, design of CNN architecture and recent works on model compression using quantization.

2.1. Object Detection System

Most previous CNN-based detection methods, for instance, R-CNN[15], begins by first proposing various locations and scales in a test image to be the input to classifiers of objects, then train and return the classifiers on each of these proposed regions to detect an object. After classification, there are usually post-processing to refine the bounding boxes, and rescore the boxes based on other objects in the scene. The improved versions of R-CNN, like Fast-RCNN[7] and Faster-RCNN[17], used more strategies to reduce computation of region proposal and reached a inference speed of 5 FPS on a K40 GPU device. Faster-RCNN based methods have been reported to reach top accuracies on KITTI[6], a dataset for object detection of traffic, but none of them pushed real time inference speed to a new level, which means there is still large room for improvement in the aspect of speed if people want to apply such methods to real life. A YOLO[16] system, however, broke the bottleneck by integrating region proposal and classification into a single regression problem straight from image pixels to bounding box coordinates and class probabilities and evaluate each full image with a single run. Since the whole detection pipeline is a single network, it can be optimized end-to-end directly on detection performance. YOLO is the first framework to reach real time detection standard with 45 FPS (on GPU) and a mAP of 63.4% on VOC2007 [4], but still has drawback in detecting smaller objects. This was later remedied by SSD [14] through combining anchor box proposal system of faster-RCNN and using multi-scale features to do detection layer. SSD further improved mAP on VOC2007 to 73.9% while maintaining similar speed as YOLO.

2.2. Network Compression with CNN Design

Although the original YOLO system could run fast on GPU, it's based on GoogLeNet [18] and the model size could be too huge to fit the RAM on device like Raspberry Pi, therefore strategies to reduce model size is another important aspect to be considered. An instinctive approach proposed by [3] is to apply singular value decomposition (SVD) to a pretrained CNN model. A similar method by [10] is Network Pruning, which begins by replacing parameters of a pretrained model that are below a certain threshold with zeros to form a sparse matrixwith and then performs a few iterations of training on the sparse CNN. Some of the recent researches also shows that carefully designed CNN structures could considerably reduce model size while not compromising or even improving accuracy. In SqueezeNet[12] structure, a fire module microstructure was raised and employed some of the design intuition to cut the size of parameters of convolutional layers. A SqueezeNet consist of several fire modules was reported to achieve 50X reduction in model size compared to AlexNet [13], while meeting or exceeding the top-1 and top-5 accuracy of AlexNet. A hybrid child of the thoughts of both YOLO and SqueezeNet is Squeezedet[22]. It is mainly based on YOLO but replace the detection layer of YOLO with a designed structure called ConvDet layer and use SqueezeNet as the backbone of CNN network. ConvDet layer enables SqueezeDet to generate tens-of-thousands of region proposals with much fewer model parameters compared to YOLO, and the use of SqueezeNet further compressed the size to 7.9 MB while maintaining a FPS of 57.

2.3. Model Compression With Quantization

Apart from redesigning CNN and cut the total number of parameters, another way to achieve the goal of accelerating inference process and reducing storage of CNN is to cut the computation of multiplication in convolution layers.

Provided that the multiplication of float is much more time consuming than integer, an instinct is to transform float numbers to 8 bit integers. This could be implemented using shrinks by storing the min and max value of weights for each layer, and then compressing each float value to an eight-bit integer representing the closest real number in a linear set of 256 within the range [21]. This method in theory could reduce the model size by 75%. A support package recently released by TensorFlow develop team enables the 8 bit quantization of tensors [9].

3. Technical approach

In this section we introduce our technical approach to the object detection problem. We describe two major parts of our solution in the following subsections, including network architecture and processing pipeline.

3.1. Network architecture

The backbone architecture of our system is MobileNets [11], a novel deep NN model proposed by Google, designed specifically for mobile vision applications. We first introduce the intuition behind MobileNets, then describes our model structure.

3.1.1 Depthwise separable convolution

The main thing that makes MobileNets stand out is its use of depthwise separable convolution (DSC) layer, as shown in Fig. 1.

