Real-time object detection and monitoring system

Submitted in Partial Fulfilment of the Requirements for the Degree of

BACHELOR OF ENGINEERING

in

Electronics and Communication

Names of the students:

Anudeep Kumar	USN: 1MS16EC016
Gaurav Chaudhary	USN: 1MS16EC034
Jeevan K	USN: 1MS16EC043
Jesuraja Bandekar	USN: 1MS16EC044

Under the guidance of:

Dr. Raghuram S
Associate professor
Department of ECE

Department of Electronics and Communication

Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)
Accredited by National Board of Accreditation & NAAC with 'A' Grade
MSR Nagar, MSRIT Post, Bangalore – 560054

2020

Certificate

This is to certify that the project work titled "**Real-time Object Detection and Monitoring System**" is carried out by **Anudeep Kumar (USN: 1MS16EC016)**, **Gaurav Chaudhary (USN: 1MS16EC034)**, **Jeevan K (USN: 1MS16EC043)** and **Jesuraja Bandekar (USN: 1MS16EC044)** bonafide students of Ramaiah Institute of Technology, Bangalore, as part of Final year project work carried out in eighth semester of Bachelor of Engineering in Electronics and Communication during the year 2019-20. It is certified that all corrections / suggestions indicated for Internal Assessment have been incorporated in the report. The report has been approved as it satisfies the academic requirements prescribed.

Guide	HOD	Principal
Dr. Raghuram Srinivasan	**Dr. Sethu Selvi**	**Dr. N V R Naidu**
Associate Professor,	Professor and HOD,	Principal,
ECE Department	ECE Department	RIT, Bangalore
RIT, Bangalore	RIT, Bangalore	

Name & Signature of Examiners with date:

1.

2.

Declaration

We hereby declare that the mini-project entitled "**Real-time Object Detection and Monitoring System**" has been carried out independently at Ramaiah Institute of Technology under the guidance of Dr. Raghuram Srinivasan, Associate Professor, Department of Electronics and Communication Engineering, RIT, Bangalore.

Name & Signature of Examiners with date:

1. Anudeep Kumar (USN: 1MS16EC016)

2. Gaurav Chaudhary (USN: 1MS16EC034)

3. Jeevan K (USN: 1MS16EC043)

4. Jesuraja Bandekar USN: (1MS16EC044)

Place:

Date:

Acknowledgement

The immense satisfaction that accompanies the successful completion of the project would be incomplete without the mention of the people who made it possible. We consider it our honour to express our deepest gratitude and respect to the following people who always guided and inspired us during the course of the Project.

We are deeply indebted to **Dr. N. V. R. Naidu**, Principal, RIT, Bangalore for providing me with a rejuvenating undergraduate course under a very creative learning environment.

We are much obliged to **Dr. S. Sethu Selvi**, Professor & HOD, Department of Electronics and Communication Engineering, RIT, Bangalore for her constant support and motivation.

We sincerely thank our guide **Dr. Raghuram Srinivasan**, Associate Professor, Department of Electronics and Communication Engineering, RIT, Bangalore and express our humble gratitude for his valuable guidance, inspiration, encouragement and immense help which made this project work a success.

We sincerely thank all the faculty members of Department of E&C, RIT for their kind support to carry out this project successfully.

Last but not the least we would like to express our heartfelt gratitude to our parents, relatives and friends for their constant support, motivation
and encouragement.

Abstract

Object detection and monitoring is one of the applications of Deep Neural Networks that has shown great promise in recent years. Networks like R-CNN, YOLO, and SSD have proven accuracies for a large class of objects, all within the same image. At the next level of application, DNNs have to be deployed at the edge, to add intelligence to traditional surveillance modalities. As part of this effort, we propose to use DNNs for traffic monitoring through surveillance video cameras. However, the architecture has to be extremely lightweight if it has to run on embedded systems in real-time. The primary objective of this project is to develop such networks and test them on a low cost embedded platform such as Raspberry-Pi. Applications such as congestion, lane violations, obstacle detection etc. are studied as part of this project.

Nomenclature

YOLO	You Only Look Once
SSD	Single Shot Detection
DNN	Deep Neural Network
DSOD	Deeply Supervised Object Detection
DDB	Depth-wise dense block
CNN	Convolutional Neural Network
RPN	Regional Proposal Network
BN	Batch-Normalization

Contents

List of figures

List of tables

Chapter 1:
Introduction

1.1 Introduction

In today's fast moving world and the ever increasing number of cars, the roads are full of roaring fast moving machines that take us to our destination on time and gives us thrill at times too but also due to tiny bit of carelessness result in disastrous mishaps that leave us thinking what could have been done to prevent that. Real-time vehicle detection, tracking of vehicles is of great interest for engineers and is a need of the society in general for comfortable, smooth and safe movements of vehicles in cities. If we are able to accurately track the moving vehicles sudden changes in acceleration we can implement automatic control of speed and prevent numerous road disasters.

1.2 Problem Statement

Study Lightweight Networks to detect and track vehicles and test them on a low cost embedded platform such as Raspberry-Pi.

1.3 Objectives

- Study various lightweight neural networks with low processing requirements.
- Test on Raspberry pi and determine the optimum network for applications of vehicle Detection and Tracking.
- Hence we aim to study and inspect the models and try to implement and observe the performance of the models and conclude with the model with a practical performance.

Chapter 2:
Literature Survey

2.1 Paper 1:

Li, Yuxi, et al. "Tiny-dsod: Lightweight object detection for resource-restricted usages." *arXiv preprint arXiv:1807.11013* (2018). [1]

This paper considers the resource and accuracy trade-off for resource-restricted usages during designing the whole object detection framework. This is based on the deeply supervised object detection (DSOD) framework. Tiny-DSOD is dedicated to resource-restricted usages. It introduces two innovative and ultra-efficient architecture blocks :

1. Depth-wise dense block (DDB) based backbone,
2. Depth-wise feature-pyramid-network (D-FPN) based front-end.

Dataset :

- PASCAL VOC 2007
- KITTI
- COCO

Tiny-DSOD achieves 72.1% mAP with only 0.95M parameters and 1.06B FLOPs

2.2 Paper 2:

Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017). [2]

A class of efficient models called MobileNets for mobile and embedded vision applications. MobileNets are based on a streamlined architecture that uses depth-wise separable convolutions to build light weight deep neural networks. Consist of two simple global hyper-parameters that efficiently trade off between latency and accuracy. These hyper-parameters allow the model builder to choose the right sized model for their application based on the constraints of the problem:

1. Width Multiplier: Thinner Models

The width multiplier α is for thinning a n/w uniformly at each layer.

2. Resolution Multiplier: Reduced Representation

The resolution multiplier ρ reduces the computational cost of a neural network.

The MobileNet model is based on depth-wise separable convolutions which is a form of factorised convolutions which factorise a standard convolution into a depth-wise convolution and a 1×1 convolution called a point-wise convolution.

Dataset :

- ImageNet (Classification)

2.3 Paper 3:

Liu, Wei, et al. "Ssd: Single shot multi-box detector." European conference on computer vision. Springer, Cham, 2016. [3]

SSD discretises the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. The network generates scores for the presence of each object category in each default box and produces adjustments to the box to better match the object shape at prediction time. The network combines predictions from multiple feature maps with different resolutions to naturally handle objects of various sizes.

It eliminates proposal generation and subsequent pixel or feature resampling stages and encapsulates all computation in a single network making SSD easy to train and straightforward to integrate into systems that require a detection component. Provides a unified framework for both training and inference.

Dataset :

- PASCAL VOC
- COCO
- ILSVRC

For 300 × 300 input, SSD achieves 74.3% mAP1 on VOC2007. SSD has much better accuracy even with a smaller input image size compared to other methods.

2.4 Paper 4:

Sang, Jun, et al. "An improved YOLOv2 for vehicle detection." Sensors 18.12 (2018): 4272. [4]

YOLOv2 is improvement on YOLO algorithm. Various are used techniques used to improve efficiency of the network.

Batch normalization is used on all convolutional layers in YOLOv2. All fully connected layers are removed and anchor boxes are used to predict bounding boxes. One pooling layer is removed to

increase the resolution of output. The sizes and scales of Anchor boxes were pre-defined without getting any prior information. Uses k-means clustering which leads to good IOU scores. Network runs faster at smaller-size images.

Implementation with Dark-net-19. Dark-net-19 has many 1×1 convolutions to reduce the number of parameters. Dark-net-19 can obtain good balance between accuracy and model complexity.

YOLOv2 gets 76.8% mAP on PASCAL VOC 2007.

2.5 Paper 5:

Wu, Bichen, et al. "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2017. [5]

SqueezeDet, a fully convolutional neural network for object detection. Designed aiming for having small model size and energy efficient to enable embedded system deployment. In this network, convolutional layers not only used to extract feature maps but also as the output layer to compute bounding boxes and class probabilities. The detection pipeline of the model only contains a single forward pass of a neural network, thus it is extremely fast.

The model is fully-convolutional, which leads to a small model size and better energy efficiency. Main features :

- Stacked convolution filters to extract a high dimensional, low resolution feature map for the input image.

- ConvDet layer : A convolutional layer that is trained to output bounding box coordinates and class probabilities.

    - Input is the feature map

    - Computes a large amount of object bounding boxes

    - Predict their categories

- Bounding boxes are filtered to obtain the final detection.

2.6 Paper 6:

Mudunuri, Sivaram Prasad, Soubhik Sanyal, and Soma Biswas. "GenLR-Net: Deep framework for very low resolution face and object recognition with generalization to unseen categories." *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, 2018. [6]

A novel deep CNN architecture which can handle large difference in resolution as well as generalize to unseen categories. Recognise objects in LR input images with high resolution(HR) images in database. This is useful since the input to the model is of low resolution compared to the images in the dataset.

There are systematic introduction of different kinds of constraints at different stages of the architecture. Techniques used in this approach can be used to improve the accuracy of network. The average prediction log-loss after the softmax layer is employed during the training to minimise the classification error.

Chapter 3:
Methodology

3.1 Need for lightweight model

We have a number of Object detection architectures which perform at an exceptional level with comparable or even better performance with respect to humans. What draws them back is the computational power required and the lack of mobility of the system. One can argue use of the web but that comes with the restriction of High speed data transfer. In these restrictions a lightweight net with lower processing requirement can help implement a real time economical system for large scale application. A lightweight model also enables us to run it on limited and compact hardware that are mobile and easy to install, like microcontrollers.

