Object Detection in Refrigerators using Tensorflow

by

Kirti Agarwal
B.Tech., Uttar Pradesh Technical University, India 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Kirti Agarwal, 2018
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Object Detection in Refrigerators using Tensorflow

by

Kirti Agarwal
B.Tech., Uttar Pradesh Technical University, India 2011

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

ABSTRACT

Object Detection is widely used in many applications such as face detection, detecting vehicles and pedestrians on streets, and autonomous vehicles. Object detection not only includes recognizing and classifying objects in an image, but also localizes those objects and draws bounding boxes around them. Therefore, most of the successful object detection networks make use of neural network based image classifiers in conjunction with object detection techniques. Tensorflow Object Detection API, an open source framework based on Google's TensorFlow, allows us to create, train and deploy object detection models.

This thesis mainly focuses on detecting objects kept in a refrigerator. To facilitate the object detection in a refrigerator, we have used Tensorflow Object Detection API to train and evaluate models such as SSD-MobileNet-v2, Faster R-CNN-ResNet-101, and R-FCN-ResNet-101. The models are tested as a) a pre-trained model and b) a fine-tuned model devised by fine-tuning the existing models with a training dataset for eight food classes extracted from the ImageNet database. The models are evaluated on a test dataset for the same eight classes derived from the ImageNet database to infer which works best for our application.

The results suggest that the performance of Faster R-CNN is the best on the test food dataset with a mAP score of 81.74%, followed by R-FCN with a mAP of 80.33% and SSD with a mAP of 76.39%. However, the time taken by SSD for detection is considerably less than the other two models which makes it a viable option for our objective. The results provide substantial evidence that the SSD model is the most suitable model for deploying object detection on mobile devices with an accuracy

of 76.39%. Our methodology and results could potentially help other researchers to design a custom object detector and further enhance the precision for their datasets.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables viii

List of Figures x

Acknowledgements xiv

1 Introduction 1

1.1 Problem Definition and Motivation 1

1.2 Goals and Research Methodology 4

1.3 Contributions 6

1.4 Thesis Outline 6

2 Background and Related work 8

2.1 Concepts 8

2.1.1 Artificial Neural Network (ANN) 8

2.1.2 Deep Learning 9

2.1.3 Convolutional Neural network (CNN) 11

2.2 Image Classification Architectures 15

2.2.1 Residual Networks 16

2.2.2 MobileNet 18

2.3 Object Detection Techniques 21

2.3.1 Region-based Convolutional Neural Network (R-CNN) 22

2.3.2 Fast Region-based Convolutional Neural Network (Fast R-CNN) 24

2.3.3 Faster Region-based Convolutional Neural Network (Faster R-CNN) 25
2.3.4 Region-based Fully Convolutional Network (R-FCN) 27
2.3.5 Single-Shot Multibox Detector (SSD) 28
2.4 Open Source Frameworks for Object Detection 30
2.4.1 Tensorflow Object Detection API 31
2.5 Summary 32

3 Implementation of Object Detection using Tensorflow 34
3.1 Data Collection and Pre-processing 34
3.2 Tensorflow Object Detection API for Fine-tuned Model 37
3.2.1 Object Detection with SSD-MobileNet-v2 37
3.2.2 Object Detection with FRCNN-ResNet-101 44
3.2.3 Object Detection with RFCN-ResNet-101 47
3.3 Evaluation Metrics 50
3.3.1 Precision 50
3.3.2 Recall 51
3.3.3 Intersection Over Union (IoU) 51
3.3.4 Precision-Recall Curves 52
3.3.5 Mean Average Precision (mAP) 53
3.4 Summary 54