Depthwise separable convolution replaces the standard convolution with a two-step operation: 1. depthwise convolution, where each D_FxD_F filter is only in charged of filtering a single depth of the input feature map; 2. pointwise convolution: a simple 1x1 convolution layer that is used for combining channel information. DSC makes the convolution operation much efficient meanwhile uses much

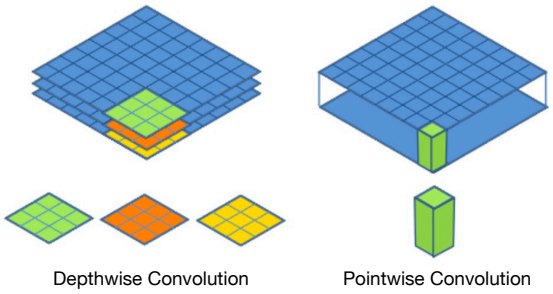


Figure 1. Illustration of depth-wise separable convolution layer structure [23]

less parameters. We highlight the parameters and computation cost at the following table:

Table 1. Parameter Size and Computation Cost in Depthwise Separable Convolution

Layer	Parameter Size	Computation Cost
Standard Conv	$F \times F \times C_1 \times C_2$	$F \times F \times D_M \times D_M \times C_1 \times C_2$
Depthwise Separable	$F \times F \times C_1 + 1 \times 1 \times C_1 \times C_2$	$F \times F \times D_M \times D_M \times C_1 + 1 \times 1 \times C_1 \times C_2$

The reduction of computation cost is therefore:

$$\frac{F \times F \times D_M \times D_M \times C_1 + C_1 \times C_2 \times D_N \times D_N}{F \times F \times D_M \times D_M \times C_1 \times C_2 \times D_N} = \frac{1}{N} + \frac{1}{F^2} \quad (1)$$

And hence the parameter reduction:

$$\frac{F \times F \times C_1 + 1 \times 1 \times C_1 \times C_2}{F \times F \times C_1 \times C_2} = \frac{1}{C_2} + \frac{1}{F^2} \quad (2)$$

Typically, in our setup, we use 3x3 convolution and the saving of computation is about 8 to 9 times, so as the model size. Nevertheless, this efficient operation does not sacrifice accuracy a lot, as we can see in the Table 2.

The Intuition behind DSC: studies [2] have showed that DSC can be treated as an extreme case of inception module, where cross-channel correlations and spatial correlations is completely decoupled, and hence can be mapped independently. In fact, if we check the receptive field of the resulting feature maps after DSC, we see that it is the same as standard convolution. The only difference is that DSC by making a strong assumption as stated above, exploit parameter sharing and degree of freedom more efficiently.

3.1.2 Model Structure

We now demonstrate our network structure in Fig. 2, which mainly consists of a series of DSC module. Each DSC module contains is basically a DSC layer, with inserted RELU and batch normalization operation between the depth-wise and point-wise operations. Note that the first layer is just the standard convolution. The last layer is an average pooling followed by a fully-connected layer. In sum, the model is similar to VGG-like network, removing the use of residual connections for speed improvements. In (b), we show that some of the layers are fixed, this is used for transfer learning, as we will discussed in section 5.1.