3.2 Lightweight architectures for embedded implementation

3.2.1 Tiny DSOD

Object detection is considered as a crucial and challenging task in the field of computer vision, since it involves the combination of object classification and localization within a scene. Accompanied with the development of modern deep learning techniques, lots of convolutional neural network (CNN) based detection frameworks, including Faster R-CNN ,YOLO [4] , SSD [3] and their variants , have been proposed and greatly promote the accuracy for object detection. In spite of the state-of-the-art accuracy these models achieved, most of them are resource hungry as they have both high computing complexity and large parameter size (or large model size). High computing complexity requires computing units with higher peak FLOPs, which usually increases the budget for power consumptions. The speed and accuracy tradeoff has been extensively studied in [1]. However, resources are not only computing resources, but also memory resources. Large model size yields large persistent memory requirements, which is not only costly, but also power inefficient for low-end embedding applications due to frequently persistent memory access. Due to these two limitations, the majority of current object detection solutions are not suitable for low-power scenarios such as applications on always-on devices or battery-powered low-end devices.

To alleviate such limitations, recently many researches were dedicated to ultra-efficient object detection network design. For instance, YOLO [4] provides a lite version named Tiny-

YOLO, which compresses the parameter size of YOLO to 15M and achieves a detection speed of more than 200 fps on PASCAL VOC 2007 dataset . SqueezeDet [5] introduces SqueezeNet based backbone into the YOLO framework for efficient autonomous driving usages. MobileNet-SSD adopts MobileNet [2] as backbone in the SSD framework, which yields a model with only 5.5M parameters and 1.14B FLOPs of computing on PASCAL VOC 2007 dataset. Although these small networks reduce the computation resource requirement to a large extent, there is still a large accuracy gap between small networks and the full-sized counterparts. For instance, there is a 9.2% accuracy drop from SSD (77.2%) to MobileNetSSD (68.0%) on PASCAL VOC 2007. In a nutshell, these small detection networks are far from achieving a good trade-off between resources (FLOPs & memory) and accuracy.

Lightweight Object Detection for Resource-Restricted Usages

Object detection has made great progress in the past few years along with the development of deep learning. However, most current object detection methods are resource hungry, which hinders their wide deployment to many resource restricted usages such as usages on always-on devices, battery-powered low-end devices, etc.

Based on the deeply supervised object detection (DSOD) framework, it is proposed Tiny-DSOD dedicating to resource-restricted usages. Tiny-DSOD introduces two innovative and ultra-efficient architecture blocks:

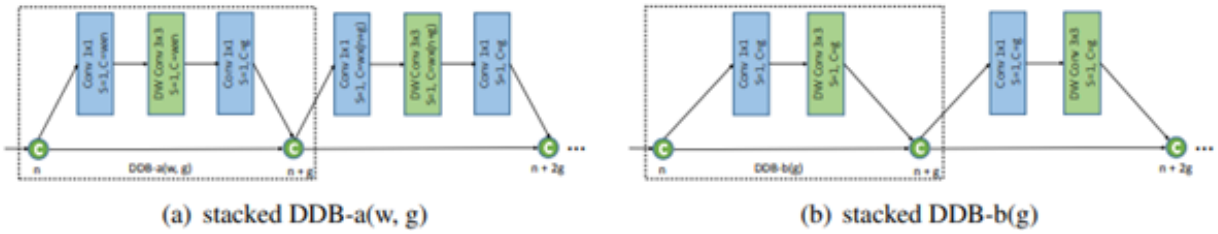
- Depth-wise dense block (DDB) based backbone
- Depth-wise feature-pyramid-network (D-FPN) based front-end.

Tiny-DSOD to the state-of-the-art ultra-efficient object detection solutions such as Tiny-YOLO, MobileNet-SSD (v1 & v2), SqueezeDet, Pelee, etc. Results show that Tiny-DSOD outperforms these solutions in all the three metrics (parameter-size, FLOPs, accuracy) in each comparison. For instance, Tiny-DSOD achieves 72.1% mAP with only 0.95M parameters and 1.06B FLOPs, which is by far the state-of-the-arts result with such a low resource requirement.

Network

The detector is based on the single-shot detector (SSD) [3] framework and the deeply-supervised object detection (DSOD) framework [1], which consists of the backbone part and the front-end part. This will be elaborated these two parts below separately:

Depth-wise Dense Blocks Based Backbone



(a) stacked DDB-a(w, g) (b) stacked DDB-b(g)

Figure 3.1.1: Illustrations of the depth-wise dense blocks (DDB). Two types of DDB are shown in the figure. In the rectangle, "S" means the stride of convolution, and "C" means the number of output channels. Numbers under the concatenating node (green C with circle) means the number of output channels after concatenation. (a) is stacked DDB-a parameterised by growth rate g and expand ratio w. (b) is stacked DDB-b parameterised by growth rate g

Inspired by DSOD [1], they also construct a DenseNet-like backbone since it is easier to be trained from scratch with relatively fewer training sets.

They propose two types of DDB units, DDB-a and DDB-b, as shown in Figure 4.1.1. The DDB-a unit in Figure 4.1.1(a) is inspired by the novel inverted residual blocks proposed in MobileNet-v2. It first expands the input channels w times to w×n, where n is the block input channel number, and w is an integer hyper-parameter to control the model capacity. It then applies the depth-wise convolution, and further projects feature maps to g channels (g is the growth rate of DDB-a) with a point-wise convolution (i.e., 1×1 convolution). Finally, we use concatenation to merge the input and output feature maps together, instead of the residual addition operation in MobileNet-v2. DDB-a has two hyper-parameters w and g, so they denote it as DDB-a(w,g).

There are two main defects of DDB-a. First, suppose L DDB-a blocks are stacked, the complexity of the stacked structure is O(L3g2).

Second, DDB-a concatenates the condensed (aka 1×1 convolution projected) feature maps, so that there are continuous 1×1 convolutions within two adjacent DDB-a units. This kind of processing will introduce potential redundancy among model parameters

With this consideration, they design the other type of depth-wise dense block named DDBb as shown in Figure 1(b). DDB-b first compresses the input channel to the size of growth rate g, and then performs depth-wise convolution. The output of depth-wise convolution is directly concatenated to the input without extra 1× 1 projection. The overall complexity of L stacked DDB-b blocks is O(L2g2), which is smaller than that of DDB-a.

Module name		Output size	Component
Stem	Convolution	64 × 150 × 150	3 × 3 conv, stride 2
	Convolution	64 × 150 × 150	1 × 1 conv, stride 1
	Depth-wise convolution	64 × 150 × 150	3 × 3 dwconv, stride 1
	Convolution	128 × 150 × 150	1 × 1 conv, stride 1
	Depth-wise convolution	128 × 150 × 150	3 × 3 dwconv, stride 1
	Pooling	128 × 75 × 75	2 × 2 max pool, stride 2
Extractor	Dense stage 0	256 × 75 × 75	DDB-b(32) * 4
	Transition layer 0	128 × 38 × 38	1 × 1 conv, stride 1 2 × 2 max pool, stride 2
	Dense stage 1	416 × 38 × 38	DDB-b(48) * 6
	Transition layer 1	128 × 19 × 19	1 × 1 conv, stride 1 2 × 2 max pool, stride 2
	Dense stage 2	512 × 19 × 19	DDB-b(64) * 6
	Transition layer 2	256 × 19 × 19	1 × 1 conv, stride 1
	Dense stage 3	736 × 19 × 19	DDB-b(80) * 6
	Transition layer 3	64 × 19 × 19	1 × 1 conv, stride 1

Table 3.1.1: Tiny-DSOD backbone architecture (input size 3 × 300 × 300). In the "Component" column, the symbol "*" after block names indicates that block repeats number times given after the symbol.s

Depth-wise FPN based Front-end

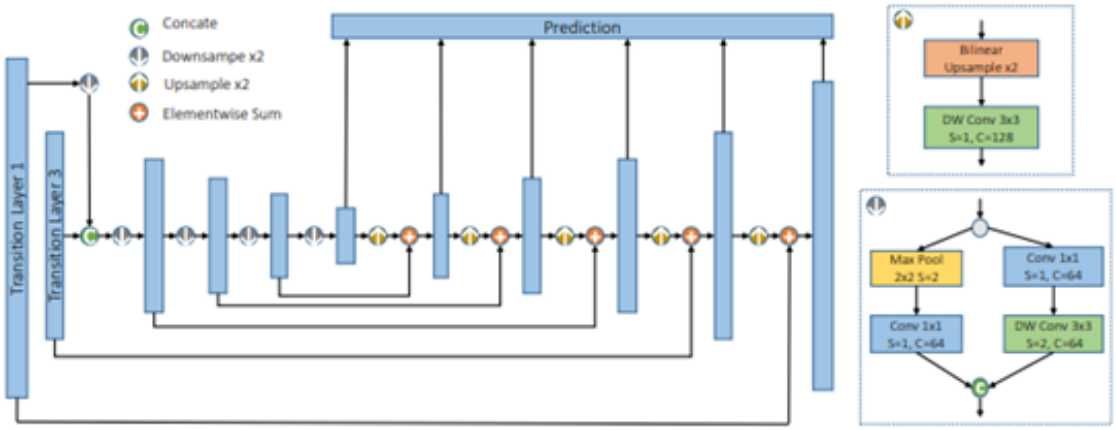


Figure 3.1.2: Illustrations of the D-FPN structure. The left part is the over structure of D-FPN, while the right part further depicts the details of the up-sampling (top-right) and down-sampling (bottom right) modules in D-FPN. Note both sampling are by factor 2, "S" is the stride of convolution, and "C" is the number of output channels

design a lightweight FPN named depth-wise FPN (DFPN) in our predictor to redirect the information flow from deeper and smaller feature maps to shallower ones. Figure 4.1.2 illustrates the structure of our front-end predictor, which consists of a downsampling path and a reverse upsampling path.

However, most of these works implement the reverse-path via deconvolution operations, which greatly increases the model complexity. To avoid this problem, they propose a cost-efficient solution for the reverse path. As shown in top-right of Figure 4.1.2, they up-sample the top feature

maps with a simple bilinear interpolation layer followed by a depth-wise convolution, this operation could be formulated as Equation 1.

$$F_c(x,y) = W_c * \sum_{(m,n)\in\Omega} U_c(m,n)\tau(m,sx)\tau(n,sy)$$

Where Fc is the c-th channel of output feature map and Uc is the corresponding channel of input. Wc is the c-th kernel of depth-wise convolution and '∗ ' denotes the spatial convolution. Ω is the co-ordinate set of input features and s is the resampling coefficient in this layer. τ(a, b) = max(0,1−| a−b|) is the differentiable bilinear operator. The resulting feature maps are merged with the same-sized feature map in the bottom layer via element-wise addition. Our experiment in section 4.2 will show that D-FPN can achieve a considerable detection accuracy boost, with slight increase of computation cost.