4 Results and Discussion 55
4.1 Pre-trained vs. Fine-tuned Models 55
4.1.1 SSD-MobileNet-v2: Pre-trained and Fine-tuned Model 56
4.1.2 FRCNN-ResNet-101: Pre-trained and Fine-tuned Model . . . 61
4.1.3 RFCN-ResNet-101: Pre-trained and Fine-tuned Model 65
4.2 Fine-tuned Models with Different Configuration Settings 69
4.2.1 SSD-MobileNet-v2 Fine-tuned Models 69
4.2.2 FRCNN-ResNet-101 Fine-tuned Models 72
4.2.3 RFCN-ResNet-101 Fine-tuned Models 74
4.3 Speed-accuracy Trade-off of Different Object Detection Models 76
4.4 Discussion 77
4.5 Summary 78

5 Conclusions 79

5.1 Summary 79
5.2 Contributions 80
5.3 Future Work 81

Bibliography **84**

List of Tables

4.1 SSD: Precision values for eight classes for pre-trained and fine-tuned models at different steps of training 58
4.2 SSD: Recall values for eight classes for pre-trained and fine-tuned models at different steps of training 59
4.3 SSD: AP values for eight classes, and mAP values for pre-trained and fine-tuned models at different steps of training 59
4.4 FRCNN: Precision values for eight classes for pre-trained and fine-tuned models at different steps of training 63
4.5 FRCNN: Recall values for eight classes for pre-trained and fine-tuned models at different steps of training 63
4.6 FRCNN: AP values for eight classes, and mAP values for pre-trained and fine-tuned models at different steps of training 64
4.7 RFCN: Precision values for eight classes for pre-trained and fine-tuned models at different steps of training 67
4.8 RFCN: Recall values for eight classes for pre-trained and fine-tuned models at different steps of training 67
4.9 RFCN: AP values for eight classes, and mAP values for pre-trained and fine-tuned models at different steps of training 68
4.10 SSD: AP values for eight classes, and mAP values for fine-tuned model with the default configuration at different steps of training 70
4.11 SSD: AP values for eight classes, and mAP values for fine-tuned model with learning rate: 0.005 at different steps of training 70
4.12 SSD: AP values for eight classes, and mAP values for fine-tuned model with learning rate: 0.003 at different steps of training 71
4.13 SSD: AP values for eight classes, and mAP values for fine-tuned model with three aspect ratios at different steps of training 71
4.14 FRCNN: AP values for eight classes, and mAP values for fine-tuned model with the default configuration at different steps of training . . 72

4.15 FRCNN: AP values for eight classes, and mAP values for fine-tuned model with five aspect ratios at different steps of training 73
4.16 FRCNN: AP values for eight classes, and mAP values for fine-tuned model with 100 proposals at different steps of training 73
4.17 RFCN: AP values for eight classes, and mAP values for fine-tuned model with the default configuration at different steps of training . . 74
4.18 RFCN: AP values for eight classes, and mAP values for fine-tuned model with five aspect ratios at different steps of training 75
4.19 RFCN: AP values for eight classes, and mAP values for fine-tuned model with 100 proposals at different steps of training 75
4.20 Results of pre-trained models vs. fine-tuned models 77
4.21 Results of fine-tuned models with different configuration settings . . . 77
4.22 Results of the three fine-tuned models 78

List of Figures

2.1 An Artificial Neural Network (ANN)[1] 9
2.2 Comparison in performance of traditional learning methods vs. deep learning methods [41] 10
2.3 A Convolutional Neural Network (CNN)[2] 11
2.4 6x6 input feature map and 3x3 filter 12
(a) Input feature map 12
(b) Filter 12
2.5 Convolution process of 6x6 input with 3x3 filter 12
(a) Convolution process 12
(b) Output feature map 12
2.6 A function of max-pooling layer with 2x2 window and stride 2 13
(a) Input feature map 13
(b) Output feature map after max-pooling 13
2.7 A residual learning block [20] 17
2.8 An example of regular convolution[3] 19
2.9 Two different steps of depthwise separable convolution[4] 19
(a) Depthwise convolution 19
(b) Pointwise convolution 19
2.10 Block diagram of MobileNet-v1 and MobileNet-v2[5] 20
(a) Depthwise separable convolution block 20
(b) Bottleneck residual block 20
2.11 Comparison of MobileNet-v1 vs. v2[6] 21
2.12 Difference between image classification and object detection 22
(a) Image classification: Tomato 22
(b) Object detection: Apple, Orange, and Pear 22
2.13 Region-based Convolutional Neural Network (R-CNN) [16] 23
2.14 Fast region-based Convolutional Neural Network (Fast R-CNN) [15] . 24
2.15 Faster region-based Convolutional Neural Network (Faster R-CNN) [46] 26