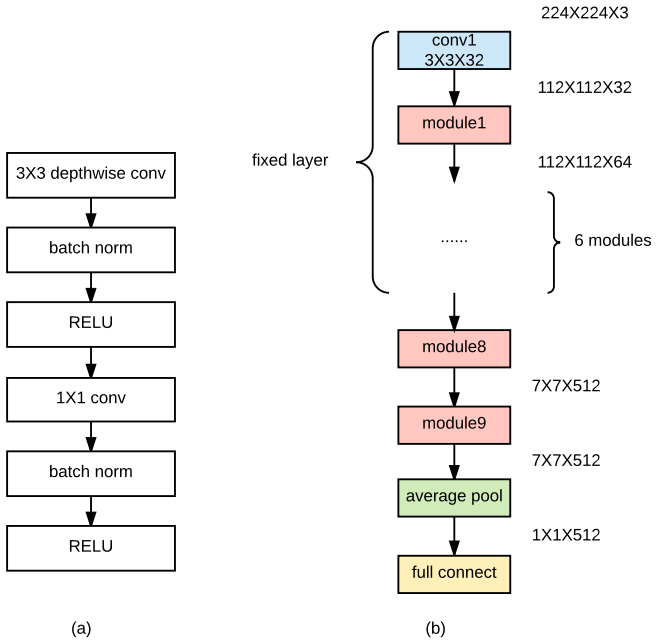


Figure 2. (a) The structure of a typical DSC module; (b) Structure of our MobileNets model

3.2. Detection Pipeline

In this subsection we introduce our detection pipeline, which includes an initialization process, region proposal and classification, as shown in Fig. 3. We explain each of the three steps in the following subsections.

3.2.1 Reference Frame

We first configure camera as video stream. However, there are several ways to generate reference frame:

- Delay two seconds to wait for the camera to stable, and use the first frame as reference frame. Do not update the reference frame. This is the most straight forward method, but clearly suffering from the problem that the light condition will change overtime. The background of the input frame will not align with the reference frame, disturbing following region proposal process.

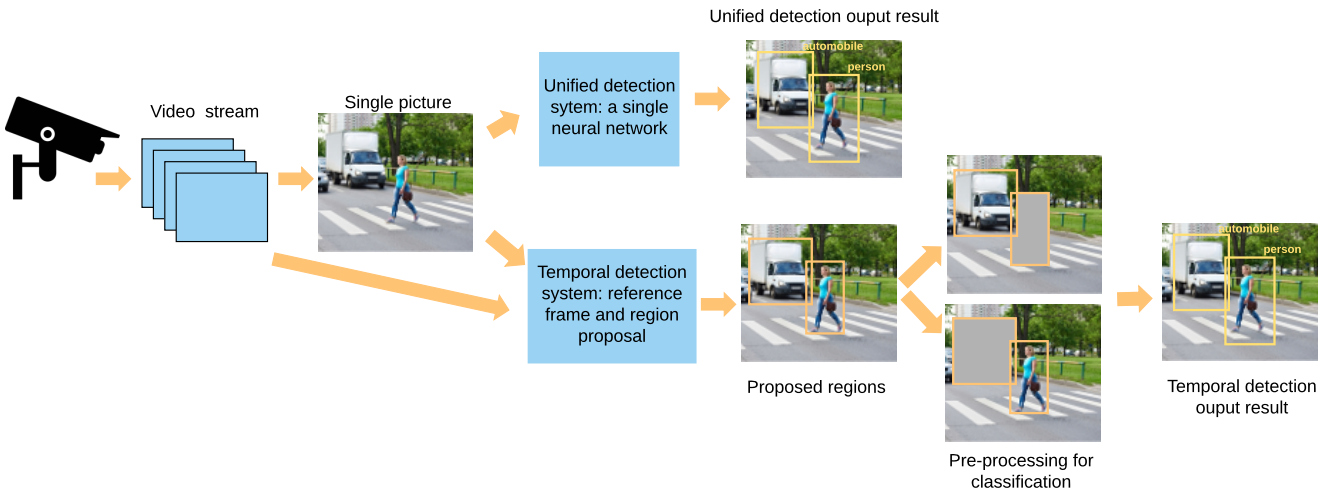


Figure 3. Annotated picture describing our experimental setup.

- An alternative method would be that we update reference periodically using the following rule: $r_{i,j} = (1-\beta)r_{i,j} + \beta c_{i,j}$ where r is reference frame, c is current input frame, and β is updating rate. The larger the updating rate, the faster we update reference frame. The problem of this method is that, if an object stays in the field of view for long enough time, it will become part of the background.

- Selective update. The method is very similar to the one mentioned above. But instead of using every input frame to update the reference frame, we only update using input frames when there is no object in the input frame. Still, this method may suffer from the problem that there might be objects staying in the field of view for a long time, making the update interval infinitely long, although we can always update areas that do not have object rather than the entire frame.