Results

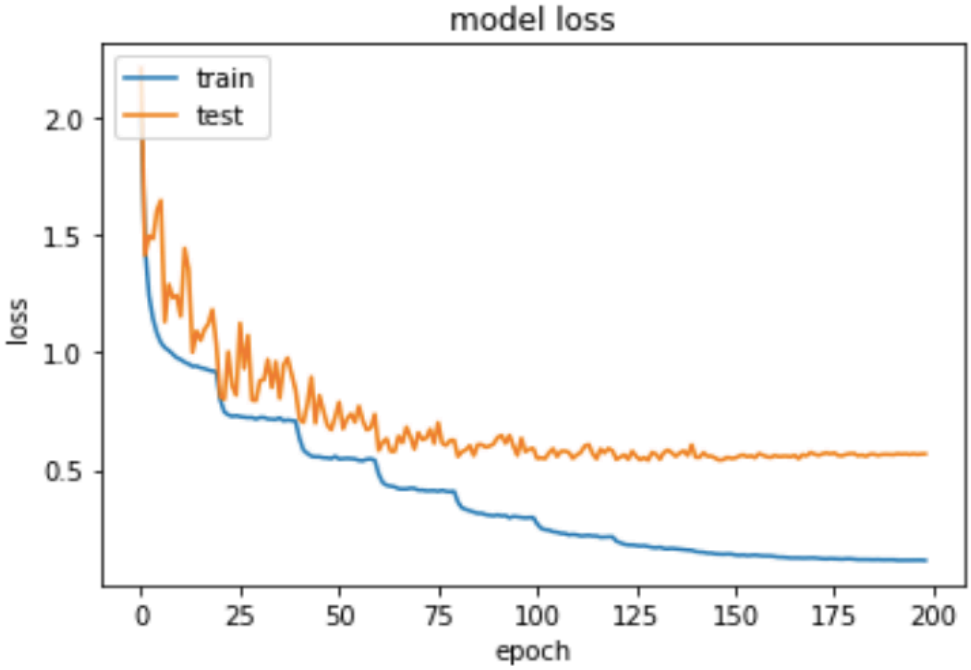


Fig 3.1.3: Model train and Test accuracy and Loss

```
391/391 [==============================] - 55s 142ms/step - loss: 0.1191 - acc: 0.9934 - val_loss: 0.5639 - val_acc: 0.8902
Epoch 184/300
391/391 [==============================] - 55s 142ms/step - loss: 0.1189 - acc: 0.9937 - val_loss: 0.5577 - val_acc: 0.8920
Epoch 185/300
391/391 [==============================] - 56s 142ms/step - loss: 0.1185 - acc: 0.9940 - val_loss: 0.5666 - val_acc: 0.8909
Epoch 186/300
391/391 [==============================] - 55s 142ms/step - loss: 0.1181 - acc: 0.9938 - val_loss: 0.5672 - val_acc: 0.8913
Epoch 187/300
391/391 [==============================] - 54s 139ms/step - loss: 0.1188 - acc: 0.9934 - val_loss: 0.5616 - val_acc: 0.8912
Epoch 188/300
391/391 [==============================] - 55s 140ms/step - loss: 0.1177 - acc: 0.9940 - val_loss: 0.5653 - val_acc: 0.8911
Epoch 189/300
391/391 [==============================] - 55s 140ms/step - loss: 0.1180 - acc: 0.9936 - val_loss: 0.5661 - val_acc: 0.8917
Epoch 190/300
391/391 [==============================] - 55s 141ms/step - loss: 0.1172 - acc: 0.9942 - val_loss: 0.5653 - val_acc: 0.8907
Epoch 191/300
391/391 [==============================] - 55s 141ms/step - loss: 0.1179 - acc: 0.9937 - val_loss: 0.5647 - val_acc: 0.8907
Epoch 192/300
391/391 [==============================] - 55s 142ms/step - loss: 0.1158 - acc: 0.9941 - val_loss: 0.5687 - val_acc: 0.8904
Epoch 193/300
391/391 [==============================] - 55s 141ms/step - loss: 0.1159 - acc: 0.9945 - val_loss: 0.5665 - val_acc: 0.8902
Epoch 194/300
391/391 [==============================] - 55s 141ms/step - loss: 0.1165 - acc: 0.9941 - val_loss: 0.5686 - val_acc: 0.8906
Epoch 195/300
391/391 [==============================] - 55s 141ms/step - loss: 0.1165 - acc: 0.9937 - val_loss: 0.5652 - val_acc: 0.8914
Epoch 196/300
391/391 [==============================] - 55s 141ms/step - loss: 0.1161 - acc: 0.9942 - val_loss: 0.5689 - val_acc: 0.8912
Epoch 197/300
391/391 [==============================] - 55s 142ms/step - loss: 0.1158 - acc: 0.9945 - val_loss: 0.5653 - val_acc: 0.8904
Epoch 198/300
391/391 [==============================] - 56s 142ms/step - loss: 0.1164 - acc: 0.9940 - val_loss: 0.5678 - val_acc: 0.8894
Epoch 199/300
391/391 [==============================] - 55s 142ms/step - loss: 0.1153 - acc: 0.9945 - val_loss: 0.5681 - val_acc: 0.8907
Epoch 00199: early stopping
```

Fig 3.1.4:Training epochs and loss

- The training stopped at 199 epochs with early stopping.
- Learning rate =0.1, Learning rate drop =20.
- Nesterov SGD used with momentum 0.9.
- Validation Accuracy at last epoch is 89%

3.2.2 Mobile Nets v1

It presents a class of efficient models called MobileNets for mobile and embedded vision applications. MobileNets are based on a streamlined architecture that uses depth wise separable convolutions to build light weight deep neural networks. It uses two simple global hyper-parameters that efficiently tradeoff between latency and accuracy. These hyper-parameters allow the model builder to choose the right sized model for their application based on the constraints of the problem.

The MobileNet structure is built on depth-wise separable convolutions except for the first layer which is a full convolution. By defining the network in such simple terms we are able to easily explore network topologies to find a good network. The MobileNet architecture is defined below. All layers are followed by a batch-norm [2] and ReLU nonlinearity with the exception of the final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification. Figure 4.2.1 contrasts a layer with regular convolutions, batch-norm and ReLU nonlinearity to the factorised layer with depth-wise convolution, 1 × 1 point-wise convolution as well as batch-norm and ReLU after each convolutional layer. Down sampling is handled with strided convolution in the depth-wise convolutions as well as in the first layer. A final average pooling reduces the spatial resolution to 1 before the fully connected layer. Counting depth-wise and point-wise convolutions as separate layers, MobileNet has 28 layers.

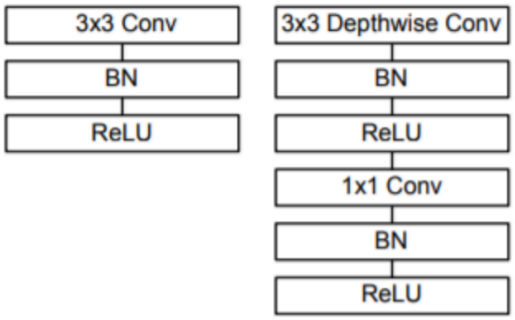


Figure 3.2.1. Left: Standard convolutional layer with batch-norm and ReLU. Right: Depth-wise Separable convolutions with Depth-wise and Point-wise layers followed by batch-norm and ReLU.

Architecture

In this section we first describe the core layers that MobileNet is built on which are depth-wise separable filters. We then describe the MobileNet network structure and conclude with descriptions of the two model shrinking hyper-parameters width multiplier and resolution multiplier.

Depth-wise separable convolutions

Depth-wise separable convolutions are made up of two layers: depth-wise convolutions and point-wise convolutions. We use depth-wise convolutions to apply a single filter per each input channel (input depth). Point-wise convolution, a simple 1×1 convolution, is then used to create a linear combination of the output of the depth-wise layer. MobileNets use both batch-norm and ReLU nonlinearities for both layers. Depth-wise convolution with one filter per input channel (input depth) can be written as:

$$\hat{\mathbf{G}}_{k,l,m} = \sum_{i,j} \hat{\mathbf{K}}_{i,j,m} \cdot \mathbf{F}_{k+i-1,l+j-1,m}$$

where Kˆ is the depth-wise convolution kernel of size DK × DK × M where the m^{th} filter in Kˆ is applied to the mth channel in F to produce the mth channel of the filtered output feature map Gˆ . Depth-wise convolution has a computational cost of:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$$

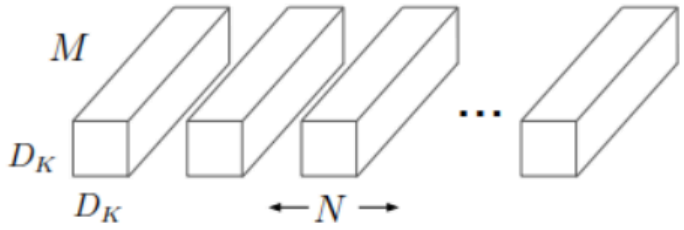
Depth-wise convolution is extremely efficient relative to standard convolution. However it only filters input channels, it does not combine them to create new features. So an additional layer that computes a linear combination of the output of depth-wise convolution via 1 × 1 convolution is needed in order to generate these new features. The combination of depth-wise convolution and 1 × 1 (point-wise) convolution is called depth-wise separable convolution which was originally introduced in [2]. Depth-wise separable convolutions cost:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

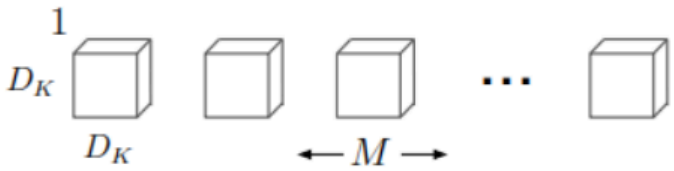
which is the sum of the depth-wise and 1 × 1 point-wise convolution. By expressing convolution as a two step process of filtering and combining we get a reduction in computation of:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

MobileNet uses 3 × 3 depth wise separable convolutions which uses between 8 to 9 times less computation than standard convolutions at only a small reduction in accuracy.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

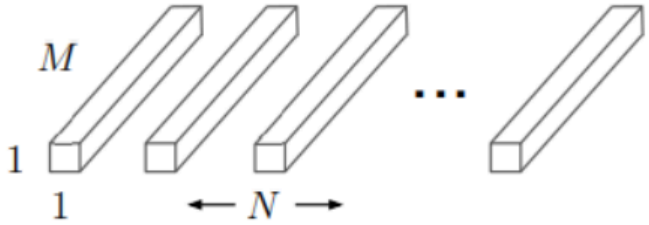


Figure 3.2.2. The standard convolutional filters in (a) are replaced by two layers: depth-wise convolution in (b) and point-wise convolution in (c) to build a depth-wise separable filter.

Width Multiplier: Thinner Models

The role of the width multiplier α is too thin a network uniformly at each layer. For a given layer and width multiplier α, the number of input channels M becomes αM and the number of output channels N becomes αN. The computational cost of a depth-wise separable convolution with width multiplier α is:

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F$$

where α ∈ (0, 1] with typical settings of 1, 0.75, 0.5 and 0.25. α = 1 is the baseline MobileNet and α < 1 are reduced MobileNets.