2.16 Region-based fully Convolutional Network (R-FCN) [7] 27
2.17 Single-Shot Detector (SSD) [38] 29
2.18 mAP vs. GPU time for different meta-architectures [25] 32

3.1 An image example from the Apple dataset of the ImageNet database 35
3.2 A sample of object label map file 38
3.3 A sample of updated configuration file 39
3.4 Decline of total loss when fine-tuning SSD-MobileNet-v2 41
3.5 Development of overall mAP when fine-tuning SSD-MobileNet-v2 . . 41
3.6 Development of overall mAP when fine-tuning SSD-MobileNet-v2 (blue= with learning rate 0.004 (the default one), light blue= with learning rate 0.003, pink= with learning rate 0.005) 43
3.7 Development of overall mAP when fine-tuning SSD-MobileNet-v2 (blue= with default aspect ratios, red= with modified aspect ratios) 43
3.8 Decline of total loss when fine-tuning FRCNN-ResNet-101 45
3.9 Development of overall mAP when fine-tuning FRCNN-ResNet-101 . 45
3.10 Development of overall mAP when fine-tuning FRCNN-ResNet-101 (red= with default aspect ratios, light blue= with modified aspect ratios) 46
3.11 Development of overall mAP when fine-tuning FRCNN-ResNet-101 (red= with 300 number of proposals (the default one), orange= with 100 number of proposals) 47
3.12 Decline of total loss when fine-tuning RFCN-ResNet-101 48
3.13 Development of overall mAP when fine-tuning RFCN-ResNet-101 . . 48
3.14 Development of overall mAP when fine-tuning RFCN-ResNet-101 (blue= with default aspect ratios, red= with modified aspect ratios) 49
3.15 Development of overall mAP when fine-tuning RFCN-ResNet-101 (blue= with 300 number of proposals (the default one), light blue= with 100 number of proposals) 50
3.16 Areas of intersection and union 52
(a) Area of intersection (overlap) 52
(b) Area of union 52

4.1 Detections for class Apple (present in pre-trained models) 56
(a) Pre-trained model 56
(b) Fine-tuned model 56

4.2 Detections for class Tomato (not present in pre-trained models) . . . 56
(a) Pre-trained model . . . 56
(b) Fine-tuned model . . . 56
4.3 SSD: Performance of pre-trained model vs. fine-tuned model for class Apple . . . 57
4.4 SSD: Performance of pre-trained model vs. fine-tuned model for class Orange . . . 57
4.5 Detections of SSD models for class Pear (not present in pre-trained model) . . . 60
(a) Pre-trained model . . . 60
(b) Fine-tuned model . . . 60
4.6 Detections of SSD models for class Apple (present in pre-trained model) 60
(a) Pre-trained model . . . 60
(b) Fine-tuned model . . . 60
4.7 Detections of SSD models for class Apple (present in pre-trained model) 61
(a) Pre-trained model . . . 61
(b) Fine-tuned model . . . 61
4.8 FRCNN: Performance of pre-trained model vs. fine-tuned model for class Apple . . . 62
4.9 FRCNN: Performance of pre-trained model vs. fine-tuned model for class Orange . . . 62
4.10 Detections of Faster R-CNN models for class Bell Pepper (not present in pre-trained model) . . . 64
(a) Pre-trained model . . . 64
(b) Fine-tuned model . . . 64
4.11 Detections of Faster R-CNN models for class Orange (present in pre-trained model) . . . 65
(a) Pre-trained model . . . 65
(b) Fine-tuned model . . . 65
4.12 RFCN: Performance of pre-trained model vs. fine-tuned model for class Apple . . . 66
4.13 RFCN: Performance of pre-trained model vs. fine-tuned model for class Orange . . . 66
4.14 Detections of R-FCN models for classes Tomato and Lemon (not present in pre-trained model) . . . 68