In our setup, we choose the selective update method for picking and updating our reference frame, i.e. using momentum update only when there is no object in the current frame.

3.2.2 Region proposal

We illustrate the process of region proposal using some example inputs, shown in Fig. 4. The region proposal include the following steps:

1. Get frame difference by subtracting current frame with reference frame. The pixel-wise value is thus defined as $diff_{i,j} = |input_{i,j} - ref_{i,j}|$. The same thing applies for all three channels. The result is shown in Fig. 9(c).

2. It is very common that there is slight offset between the backgrounds of input frame and reference frame. Since the offset is usually only a few pixels, we apply a 2-D Gaussian filter with size of 21×21 to Fig. 9(c) to smooth out the sharp lines caused by offset in images. The result is shown in Fig. 4(d).

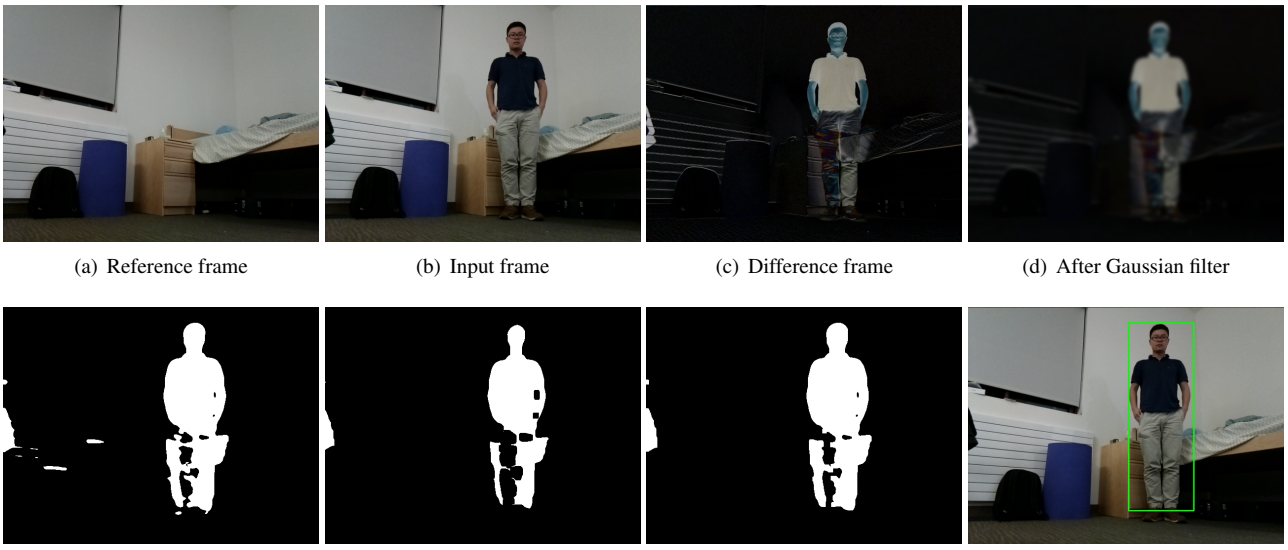
3. We then pass the Gaussian filtered difference frame into a threshold filter. The pixel value is set to 255 if any of the RGB channel exceeds a certain threshold (usually between 40 50), otherwise the pixel value is set to 0. The result is shown in Fig. 4(e).

4. From Fig. 4(e) we could find that the thresholded frame includes small areas caused by noise and camera shake, which is not desired. To remove these small areas, we put the frame into an erosion filter. We choose a filer size of 5×5, and repeat the process twice. From Fig. 4(f) we could find the small areas are disappeared.

5. However, a side effect of erosion filter is that, the boundary of target object also being eroded. From Fig. 4(f) we could find that the left leg is separated from the rest of the body. To compensate for the effect of erode filter, we apply the reverse operation, dilate filter to the frame with exactly the same kernel. The result is shown in Fig. 4(g). By doing this we can connect objects back to where they supposed to be connected.