Resolution Multiplier: Reduced Representation

The second hyper-parameter to reduce the computational cost of a neural network is a resolution multiplier ρ. We apply this to the input image and the internal representation of every layer is subsequently reduced by the same multiplier. In practice we implicitly set ρ by setting the input resolution. We can now express the computational cost for the core layers of our network as depth-wise separable convolutions with width multiplier α and resolution multiplier ρ:

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$$

where ρ ∈ (0, 1] which is typically set implicitly so that the input resolution of the network is 224, 192, 160 or 128. ρ = 1 is the baseline MobileNet and ρ < 1 is a reduced computation MobileNets. Resolution multiplier has the effect of reducing computational cost by ρ 2

Network and parameters

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 32)        896
_________________________________________________________________
depthwise_conv2d_1 (Depthwis (None, 16, 16, 32)        9248
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        2112
_________________________________________________________________
depthwise_conv2d_2 (Depthwis (None, 16, 16, 64)        36928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 128)       8320
_________________________________________________________________
depthwise_conv2d_3 (Depthwis (None, 8, 8, 128)         147584
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 128)         16512
_________________________________________________________________
depthwise_conv2d_4 (Depthwis (None, 8, 8, 128)         147584
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 128)         16512
_________________________________________________________________
global_average_pooling2d_1 ( (None, 128)               0
_________________________________________________________________
batch_normalization_1 (Batch (None, 128)               512
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 404,522
Trainable params: 404,010
Non-trainable params: 512
_________________________________________________________________
```

Figure 3.2.3. Model architecture for Simple MobileNet

- Only 404k Parameters
- Trained for 50 epochs on Cifar-10 dataset to obtain 75% accuracy
- Lightweight and quick to train

```
50000/50000 [==============================] - 60s 1ms/step - loss: 0.5430 - acc: 0.8200
Epoch 37/50
50000/50000 [==============================] - 61s 1ms/step - loss: 0.5359 - acc: 0.8201
Epoch 38/50
50000/50000 [==============================] - 60s 1ms/step - loss: 0.5334 - acc: 0.8227
Epoch 39/50
50000/50000 [==============================] - 60s 1ms/step - loss: 0.5151 - acc: 0.8291
Epoch 40/50
50000/50000 [==============================] - 59s 1ms/step - loss: 0.5083 - acc: 0.8296
Epoch 41/50
50000/50000 [==============================] - 60s 1ms/step - loss: 0.5031 - acc: 0.8303
Epoch 42/50
50000/50000 [==============================] - 60s 1ms/step - loss: 0.4965 - acc: 0.8344
Epoch 43/50
50000/50000 [==============================] - 59s 1ms/step - loss: 0.4891 - acc: 0.8361
Epoch 44/50
50000/50000 [==============================] - 59s 1ms/step - loss: 0.4746 - acc: 0.8389
Epoch 45/50
50000/50000 [==============================] - 59s 1ms/step - loss: 0.4704 - acc: 0.8430
Epoch 46/50
50000/50000 [==============================] - 59s 1ms/step - loss: 0.4663 - acc: 0.8442
Epoch 47/50
50000/50000 [==============================] - 61s 1ms/step - loss: 0.4566 - acc: 0.8470
Epoch 48/50
50000/50000 [==============================] - 62s 1ms/step - loss: 0.4448 - acc: 0.8509
Epoch 49/50
50000/50000 [==============================] - 62s 1ms/step - loss: 0.4411 - acc: 0.8514
Epoch 50/50
50000/50000 [==============================] - 61s 1ms/step - loss: 0.4286 - acc: 0.8571
```

```
[23]  1 Simple_MobileNetV1_model.save('Simple_MobileNetV1_model2.h5')
```

```
[24]  1 Simple_MobileNetV1_model.evaluate(x_test,y_test)
```

```
10000/10000 [==============================] - 3s 318us/step
[0.82435347738266, 0.7552]
```

Fig 3.2.4: Training , loss and accuracy on test set

Simple Naive Conv Net

```
[12]  1 Simple_NaiveConvNet_model = Simple_NaiveConvNet((32,32,3), 10)
      2 Simple_NaiveConvNet_model.summary()
      3 #Simple_NaiveConvNet_model.compile(optimizer=optimizers.RMSprop(0.001), loss='categorical_crossentropy', metrics=['acc'])
      4 #Simple_NaiveConvNet_history = Simple_NaiveConvNet_model.fit(x_train, y_train, epochs=50, batch_size=8)
      5 sgd = SGD(lr=0.01, momentum=0.9, decay=1e-6, nesterov=True)
      6 Simple_NaiveConvNet_model.compile(optimizer = sgd, loss = "categorical_crossentropy", metrics=['accuracy'])
      7
      8 Simple_NaiConNetV1_history = Simple_NaiveConvNet_model.fit(x_train,y_train, epochs=50, batch_size=8)
```

```
Model: "model_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 16, 16, 32)        896
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 16, 16, 128)       73856
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
global_average_pooling2d_6 ( (None, 128)               0
_________________________________________________________________
batch_normalization_11 (Batc (None, 128)               512
_________________________________________________________________
dense_11 (Dense)             (None, 128)               16512
_________________________________________________________________
batch_normalization_12 (Batc (None, 128)               512
_________________________________________________________________
dense_12 (Dense)             (None, 10)                1290
=================================================================
Total params: 259,658
Trainable params: 259,146
Non-trainable params: 512
```

Fig 3.2.5: Simple Naive Conv Net Architecture

```
50000/50000 [==============================] - 77s 2ms/step - loss: 0.6383 - acc: 0.7783
Epoch 41/50
50000/50000 [==============================] - 77s 2ms/step - loss: 0.4506 - acc: 0.8430
Epoch 42/50
50000/50000 [==============================] - 79s 2ms/step - loss: 0.4109 - acc: 0.8587
Epoch 43/50
50000/50000 [==============================] - 78s 2ms/step - loss: 0.3713 - acc: 0.8719
Epoch 44/50
50000/50000 [==============================] - 77s 2ms/step - loss: 0.3545 - acc: 0.8756
Epoch 45/50
50000/50000 [==============================] - 76s 2ms/step - loss: 0.3188 - acc: 0.8899
Epoch 46/50
50000/50000 [==============================] - 77s 2ms/step - loss: 0.2863 - acc: 0.9014
Epoch 47/50
50000/50000 [==============================] - 76s 2ms/step - loss: 0.4364 - acc: 0.8527
Epoch 48/50
50000/50000 [==============================] - 78s 2ms/step - loss: 0.3163 - acc: 0.8905
Epoch 49/50
50000/50000 [==============================] - 79s 2ms/step - loss: 0.3147 - acc: 0.8900
Epoch 50/50
50000/50000 [==============================] - 79s 2ms/step - loss: 0.3069 - acc: 0.8931
```