(a) Pre-trained model 68
(b) Fine-tuned model 68
4.15 Detections of R-FCN models for class Apple (present in pre-trained model) 69
(a) Pre-trained model 69
(b) Fine-tuned model 69
4.16 Speed-accuracy comparison of the three models 76

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Hausi A. Müller, my supervisor, for his supervision, enthusiasm, motivation, and encouragement throughout my Master's program. I want to thank him for providing me the opportunity to work with him. I am really grateful to him for the continuous guidance, support, and feedback during this research.

Dr. Alex Thomo, for being my committee member and mentor. I am really grateful to him for guiding and providing the feedback throughout this research work.

Harshit, Miguel, Priya, Roshni, and all Rigi and Pita group members, for their valuable support, advice, and feedback in my research and academics over the entire course of this beautiful journey.

My husband, family members and friends, for encouraging and inspiring me all through my journey. It would be impossible to realize this dream without their immense support and love.

Chapter 1

Introduction

1.1 Problem Definition and Motivation

Object detection is a term related to computer vision and image processing techniques for locating and annotating all the potential objects in the images. The images can be either static pictures or moving frames. The task of object detection is closely related to two other problems: image classification and object localization. Image classification also referred to as image recognition, is about classifying an image to a particular class out of all the possible classes. Object localization is more advanced than image recognition, and it demands to localize the labeled object in an image. Object detection is the most complex of all—it involves labeling and localizing multiple objects in an image. Object detection techniques are extensively used for applications related to face detection, locating pedestrian on streets or players on football grounds, and detecting vehicles on roads.

The computer vision community started growing enthusiasm towards deep learning models after the success of AlexNet, a deep convolutional neural network (CNN), at the ImageNet Challenge 2012 for image recognition [33]. The success of AlexNet led to the development of several other successful deep neural networks, including VGGNet [50], GoogleNet (Inception-v1) [55], and Residual Network (ResNet) [20] for the task of image classification. Since object detection is an extension of image recognition, most of the successful object detection models are based on deep neural networks for image classification. Consequently, most of the object detection models such as Faster Region-based Convolutional Neural Network (Faster R-CNN) [46], Region-based Fully Convolutional Network (R-FCN) [7], and Single-Shot Detector

(SSD) [38] are built on top of image classification models such as ResNet and Inception. However, a major constraint of a deep CNN is that it requires a large amount of annotated datasets for training them. To overcome the problem of enormous training datasets, several public datasets such as the ImageNet database [9], the Common Objects in Context (COCO) dataset [36], and the Open Images dataset [31] are readily available. Moreover, the development of Graphical Processing Units (GPUs) helps in training the computationally expensive deep neural networks.

Tensorflow Object Detection API,[1] OpenCV's DNN library,[2] and Microsoft Cognitive Toolkit[3] provide open-source frameworks to construct, train and deploy object detection models. We are using Tensorflow because of its popularity and ease of use. Tensorflow Object Detection API also presents pre-trained models for object detection. The pre-trained models are trained on some of the public datasets such as COCO, Kitti [14], Open Images, and Atomic Visual Actions (AVA) v2.1 [19]. The performance of the pre-trained models is sufficient for recognizing objects, such as a person, television and cars, but not good enough for detecting the food classes relevant for our purposes. Moreover, the pre-trained models do not even detect some of the food classes expected in our application. Therefore, it is essential to fine-tune the existing models for our particular datasets and application requirements.