6. After all the filtering process we find contour on difference frame and construct rectangle bounding box based on contour. We discard region that is overly small, say 2000 square pixels (approximately 44*44 pixels). The result is shown in Fig. 4(h). It can correctly draw bounding box on a human.

The average time spend on region proposal is 83ms.



(a) Reference frame (b) Input frame (c) Difference frame (d) After Gaussian filter

(e) Thresholded difference frame (f) After Erosion filter (g) After Dilate filter (h) Output result

Figure 4. An illustration of our region proposal pipeline, from reference frame to detection output.

3.2.3 MobileNets Classifier

Now it's time for our MobileNets to label each proposed image, forwarded by the region proposal system. We preprocess the current frame and generate multiple images by zero-outing all but one objects each time using the input box coordinates. As a result, we get the same number of images as the detected boxes for each input frame, we then feed all the images to the classifier to generate labels. Finally, We take the classification output and combine them with the bounding boxes, overlaying on the original frame.

Readers may be curious about why we do not use the cropped bounding box directly as input to the classifier. The reason is that the classifier is trained for a fixed size images, while in practice small objects appear very often. If we upsample the objects, the resulting images might have too low resolution to be recognized correctly. Another reason is that we assume the background information is helpful for prediction, hence we keep it.

3.3. Dataset and Preprocessing

Our image data is supplied by PASCAL VOC2012 detection dataset [5]. The dataset consists of 20 classes, including objects most commonly captured in traffic cameras like bus, car, bicycle, motorcycle and person. The data has been split into 50% for training/validation and 50% for testing, with the distributions of images and objects by class are approximately equal across the training/validation and test sets. This comes to a training set of 5717 images and a validation set of 5823 images.

Since we assumes that the input to the classifier in our system only contains one target object, we preprocess pictures with multiple objects to guarantee that each processed image only contains one region of interest. We implemented this step by first replacing the regions of all objects in a image with mean value of the picture for 3 channels respectively and then placing a single region back to it's original place, loop this for each region of interest and we would generate a series of pictures with one object, as shown in Fig. 5.

This step produced a training set of 13609 pictures and validation set of 13841 pictures. We also resize each picture to 224x224 pixels to ensure pictures are of the same size.

4. Mobile-Det

We also implement a detector version of MobileNet, namely Mobile-Det, by combining MobileNet classifier and Single Shot MultiBox Detector (SSD) framework [14]. The reason we want to do this is to further analyze the benefit of MobileNet model, and have a fair comparison between the state-of-art detection model like VGG-based SSD and YOLO. The details of SSD is massive and beyond the scope of this project, so we will only have a brief introduction to how it works in the following parts.

In short, SSD framework uses multiple feature layers as classifiers, where each feature map is evaluated by a set of different (aspect ratio) default boxes at each location in a convolutional manner, and each classifier predicts class scores and shape offset relative to the boxes. At training time, a default box is considered to be predicting correctly if its jaccard overlap with the ground truth box is larger than the threshold (0.5). The loss is then measured by both the confident score and localization score (Smooth L1 [8]). The

Figure 5. Data preprocessing by covering other region of interest

demonstration of the SSD model is shown in fig. 6:

The structure of Mobile-Det is similar to ssd-vgg-300: the original SSD framework. The difference is that rather than using VGG, now the backbone is MobileNets, and also all the following added convolution is replaced with depth-wise separable convolution.

The benefit of using SSD framework is evident: now we have a unified model and is able to train end-to-end; we do not rely on the reference frame and hence the temporal information, expanding our application scenarios; it is also more accurate in theory. However, the main issue is that, the model becomes very slow, as a large amount of convolution operations are added.

5. Experimental Results

In this section we describe some experimental results about our system, including a brief description about our experiment setup, some different methods we've tried and their performance/statistics.

5.1. Transfer Learning of MobileNet

Our primary use case is detecting traffic objects, including pedestrians, cars, bus and so on. Although there is ImageNet pre-trained weights available online for MobileNets[ref], the model is unnecessarily large than our need. There is also no class about people in ImageNet-1000 datasets. On the other hand, PASCAL VOC 2012 itself, only contains about 5K images (even after pre-processing we only have 15k). Hence, we need to use transfer learning technique in order to train a high accuracy model.