```
[13]  1 Simple_NaiveConvNet_model.save('Simple_NaiveConvNet_model2.h5')
```

```
1 Simple_NaiveConvNet_model.evaluate(x_test,y_test)
```

```
10000/10000 [==============================] - 2s 164us/step
[0.8257833055496215, 0.7434]
```

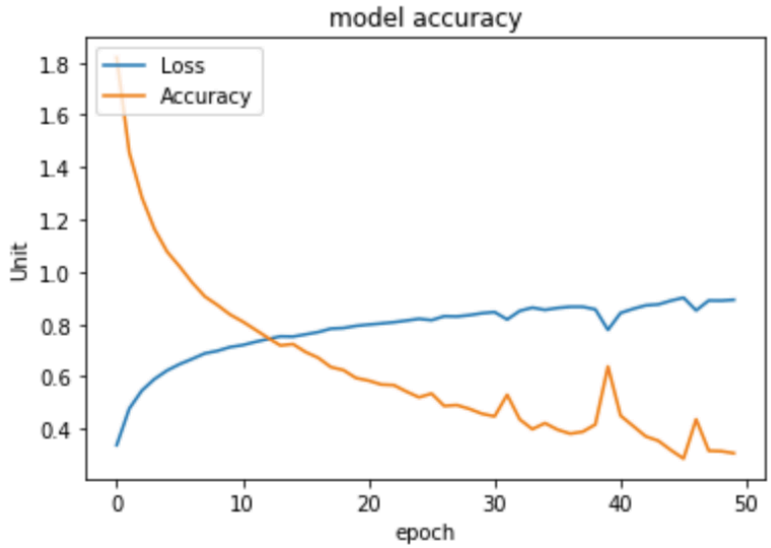


Fig 3.2.6: Simple Naive Conv Net Training and Model accuracy

- Ran for 50 epochs ~ 1 hour
- Nesterov SGD used with learning rate =0.01
- Accuracy obtained 74.34% with just 259k parameters on CIFAR-10.

3.2.3 SSD

By using SSD, we only need to **take one single shot to detect multiple objects within the image**, while regional proposal network (RPN) based approaches such as R-CNN series that need two shots, one for generating region proposals, one for detecting the object of each proposal. Thus, SSD is much faster compared with two-shot RPN-based approaches [3].

- **Single Shot:** this means that the tasks of object localisation and classification a re done in a *single forward pass* of the network.
- **MultiBox:** this is the name of a technique for bounding box regression developed by Szegedy et al. (we will briefly cover it shortly).
- **Detector:** The network is an object detector that also classifies those detected objects.

Network

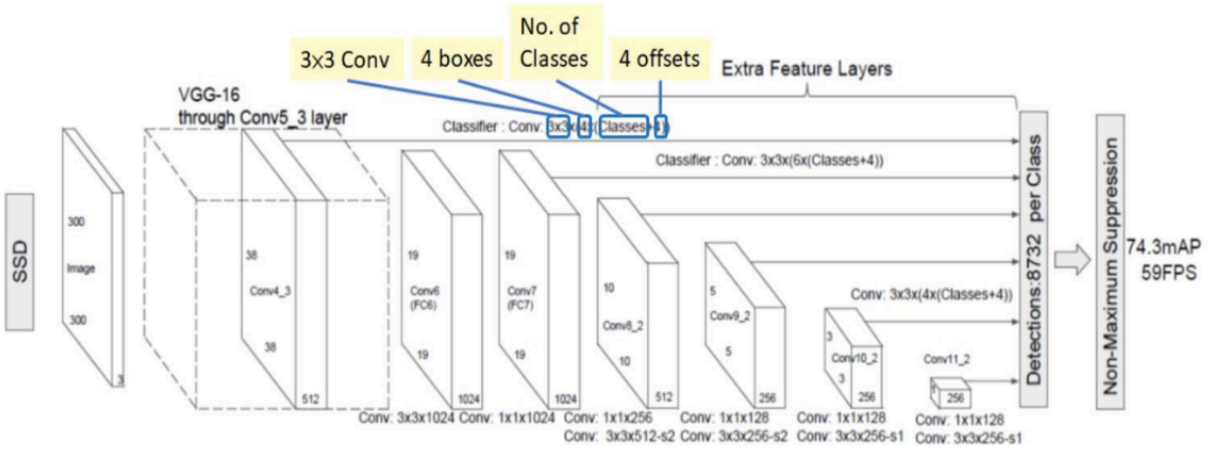


Fig 3.3.1: VGG-16 Network

If we sum them up we get 5776 + 2166 + 600 + 150 + 36 + 4 = 8732 boxes in total

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	$1 \times 1 \times 1000$

Table 3.3.1: MobileNet v1 Architecture details

Training details

Hard Negative Mining

Instead of using all the negative examples, we sort them using the highest confidence loss for each default box and pick the top ones so that the ratio between the negatives and positives is at most 3:1.

Data Augmentation

Each training image is randomly sampled by:

- Entire original input image
- Sample a patch so that the overlap with objects is 0.1, 0.3, 0.5, 0.7 or 0.9.
- Randomly sample a patch

Result

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Table 3.3.2: Result comparison with other networks

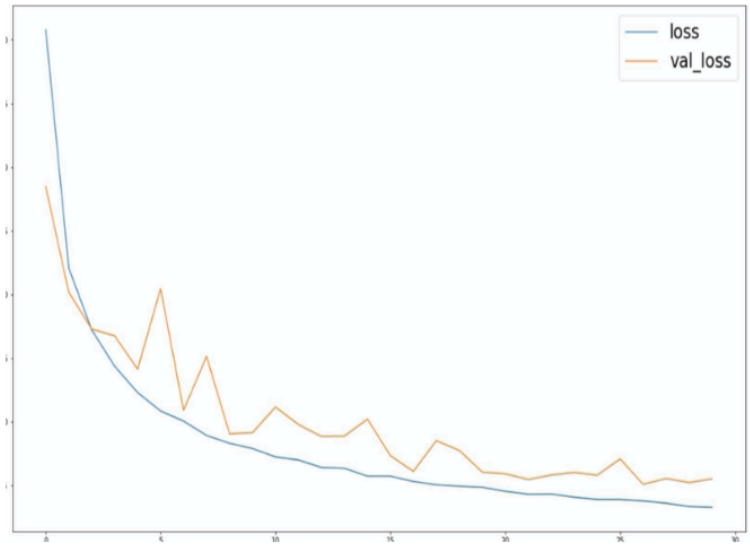


Fig 3.3.2: Loss vs Validation loss curve for training

3.2.4 YOLO V2

YOLOv2 became more accurate and faster than the previous version (YOLO). This is because YOLOv2 uses some techniques that YOLO didn't use, such as Batch-Normalization and Anchor-Boxes.

Batch-Normalization or BN is used to normalise the outputs of hidden layers. This makes learning much faster. Anchor-Boxes are assumptions on the shapes of the bounding boxes. Since the shapes of objects we are trying to detect do not vary so much, we don't have to find boxes that do not look like any of the objects we want to detect.

Let us say we want to detect humans, then the shapes of anchor boxes are usually vertical rectangles and it's less likely that they are squares or horizontal rectangles. So we don't have to search such boxes. This makes prediction much faster as compared to YOLO.

This model heavily depends on dark-flow, which is an object detection API based on YOLO. It is used to train YOLOv2 networks on custom objects or training data. This API's prediction runs very fast, because some time consuming part of the program are written in python.

Network Architecture

The input to the network is 416x416x3 image in YOLOv2. There is no fully connected layer in it.

Layer description	Layer output	
Network Input	416x416x3	
Convolutional, size=3, stride=1, pad=1, filters=16, bn=1	416x416x16	
Leaky ReLu	416x416x16	
Maxpool, size=2, stride=2	208x208x16	
Convolutional, size=3, stride=1, pad=1, filters=32, bn=1	208x208x32	
Leaky ReLu	208x208x32	
Maxpool, size=2, stride=2	104x104x32	
Convolutional, size=3, stride=1, pad=1, filters=64, bn=1	104x104x64	
Leaky ReLu	104x104x64	
Maxpool, size=2, stride=2	52x52x64	
Convolutional, size=3, stride=1, pad=1, filters=128, bn=1	52x52x128	
Leaky ReLu	52x52x128	
Maxpool, size=2, stride=2	26x26x128	
Convolutional, size=3, stride=1, pad=1, filters=256, bn=1	26x26x256	
Leaky ReLu	26x26x256	
Maxpool, size=2, stride=2	13x13x256	
Convolutional, size=3, stride=1, pad=1, filters=512, bn=1	13x13x512	
Leaky ReLu	13x13x512	
Maxpool, size=2, stride=1	13x13x512	
Convolutional, size=3, stride=1, pad=1, filters=1024, bn=1	13x13x1024	
Leaky ReLu	13x13x1024	
Convolutional, size=3, stride=1, pad=1, filters=B(5+C), bn=0	13x13xB(5+C)	← Network Output

Table 3.4.1: Layer description for YOLOv2

Anchor-Boxes for more accuracy

In YOLO, which is the previous version of YOLOv2, the prediction of the box shape was done randomly. That is, the network did not know what shapes of bounding boxes are most likely to detect an object of a certain class[4]. For example, when the network tries to detect humans, it searches for humans with square bounding box and vertical rectangle equally likely. However, in the real world, humans usually fit more in vertical rectangles than square boxes. So, the network should search humans with vertical rectangles. This is the motivation of using Anchor-Boxes.

IOU

This is short for Intersection Over Union. This is calculated by dividing the overlapped area of a predicted box and the truth box by the whole area made by the two boxes[4]. This value is used as a measure of how good the prediction is.

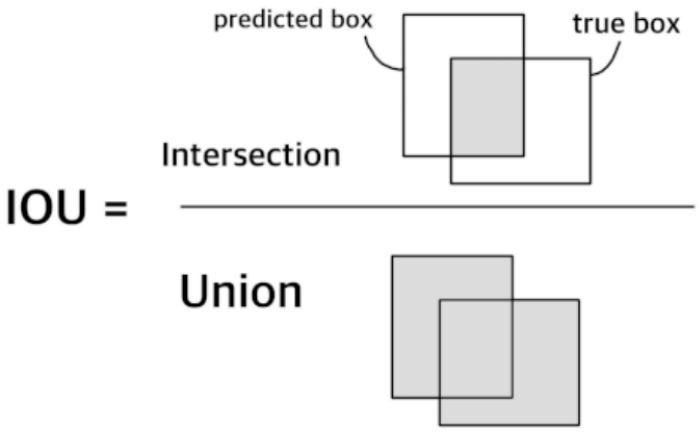


Fig 3.4.1: Intersection of Union explanation

Loss Function composed of 3 losses

Since the architecture and the output shape differ from the one used in YOLO, the loss function is also slightly different. Even in the paper of YOLOv2, the loss function is not given as a mathematical formula. So, I decided to extract the loss function from an implementation of YOLOv2 called dark-flow. dark-flow is a machine learning API which can be used to train YOLO and YOLOv2. The loss function is implemented with TensorFlow in this file

The prediction of YOLOv2 is composed of three parts:

- The four coordinates for x, y, w and h
- The probability P(obj) that an object exists in a bounding box
- The conditional probability C_i = P (the obj belongs i^{th} class | and obj exists in this box)

Therefore, the loss is calculated for each prediction and then combined.

Loss for x, y, w and h

The coordinates of x and y lie between 0 and 1. This is because they are given by applying sigmoid on the x and y part of the output from the network (at least in dark-flow). This prediction is done relative to the grid cell. For example, if both x and y are 0.5, then this means that the box's centre falls on the centre of the grid cell. The reason why the centre coordinates are predicted this way is just we don't need to know the absolute coordinates if we know this grid cell position.
Then the loss is defined as a sum of squares of error:

$$\sum_{i=0}^{13^2-1} \sum_{j=0}^{B-1} 1_{i,j}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2]$$

The sum is over all grid cells and all bounding boxes. 1_{i,j}^{obj} is 1.0 if the bounding box is 'responsible' for detecting this object, 0.0 otherwise. The term 'responsible' means that this bounding box has the highest IOU value among the B bounding boxes, and there is actually an object on this grid cell.

Output of the Model

In object detection, we also must predict the location and the shape of an object, not only classification. Therefore, the output of an object detection network becomes a little bit complicated. In our case of YOLOv2 (YOLO), the output is a 3-dimensional array (or Tensor in TensorFlow). Particularly in YOLOv2, the shape of output is 13x13xD, where D varies depending on how many classes of object we want to detect (D=5 for single class). The first 2-dimensional array (13x13) is called grid cells. So, there are 169 grid cells in total.