To test and refine the pre-trained models, we are using eight food datasets from the ImageNet database. ImageNet is a large-scale hierarchical database with millions of images in more than 20,000 classes. Researchers use the ImageNet database for image classification applications extensively [33, 50, 55, 20]. There are several pre-trained models for image classification based on this database. However, the pre-trained object detection models for the ImageNet database are not available yet. The advantage of using the pre-trained network is that if we want to train a CNN model from scratch, it might take weeks or months for the model to converge and perform reasonably well. However, if we use a pre-trained model as a starting point, our CNN model can be fine-tuned in considerably less time and with better accuracy. The method of fine-tuning a pre-trained model on another dataset is known as transfer learning. In transfer learning, a pre-trained model transfers the learned features or weights to initiate the process of fine-tuning. Another advantage of transfer learning is that it allows us to train an object detector with the limited amount of a training

[1] https://github.com/tensorflow/models/tree/master/research/object_detection
[2] https://github.com/opencv/opencv/wiki/Deep-Learning-in-OpenCV
[3] https://www.microsoft.com/en-us/cognitive-toolkit

dataset [43, 23, 30].

Moreover, the ImageNet database has bounding box data for some of the datasets, and the bounding box data for each class dataset is labeled for that particular class only. However, object detection requires bounding box data to be annotated for all the possible classes and not just for one class. So, the ImageNet bounding box data requires pre-processing before it can be used for training and evaluating an object detection model.

We extracted and pre-processed eight food datasets from the ImageNet database to evaluate pre-trained object detection models. Moreover, these datasets are also used for fine-tuning the pre-trained models to enhance the performance and henceforth, achieve better results for our application.

To improve the accuracy of object detector further, we can also leverage the available contextual information. For instance, we may utilize the grocery context like the list of ingredients that a user bought to say whether that object could be present in a refrigerator or not. Alternatively, if there are two bounding boxes on one object, we can infer better based on the grocery context. Suppose, a user does not initially have apples and tomatoes in her refrigerator, and the user buys tomatoes, but not apples as gathered from her grocery shopping receipt. If the object detector detects two bounding boxes on one object saying apple and tomato, we can infer it would be tomato and not apple based on the grocery context. Another approach is to use the personal context of a user. Suppose, a user is allergic to milk and its products. If the object detector detects milk, we might argue that since the user is allergic to milk, the user will not have milk in her refrigerator.

Smart refrigerators, like Samsung Family Hub and LG Smart InstaView, also utilize object detection technology to track what is kept inside them. However, the smart refrigerators mostly work on a closed platform which is not readily available for research and development purposes. Another problem with those refrigerators is that they are expensive compared to regular refrigerators. Through this work, we are trying to transform a regular refrigerator into a smart one by enabling object detection without a substantial increase to their cost.

There are numerous objects kept in our refrigerator, but as part of this thesis, we are focusing only on detecting eight objects—*Apple, Bell pepper, Cauliflower, Lemon, Orange, Pear, Tomato and Turnip*. Also, in the scope of this thesis we are fine-tuning the pre-trained models for only three networks, SSD with MobileNet-v2, Faster R-CNN with ResNet-101, and R-FCN with ResNet-101. In the end, we are

comparing the three models considered above to come up with a suitable model for our application of detecting the eight objects stated earlier in a refrigerator. To evaluate and compare the object detection models, Tensorflow uses mAP (Mean Average Precision) score as an evaluation protocol.[4] Mean Average Precision (mAP) is a key evaluation protocol to measure and compare the accuracy of object detection models. mAP is the mean of average precision (AP) of all the classes used to train a model. Average precision (AP) gives us an idea about the area under Precision-Recall (PR) curve for each object class (described in detail in Section 3.3.5).