The transfer learning process is rather simple: we initialize and fix the first 7 layers of weights based on the pre-trained ImageNet model (see fig. 2), and train the rest of the layers by back-propagation. Compared to the original MobileNets Model, we use less layers, based on the intuition that now we have less classes. This helps us achieve a fast inference speed, smaller model size while sacrifice little to no accuracy. We call this 'truncated' model as MobileNets-Small.

Note that the model for our customized dataset is under-optimized (i.e. we haven't spent too much time on fine-tuned it; hence a little more performance gain should be expected)

Table 2. MobileNets Classifier Accuracy (ImageNet accuracy is cited from the Google paper [11]

Model	Dataset	Accuracy (Top-1)
MobileNets-Full	ImageNet	70.5%
VGG16	ImageNet	71.5%
MobileNets-Full	Customized VOC	69.8%
MobileNets-Small	Customized VOC	67.6%

5.2. Description of setup and sample result

Our experimental setup is shown in Figure. 7: including a Raspberry Pi V3 board, a 5 million pixel RGB camera, a cooling fan and a 10 inch HDMI display. The Raspberry pi is based on a 1.2GHz 64-bit quad-core ARMv8 CPU with 1GB of DRAM and 32GB SD card storage.

We naively transplant Tiny-Yolo-Voc to our setup and test its performance. An example of detection result is shown in Figure. 8. It correctly detects and locates a chair, a person and a monitor (although there are actually two monitors).

5.3. Detection Results and Model Comparison

In this section we compare the performance of several different detection models, we use the state-of-art model yolo2 as comparison. For the 8-bit quantization model, we fully quantize both the weights and operation (i.e. matrix multiplication are also done in 8-bit). Table 3 contains all the results:

We can see from the table that the original full-version yolo2 pretrained model even cannot fit in Pi with 1G memory, not to mention other micro-processors which typically has smaller RAM. For tiny yolo2, which is going to be our baseline model, it is able to run detection in 0.487fps. That is to say, object movements can only be detected after 2s, this is however unacceptable in real-world scenario, especially for our intended application–traffic.

For the models quantized using tensorflow graph-transform tool, in theory, we should observe that both the model size and speed get better. However, in the experiment it does not perform well in terms of speed. There are currently two major problems out here: 1. during inference, the model needs to re-evaluate the maximum and minimum value of the input at each layer, which typically slows the model; 2. as documented by the tensor-

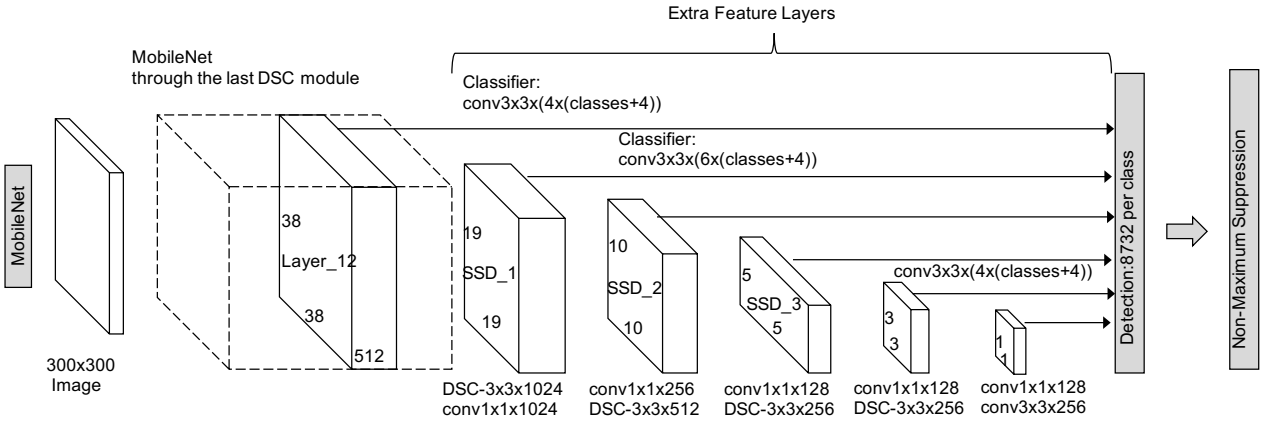


Figure 6. Mobile-Det: SSD-based detection with MobileNet as backbone, modified based on [14]