One grid cell is 'responsible' for detecting 5 bounding boxes, that is why we can detect up to 5 boxes on a grid cell[4]. This means that the network can detect up to 169 x 5 = 845 boxes at once. This number of bounding boxes a grid cell can detect is the number of Anchor-Boxes we prepare,

and we can change this number to whatever we want. So, for example, if we want to detect humans and cars and think that just two Anchor-Boxes (vertical rectangle for humans, and horizontal rectangle for cars) are enough to detect them, then the number 5 above becomes 2. (In the paper of YOLOv2, this number is denoted as 'B')

3.2.5 SqueezeDet

Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection

Lightweight

SqueezeDet's architecture consists of approximately 2 million trainable parameters, in comparison with ResNet-50 with 25 million parameters and VGG19 with 143 million, SqueezeDet is extremely lightweight and has very small size.

Fully convolutional

A fully connected layer each neuron has a connection with a unique weight to each input. In convolutional layers, a neuron (filter), slides over the input, only processing a small part at a time, but with the same weights.

Network

SqueezeDet is built on SqueezeNet architecture

- Using 1x1(point-wise) filters to replace 3x3 filters, as the former only 1/9 of computation.
- Using 1x1 filters as a bottleneck layer to reduce depth to reduce computation of the following 3x3 filters.
- Downsample late to keep a big feature map.

SqueezeNet is built upon *Fire Module*, which is comprised of a *squeeze* layer as input, and two parallel *expand* layers as output. The *squeeze* layer is a 1x1 convolutional layer that compresses an input tensor with large channel size to one with the same batch and spatial dimension, but smaller channel size. The *expand* layer is a mixture of 1x1 and 3x3 convolution filters that takes the compressed tensor as input, retrieve the rich features and output an activation tensor with large

channel size. The alternating *squeeze* and *expand* layers effectively reduces parameter size without losing too much accuracy.

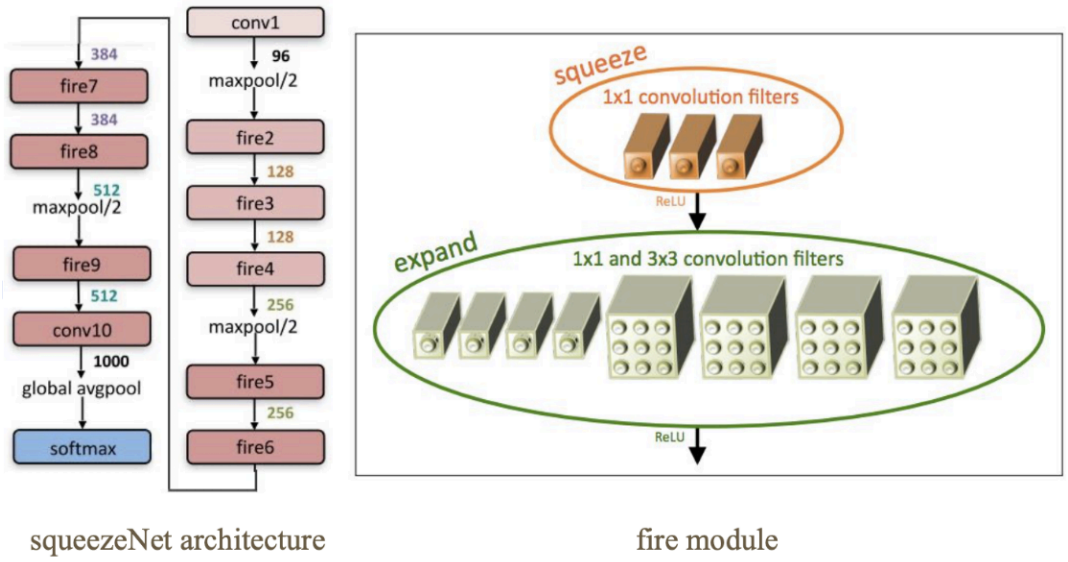


Fig 3.5.1 squeezeNet architecture

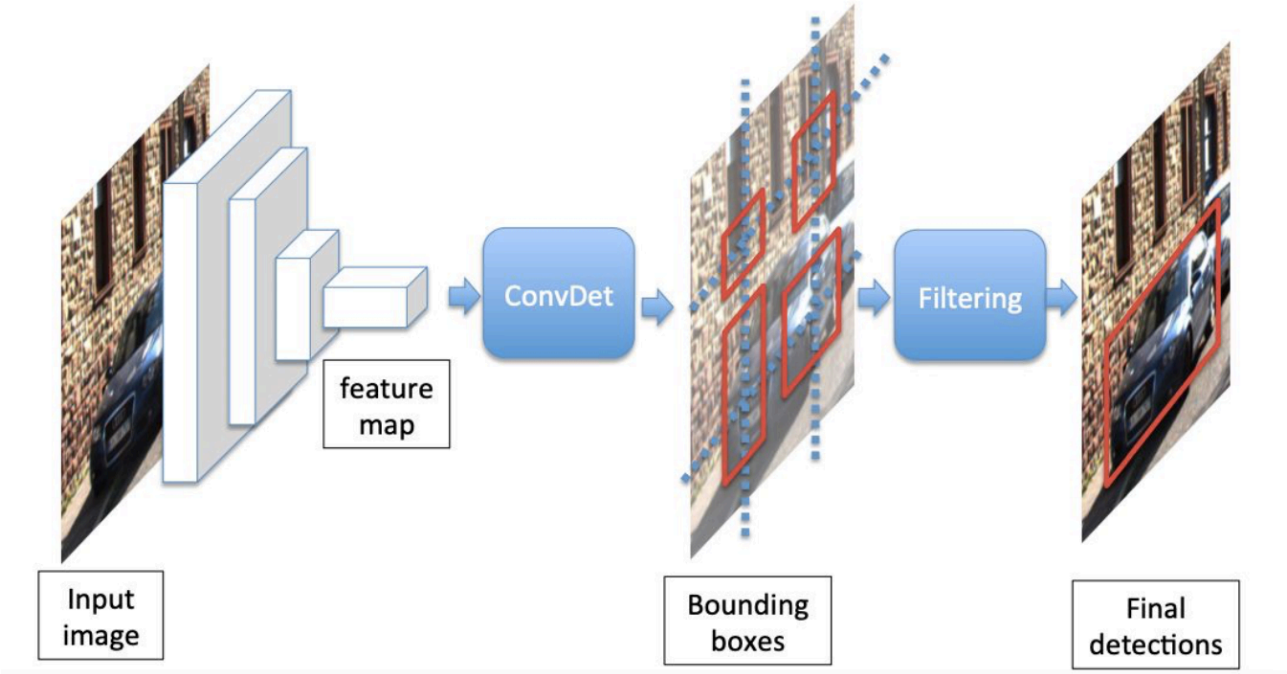


Fig 3.5.2: Flow of input

ConvDet

ConvDet layer enables SqueezeDet to generate tens-of-thousands of region proposals with much fewer model parameters. It's a convolutional layer that is trained to output bounding box coordinates and class probabilities. *ConvDet* is essentially a convolutional layer that is trained to output bounding box coordinates and class probabilities. It works as a sliding window that moves through each spatial position on the feature map.

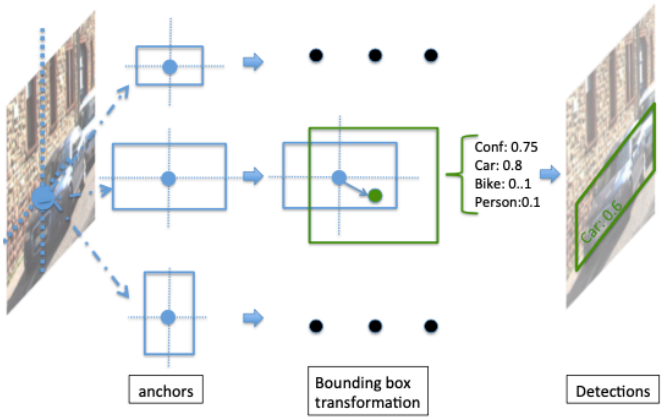


Fig 3.5.3: Bounding box transformation. Each grid centre has K anchors with pre-selected shapes. Each anchor is transformed to its new position and shape using the relative coordinates computed by the *ConvDet* layer. Each anchor is associated with a confidence score and class probabilities to predict the category of the object within the bounding box.

Here, K is the number of reference bounding boxes with preselected shapes. At each position, it computes K × (4 + 1 + C) values that encode the bounding box predictions.

Training

- Deploys a (4-step) alternating training strategy to train RPN and detector network.
- **Multi-task loss function** : To train the ConvDet layer to learn detection, localization and classification.

Bounding box regression $(\delta x_{ijk}, \delta y_{ijk}, \delta w_{ijk}, \delta h_{ijk})$ corresponds to the relative coordinates of anchor-k located at grid centre-(i, j).

Confidence score regression (using o/p of ConvDet)

$$\frac{\lambda_{bbox}}{N_{obj}} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} I_{ijk}[(\delta x_{ijk} - \delta x_{ijk}^{G})^2 + (\delta y_{ijk} - \delta y_{ijk}^{G})^2 + (\delta w_{ijk} - \delta w_{ijk}^{G})^2 + (\delta h_{ijk} - \delta h_{ijk}^{G})^2]$$

$$+ \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \frac{\lambda_{conf}^{+}}{N_{obj}} I_{ijk}(\gamma_{ijk} - \gamma_{ijk}^{G})^2 + \frac{\lambda_{conf}^{-}}{WHK - N_{obj}} \bar{I}_{ijk} \gamma_{ijk}^{2}$$

$$+ \frac{1}{N_{obj}} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{k=1}^{K} \sum_{c=1}^{C} I_{ijk} l_{c}^{G} \log(p_c).$$

The ground truth bounding box is computed as,

$$\delta x_{ijk}^{G} = (x^{G} - \hat{x}_i)/\hat{w}_k,$$
$$\delta y_{ijk}^{G} = (y^{G} - \hat{y}_j)/\hat{h}_k,$$
$$\delta w_{ijk}^{G} = \log(w^{G}/\hat{w}_k),$$
$$\delta h_{ijk}^{G} = \log(h^{G}/\hat{h}_k).$$

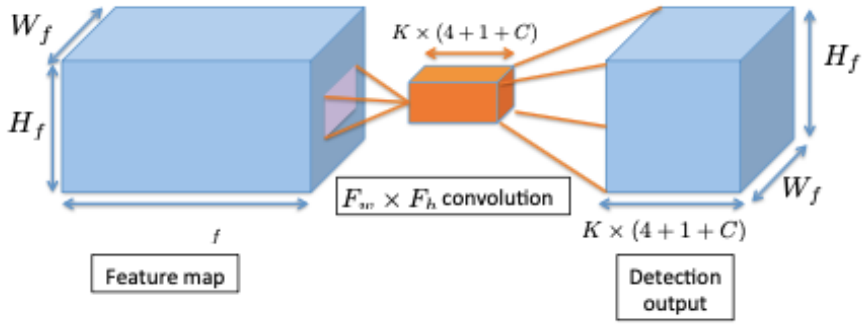


Fig 3.5.4: The *ConvDet* layer is a $F_w \times F_h$ convolution with output size of K × (5 + C). It's responsible for both computing bounding boxes and classifying the object within. The parameter size for this layer is $F_w F_h Ch_f K(5 + C)$.

Model size

The SqueezeDet model is 61X smaller than the *Faster R-CNN + VGG16* model, and it is 30X smaller than the *Faster R-CNN + AlexNet* model. Al- most 80% of the parameters of the VGG16 model are from the fully connected layers. Thus, after we replace the fully connected layers and RPN layer with *ConvDet*, the model size is only 57.4MB. Compared with YOLO which is comprised of 24 convolutional layers, two fully connected layers with a model size of 753MB, SqueezeDet, without any compression, is 95X smaller.

Comparison

Method	Car mAP	Cyclist mAP	Pedestrian mAP	All mAP	Model size (MB)	Speed (FPS)
FRCN + VGG16[2]	86.0	-	-	-	485	1.7
FRCN + AlexNet[2]	82.6	-	-	-	240	2.9
SqueezeDet	82.9	76.8	70.4	76.7	**7.9**	**57.2**
SqueezeDet+	85.5	**82.0**	**73.7**	**80.4**	26.8	32.1
VGG16-Det	**86.9**	79.6	70.7	79.1	57.4	16.6
ResNet50-Det	86.7	80.0	61.5	76.1	35.1	22.5

Table 3.5.1: Comparison of SqueezeDet

3.2.6 GenLR-Net

The VGG architecture is shown by the shaded portion of the network. This architecture is for classification and has 16 trainable layers including convolutional and fully connected layers. The average prediction log-loss after the softmax layer is employed during the training to minimise the classification error. The goal is to verify whether a pair of LR and HR image belongs to the same subject or not. Thus the HR and LR images belonging to the same subjects to come closer to one another and those from different subjects to move apart. Since there are two images of different resolutions, the network has two channels, where one channel takes the input as HR image and the other channel takes the input as the LR image. The HR channel is kept fixed (the shaded part indicates that the weights are locked and not updated during training) since the network has already been trained for computing discriminative features from HR images. The modification is that the final classification layer of the VGG network is replaced with a contrastive layer for the higher level feature, i.e. the final fully connected layer (f c7).

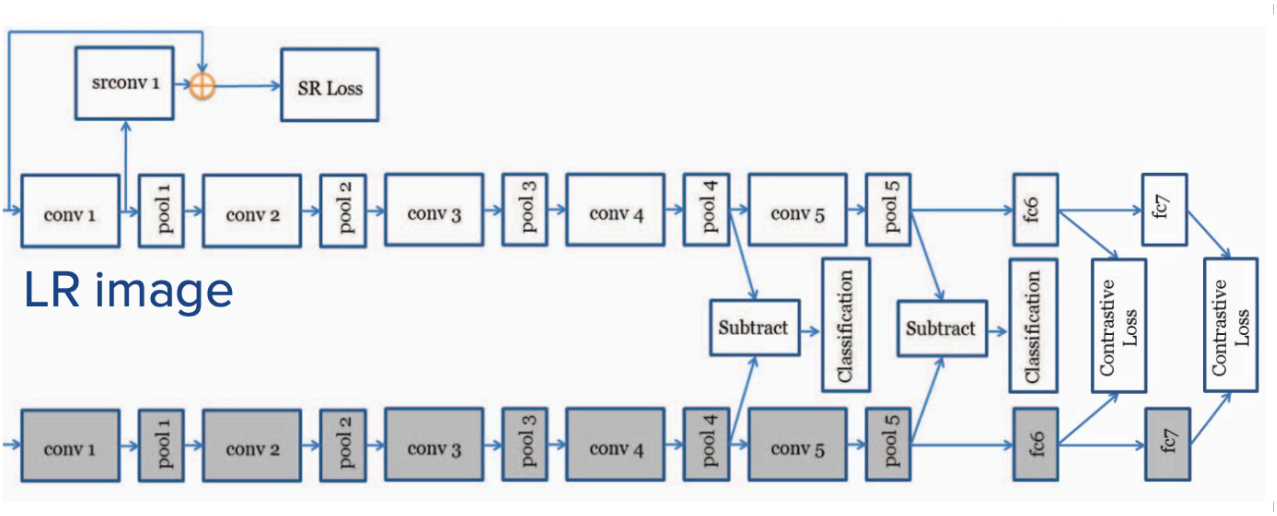


Fig 3.6.1 GenLR Net architecture.

Contrastive Loss at the higher level features: First, contrastive loss is applied between fc7 layers of both the channels of the network and the network is trained. This will help to bring the HR and LR positive samples closer and move the negative samples far apart. The contrastive loss between any two features are computed as,

$$L_{Cont} = \frac{1}{2N}\sum_{i=1}^{N} l_i D_i^2 + (1 - l_i) max(\delta - D_i, 0)^2$$