Based on the above problems and motivation, we formulated four Research Questions (RQs):

RQ1 How can we exploit the food datasets from image databases, such as the ImageNet database, to evaluate and refine object detection models?

RQ2 How can we evaluate the pre-trained object detection models based on a test dataset derived from the ImageNet database?

RQ3 How can we fine-tune the existing object detection models to detect objects in a refrigerator?

RQ4 How can we infer which object detection model works best for detecting objects in refrigerators?

1.2 Goals and Research Methodology

The principal objective of this thesis is to train an object detection model that can be deployed to detect objects kept in a refrigerator. The detected objects can, in turn, be used to recommend recipes based on them. Additionally, to train an object detector, we need a large number of labeled training examples. Therefore, gathering and annotating images is another important contribution of this thesis.

To alleviate the aforementioned problems and accomplish the objectives mentioned above, the primary goals of this thesis are:

- Extract food datasets and utilize them for fine-tuning the pre-trained models and assessing the pre-trained and fine-tuned models.

[4] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/evaluation_protocols.md

- Evaluate the performance of the selected pre-trained models on the test dataset.
- Improve the performance by fine-tuning the pre-trained models based on the training examples from food datasets.
- Compare different fine-tuned models to conclude which one is suitable for our objective.

The stated goals are addressed and realized in several steps as described below:

- First, after selecting eight food classes, their image datasets are collected from the ImageNet database. Each dataset has bounding boxes for their class and not for other classes. However, for object detection, we are supposed to be working with all the eight classes and not the individual classes. So, we pre-process each dataset to draw bounding boxes for all the potential classes, using an annotation tool known as LabelImg.[5] After pre-processing, we divide each class dataset into three datasets, training, validation, and test dataset. Then, combine the corresponding datasets from each class into one dataset. In the end, we have one training dataset, one validation dataset, and one test dataset.
- Second, to evaluate the performance of pre-trained models, we selected the following three Tensorflow object detection models: SSD with MobileNet-v2, Faster R-CNN with ResNet-101, and R-FCN with ResNet-101. The three models are selected based on the literature review of object detection techniques explained in Chapter 2. As discussed in the literature review, the SSD model is the fastest, Faster R-CNN is most accurate, and R-FCN is in between the two with respect to performance. The three models are tested based on the test dataset by calculating mAP (Mean Average Precision) based on AP of all the classes for each of the models mentioned above.
- Third, to enhance the accuracy further, we fine-tune the existing models based on the training and validation datasets derived in step one. To train a model in Tensorflow, we have to convert training and validation datasets to training and test TFRecords, a record format supported by Tensorflow for training and validation purposes.[6] The three models mentioned above are fine-tuned for eight classes with the training and test TFRecords.

[5] https://github.com/tzutalin/labelImg

[6] https://www.tensorflow.org/api_guides/python/reading_data#file_formats

- Lastly, we compare the pre-trained models with their fine-tuned counterparts to investigate the improvement in performance. Moreover, the three fine-tuned models are evaluated and examined based on their speed and accuracy for the test dataset.

The best model is then deployed to detect objects (ingredients) in a refrigerator.

1.3 Contributions

The following section highlights the contributions of this thesis:

C1 A method for pre-processing the dataset for eight classes to create training, validation, and test datasets required for object detection.

C2 A survey of different techniques of object detection to aid in selecting pre-trained models.

C3 A method for fine-tuning the existing models to improve the performance from 45.47% to 76.39% for SSD, 63.09% to 81.74% for Faster R-CNN, and 69.71% to 80.33% for R-FCN.

C4 A technique for selecting the most suitable model for our application.

1.4 Thesis Outline

This first chapter presented our motivation, research objectives, and a summary of contributions. The remaining chapters of this thesis are organized as follows:

Chapter 2 describes the key concepts of this thesis followed by the literature review of the state-of-the-art image recognition and object detection methods. It also presents the related work in this area of research.