Table 3. Comparison of different models (Note: the result of Mobile-Det is still preliminary)

Type of model	Size on disk	Detection speed	mAP
Yolo2-full	269.9MB	Out of memory	76.8
Yolo2-tiny	60.5MB	0.487fps	57.1
Yolo2-full eight-bit	64.4MB	0.153fps	61.3
Yolo2-tiny eight-bit	15.2MB	0.343fps	49.8
Temporal Detection	4.4MB	2.566fps	*
Mobile-Det	27.5MB	0.712fps	41.9

*Under-defined, please see section 5.4

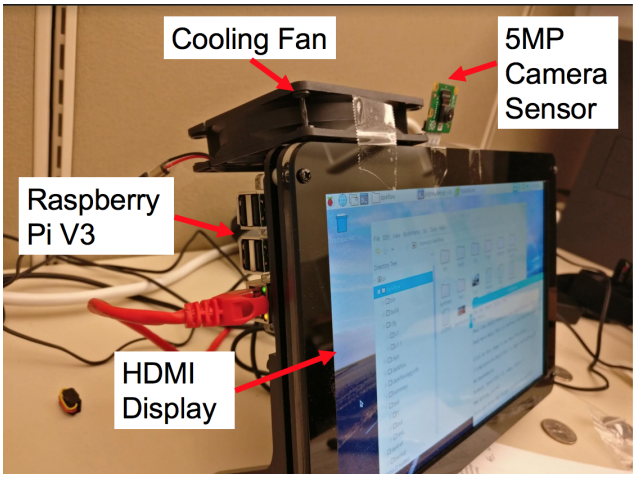


Figure 7. Annotated picture describing our experimental setup.

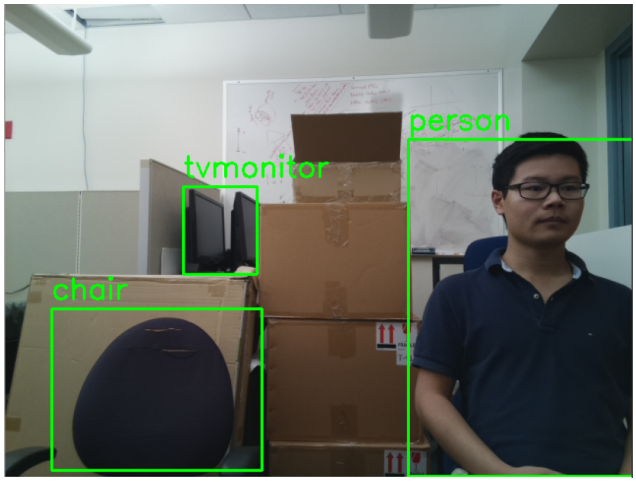


Figure 8. Example of detection result.

flow team, the quantized-version operations are not well-optimized yet, and hence unless you are using 8-bit instruction set/machine, the eight-bit model will even be slower. Despite that, we could observe that we don't lose much accuracy after quantizing the model of both tiny and full yolo2, this shows quantization is a tool of strong potential and could be used for deeper compression following any network compression strategies.

For our proposed Temporal Detection model, we can see that the inference speed is increased by 5x, compared to the fastest tiny yolo2. Meanwhile, the uncompressed model is almost 20x smaller than tiny-yolo, 5x smaller than its 8-bit quantized version. This is well expected since our region proposal is based on frame difference and therefore could takes use of temporal correlation between frames. This proposing method also well served the purpose of monitor camera since most users are only concerned about moving objects in frames and are not interested in trees or other

(a) Video1 (b) Video2 (c) Video3

Figure 9. Sample images in the VISOR video dataset.

environment objects in background. Compared to the region proposal of YOLO-like system which scans through each region of every frame and aims to label everything, our system would undoubtedly run much faster. However, the model suffers from low mAP and are highly sensitive to the use case, and hence needs more fine-tuned and further improvements on refining the differential region proposal methods and post-processing of bounding boxes.