D_i is the L_2 distance between the corresponding features and δ is the margin by which the features of dissimilar subjects are to be separated. N is the total number of training samples and l_i is the corresponding binary label of each training pair. The loss will only be propagated through the channel which takes the LR input data, since the channel weights corresponding to the HR image is fixed and not updated. This is different from the standard metric learning where the weights corresponding to both the channels are updated simultaneously. With this modification, we obtain a verification rate of 84.00%. It has been observed that enforcing the losses between intermediate layers can boost the performance. The impact of introducing the contrastive loss between the previous layers (like fc6, pool5 and pool4) and so on. The results improve to 86.00% when this loss is incorporated in the fc6 layer. But the same loss in the previous layers, the performance decreases. This is because this loss is not suitable for the mid-level or low-level features, which needs more flexibility to adapt to the different characteristics of the input data.

Inter-intra Classification Loss at the mid-level features

Incorporating a constraint which is *softer* than the contrastive loss, and thus may be more suitable for the mid-level features. The difference between the two images (HR and LR) is computed and classified as 1 if they belong to the same class (subject) and 0, if they belong to different classes. Thus a N-class problem is converted to a 2-class problem. This loss also tries to bring samples from the same class closer and push samples from different classes apart. This is less constrained than the contrastive loss since it does not enforce a strict margin between same and different classes. The difference between the activations of pool5 of both the channels are taken and applied to the classification loss on these features, and the verification performance with this modification improves. By also including this loss on the difference vectors at pool4, the performance further improves. As with the contrastive loss, the performance starts decreasing if the loss is enforced in the initial low-level features like pool3, pool2, etc. So in the final network, this loss is applied only on the pool5 and pool4 layers.

Let f1 and f2 be the activations of the pool5 layers of the LR and HR channel respectively. In the face verification experiment, both f1 and f2 $\in R7\times7\times512$ since there are 512 filters at the last convolution layer of conv5. The difference feature is denoted as (f1 - f2) and the resultant tensor is converted into a column vector (denoted by f). This vector is connected to a softmax layer with two

nodes, such that f is classified as 1 if the input image pair belongs to the same class and 0 otherwise. Let θf be the weights connecting f to the softmax layer, the activations can be computed as

$$g = \phi\left(\theta_f^T f\right)$$

Here, φ is the non linear function *reLU*. The softmax prob- abilities are computed as follows

$$P_j = \frac{exp(g_j)}{\sum_k exp(g_k)}$$

The cross-entropy classification loss [29] is employed on the softmax probabilities for classifying the input pair. Here $1[l_i = k] = 1$ if the input pair belongs to class k and zero otherwise.

$$L_{Cls} = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=0}^{1} -\mathbb{1}[l_i = k] log P_k$$

Super-Resolution Loss at the low-level features

One way of matching a LR image with a HR image is to first apply super-resolution on the LR image to make it HR and then perform matching. Though super-resolution (SR) approaches are very useful for enhancing the image resolution, they are not designed to perform well for recognition applications .

Applying SR algorithms on LR images separately and using the enhanced images does not result in significant improvement in the recognition performance. So, the super-resolution objective is included along with the verification task. This should result in a boost in the performance since the weights of the network are now responsible for simultaneously improving the resolution and the verification performance. This loss is used only if the HR images corresponding to the LR images are available. The output of *conv1* layer and give it as an input to a *srconv1* layer. The *srconv1* layer predicts the residual images and so this output is added with the corresponding LR images to give the final super-resolution output. Next, the reconstruction loss between the super-resolution output and the original HR image is computed. Let s^l_i be the output of the *srconv1* layer, then the reconstruction loss is given below as, Here, $(x_i)_{hr}$ is the HR image of the corresponding x_i .

$$L_{SR} = \left\| \left(\mathbf{s}_i^l + \mathbf{x}_i^l\right) - \left(\mathbf{x}_i^l\right)_{hr} \right\|_2^2$$

Here, $(x_i)_{hr}$ is the HR image of the corresponding x_i .

Here, it is assumed that the HR version of the LR images are available for training which is usually the case with super- resolution tasks. With this, we observe that the verification rate improved to 90.00%. In summary, the entire network is trained by jointly minimising all the losses. Specifically, there is a super-resolution loss after *conv1* layer, two classification losses each at *pool4* and *pool5* layers and two contrastive losses each at fc6 and fc7 layers. The final loss can be formulated as below,

$$L = \lambda_1 L_{SR} + \lambda_2 L_{Cls}^{pool4} + \lambda_3 L_{Cls}^{pool5} + \lambda_4 L_{Con}^{fc6} + \lambda_5 L_{Con}^{fc7}$$

Chapter 4:

Results and Discussion

4.1 Implemented Model

For our project we are using mobilenet ssd for the following reasons:

1. SSD300 has 79.6% mAP and SSD512 has 81.6% mAP which is faster and more accurate than R-CNN of 78.8% mAP. As we are trying to make it real time we compromise accuracy for speed.
2. Mobilenet ssd is a smaller, less heavy network compared to other deep learning models which makes it ideal to run it on embedded systems.

Even after using mobilenet SSD, it's still hard to run it on small scale embedded systems. For that we need to track the objects while skipping a few frames [7]. This makes the tracking process faster and gives a bit of fps boost. As stated in the paper Multiple object tracking (MOT) is one of computer vision tasks that combines three main components: detection, tracker, and matching process that pairs detection and tracker results. Combining accurate but slow detection with fast and moderately accurate tracker can lead to good MOT performance. However, running detection and matching processes in every frame can make the system slow, especially in low-resources machines. Therefore, to speed-up the system, we propose running detection and matching processes in every k frame and running only a tracker in between. The speed-up results are observed in both high-resources machines and low-resources embedded boards. [7]

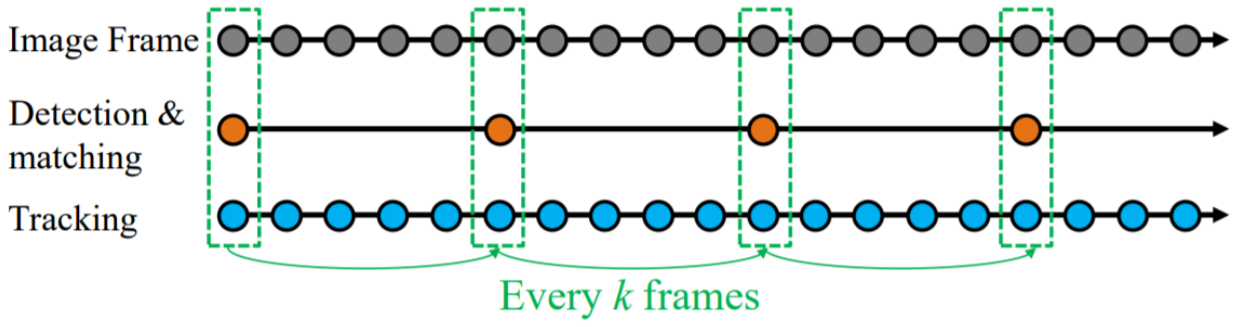


Fig.4.1: Mechanism for frame skipping.

Combining slow-accurate deep-learning based detection + fast moderately accurate tracking. To compensate for the slow detection, run detection & matching algorithms in every k frame. −Run only fast tracking in between. −Focus on the speed gain of MOT system

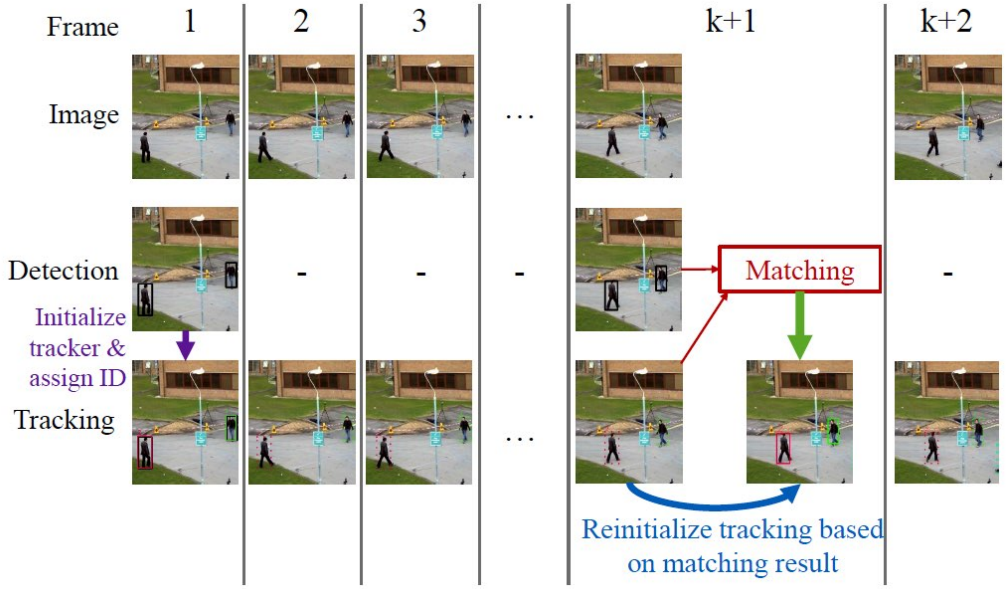


Fig 4.2 Methodology and Frame-wise explanation of frame skipping

4.2 Tracking algorithm used

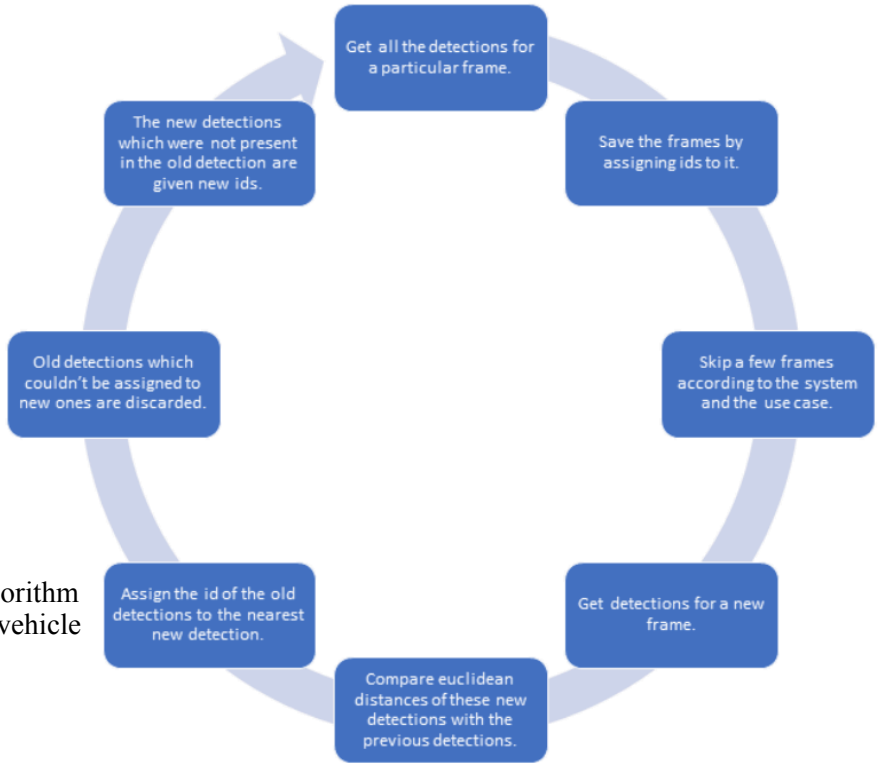


Fig.4.3: The Detection algorithm followed for vehicle detection.

4.3 Code

```
from imutils.video import VideoStream
from imutils.io import TempFile
from imutils.video import FPS
from datetime import datetime
from threading import Thread
import numpy as np
import argparse
import imutils
import dlib
import time
import cv2
import os
from scipy.spatial import distance as dist
from collections import OrderedDict

class CentroidTracker():
    def __init__(self, maxDisappeared=5):
        self.nextObjectID = 0
        self.objects = OrderedDict()
        self.disappeared = OrderedDict()
        self.maxDisappeared = maxDisappeared

    def register(self, centroid):
        self.objects[self.nextObjectID] = centroid
        self.disappeared[self.nextObjectID] = 0
        self.nextObjectID += 1

    def deregister(self, objectID):
        del self.objects[objectID]
        del self.disappeared[objectID]


    def update(self, rects):
        if len(rects) == 0:
            for objectID in list(self.disappeared.keys()):
                self.disappeared[objectID] += 1
                if self.disappeared[objectID] > self.maxDisappeared:
                    self.deregister(objectID)

                return self.objects

        inputCentroids = np.zeros((len(rects), 2), dtype="int")


        inputCentroids = np.zeros((len(rects), 2), dtype="int")
        for (i, (startX, startY, endX, endY)) in enumerate(rects):
            cX = int((startX + endX) / 2.0)
            cY = int((startY + endY) / 2.0)
            inputCentroids[i] = (cX, cY)

        if len(self.objects) == 0:
            for i in range(0, len(inputCentroids)):
                self.register(inputCentroids[i])

        else:
            objectIDs = list(self.objects.keys())
            objectCentroids = list(self.objects.values())

            D = dist.cdist(np.array(objectCentroids), inputCentroids)
            rows = D.min(axis=1).argsort()
            cols = D.argmin(axis=1)[rows]

            usedRows = set()
            usedCols = set()

            for (row, col) in zip(rows, cols):
                if row in usedRows or col in usedCols:
                    continue

                objectID = objectIDs[row]
                self.objects[objectID] = inputCentroids[col]
                self.disappeared[objectID] = 0
                usedRows.add(row)
                usedCols.add(col)

            unusedRows = set(range(0, D.shape[0])).difference(usedRows)
            unusedCols = set(range(0, D.shape[1])).difference(usedCols)

            if D.shape[0] >= D.shape[1]:
                for row in unusedRows:
                    objectID = objectIDs[row]
                    self.disappeared[objectID] += 1

                    if self.disappeared[objectID] > self.maxDisappeared:
                        self.deregister(objectID)


            else:
                for col in unusedCols:
                    self.register(inputCentroids[col])
```

```
        return self.objects

model_path= 'MobileNetSSD_deploy.caffemodel'
prototxt_path = 'MobileNetSSD_deploy.prototxt'

CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
        "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
        "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
        "sofa", "train", "tvmonitor"]




net = cv2.dnn.readNetFromCaffe(prototxt_path,
        model_path)
#net.setPreferableTarget(cv2.dnn.DNN_TARGET_MYRIAD)

ct = CentroidTracker()
cap = cv2.VideoCapture("project_video.mp4")

t = time.time()
frames_count = 0
frame_read = 0

while True:
    frame_read +=1
    if time.time() - t > 1:
        print(frames_count / (time.time() - t))
        frames_count = 0
        t = time.time()

    ret, frame = cap.read()
    if frame is None:
        break

    if frame_read % 10  == 0:
        frame_resized = cv2.resize(frame,(300,300))
        blob = cv2.dnn.blobFromImage(frame_resized, 0.007843, (300, 300), (127.5, 127.5, 127.5), False)
        net.setInput(blob)
        detections = net.forward()
        #dets.append(detections)
```

```
    if frame_read % 10  == 0:
        frame_resized = cv2.resize(frame,(300,300))
        blob = cv2.dnn.blobFromImage(frame_resized, 0.007843, (300, 300), (127.5, 127.5, 127.5), False)
        net.setInput(blob)
        detections = net.forward()
        #dets.append(detections)

        rects = []

        h, w = frame.shape[:2]

        detections[0,0,:,3:7] = detections[0,0,:,3:7] * np.array([w,h,w,h])
        detections[0,0,:,3:7] = detections[0,0,:,3:7].astype("int")

        for i in range(detections.shape[2]):
            if detections[0,0,i,2] > 0.5:
                rects.append(detections[0,0,i,3:7].astype("int"))

                cv2.rectangle(frame, (detections[0,0,i,3],detections[0,0,i,4]), (detections[0,0,i,5],detections[0,0,i,6]), (0,255,0))
                label = "{} : {:.2f}%".format(CLASSES[int(detections[0,0,i,1])], detections[0,0,i,2]*100)
                cv2.putText(frame, label, (detections[0,0,i,3],detections[0,0,i,4]), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

        #print(len(rects))
        objects = ct.update(rects)

        for (objectID, centroid) in objects.items():
            text = "ID {}".format(objectID)
            cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10),cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
            cv2.circle(frame, (centroid[0], centroid[1]), 4, (0, 255, 0), -1)




        cv2.imshow('frame',frame)

    frames_count += 1
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

4.4 The Results

Fig.4.4:Results on Live video feed. The cars can be seen marked with ids

4.5 Important Observations

- The detection by SSD model is known by bounding box.
- The model although it fails to detect the object sometimes , the tracking algorithm keeps track of the object. This compensates for the trade off we did for speed with accuracy.
- Additionally, We don't need to detect objects in every frame owing to the object tracking algorithm.

Chapter 5:
Conclusion and Future Scope

5.1 Conclusion

The results conclude that it is possible to achieve this frame rate with the help of an SSD algorithm. If worked on more with better resources it is possible to improve the performance for better real time application. Additional hardwares like Intel Movidius stick or an external GPU, can although increase the cost but performance will increase exponentially. Thus the trade off between cost and performance has to be balanced based on the use case. Further the networks individually being very effective and remarkable, are somewhere or the other lacking to be wholesome for the application and hence driving us to use SSD algorithm.

5.2 Future Scope

There are a number of applications that can be implemented using this technology.

- Most traffic jams happen due to the slow reaction time of the drivers. Using this device we can calculate the distance and position between two vehicles and we can analyse the correct time and speed to move.
- An intelligent system to do the mundane tasks of surveillance including counting vehicles, tracking specific vehicle models or colour.
- It can process large amounts of videos in a short time and can be used in real time investigation and processing.

Chapter 6:

References

[1] Li, Yuxi, et al. “Tiny-dsod: Lightweight object detection for resource-restricted usages.” *arXiv preprint arXiv:1807.11013* (2018).

[2] Howard, Andrew G., et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications.” arXiv preprint arXiv:1704.04861 (2017).

[3] Liu, Wei, et al. “Ssd: Single shot multibox detector.” European conference on computer vision. Springer, Cham, 2016.

[4] Sang, Jun, et al. “An improved YOLOv2 for vehicle detection.” Sensors 18.12 (2018): 4272.

[5] Wu, Bichen, et al. “Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving.” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2017.

[6] Mudunuri, Sivaram Prasad, Soubhik Sanyal, and Soma Biswas. “GenLR-Net: Deep framework for very low resolution face and object recognition with generalization to unseen categories.” *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, 2018.

[7] Astrid, Marcella & Lee, Seung-Ik & Seo, Beom-Su. (2018). Speeding-up Multiple Object Tracking by Frame Skipping.