Finally, we have a preliminary result of Mobile-Det, which achieves both smaller size and fast inference speed compared to tiny-yolo. We call this preliminary because unfortunately we ran out of time training the model on more datasets. The reported results of yolo2 are trained from both VOC2007 and VOC2012, with carefully finetuning, while our Mobile-Det just finished a preliminary training (haven't done hyper-parameters tuning) on VOC2007. The training is slow, the model takes two days to start from an mAP of 0 to 41.9%. Nevertheless, based on the current results, we are confident to achieve a mAP higher than 50%, while gaining a 1.5x increase of inference speed compared to tiny-yolo. Unfortunately it is still much slower than the temporal detection methods.

5.4. Benchmark of region proposal in temporal detection

We use three videos from VISOR [20] to test the accuracy of our region proposal. We choose this dataset because the camera in the selected dataset is stationary and the scenario is well-suitable for our target application. Three sample videos are shown in Fig. 9. One challenge we're facing in working with the dataset is that, it is required to register an account to download the label data. To skip the time waiting for the registration to get approved, we use Yolo2 as our baseline.

The mAP accuracy of our region proposal is shown in Table. 4.

Table 4. mAP accuracy of region proposal for three video clips

	Video 1	Video 2	Video 3	Combined
No. of frames	662	425	919	2006
mAP	0.48	0.72	0.50	0.54

6. Conclusion

In this project, we tested a variety of detection models, including the state-of-art YOLO2, and two our newly proposed models: Temporal Detection and Mobile-Det. We conclude that the current object detection methods, although accurate, is far from being able to be deployed in real-world applications due to large model size and slow speed. Our work of Mobile-Det shows that the combination of SSD and MobileNet provides a new feasible and promising insight on seeking a faster detection framework. Finally, we present the power of temporal information and shows differential based region proposal can drastically increase the detection speed.

7. Future work

There are a few aspects that could potentially improve the performance but remains to be implemented due to limited time, including:

- Implement an efficient inference module of 8bit float in C++ to better take advantage of the speed up of quantization to inference step on small device.
- Try to combine designed CNN modules like MobileNe t module and fire module with other real-time standard detection frameworks.

8. Acknowledgement

We thank our project TA Han Song for proving insightful discussions on model compression.

Github repo we've referenced include: [1] [19] [11] [9]

The repository above has all been linked in the reference section.

References

[1] balancap. A tensorflow implementation of google's mobilenets: Efficient convolutional neural networks for mobile vision applications. https://github.com/balancap/SSD-Tensorflow.

[2] F. Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint arXiv:1610.02357*, 2016.

[3] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

[4] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[5] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[6] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[7] R. Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[8] R. Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[9] Google. Tensorflow Graph Transform Tool. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/graph_transforms/README.md.

[10] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[11] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[12] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[14] 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.

[15] T. D. R. Girshick, J. Donahue and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *Computer Vision and Pattern Recognition (CVPR),2014*, 2014.

[16] J. Redmon, S. Divvala, R. Girshick, and A. 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.

[17] 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.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[19] thtrieu. Translate darknet to tensorflow. load trained weights, retrain/fine-tune using tensorflow, export constant graph def to mobile devices. https://github.com/thtrieu/darkflow.

[20] VISOR. Visor video dataset. http://imagelab.ing.unimore.it/visor/video_videosInCategory.asp?idcategory=3.

[21] P. Warden. How to Quantize Neural Networks with Tensorflow. https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/.

[22] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. *arXiv preprint arXiv:1612.01051*, 2016.

[23] Zehaos. A tensorflow implementation of google's mobilenets: Efficient convolutional neural networks for mobile vision applications. https://github.com/Zehaos/MobileNet.