Chapter 3 presents the image datasets and evaluation metrics used throughout this research. It also illustrates the method of fine-tuning the pre-trained models of object detection and explains the development of mAP during the training phase.

Chapter 4 evaluates and analyzes the performance of pre-trained and fine-tuned models on the test dataset. This chapter also discusses the performance of fine-tuned models in terms of speed and accuracy to help select the best model for our application.

Chapter 5 summarizes our research and offers some insights for future research.

Chapter 2

Background and Related work

Image classification and object detection are terms related to computer vision and image processing techniques. Image classification is classifying an image into a particular class, while object detection is about recognizing, locating and labeling objects in all the possible classes [28]. Object detection has been used in many applications, including: Human-Computer Interaction (HCI), visual computing, robotics, security, transportation, and consumer electronics. Deep learning started dominating computer vision since 2012, when Krizhevsky et al. [33] won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by releasing AlexNet. This chapter highlights the contributions made to the fields of image classification and object detection. The first is an overview of the concepts essential for this thesis, followed by the state-of-the-art of different image classification architectures, and object detection techniques. Lastly, we discuss the open source frameworks available for object detection and speed-accuracy trade-off of various object detection models.

2.1 Concepts

2.1.1 Artificial Neural Network (ANN)

Artificial Neural Network, loosely inspired by the neural structure of our brain, consists of a network of connected nodes known as neurons. Neurons are the central processing unit of the neural networks, and they are connected just like neurons and synapses in a biological brain. The weighted neural connections assist in transmitting signals (features) from one neuron to another [40]. Artificial neural networks learn by examples rather than programming them with task-specific rules. For instance, in

visual pattern recognition, they might learn with training examples of handwritten digits from 0 to 9 and use this knowledge to identify digits in other unseen samples [42].

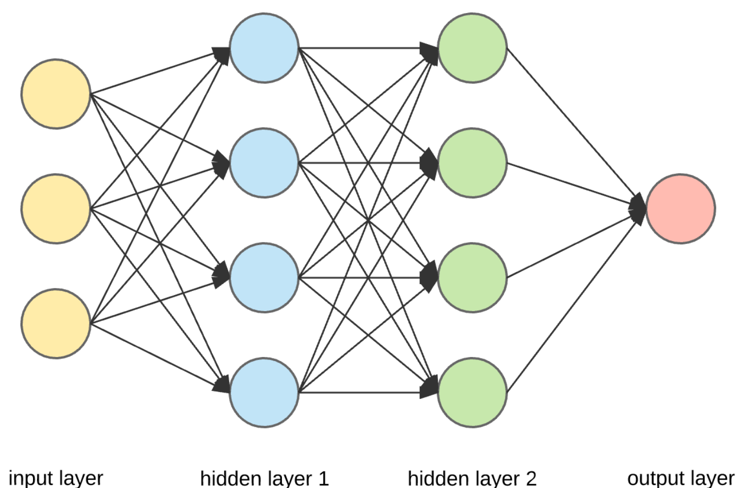


Figure 2.1: An Artificial Neural Network (ANN)[1]

Artificial neural networks (ANN) mostly consist of three types of layers, an input layer, an output layer, and one or more hidden layers. In Figure 2.1 shown above, there are two hidden layers. The first hidden layer learns simple, and basic features and passes them to the second hidden layer. Based on the inputs from the previous layer, the second layer learns features which are more complex and at a more abstract level than the first one. Similarly, layers farther in the network can learn more complicated features and so on. In this way, a multi-layered network can be used effectively to solve sophisticated problems like image classification, object detection, and others.

2.1.2 Deep Learning

Deep Learning, a subfield of machine learning and artificial intelligence, is related to the training of computational models that are composed of multi-layered artificial neural networks. The multi-layered ANN is known as a deep neural network (DNN). Deep in deep neural networks corresponds to the depth of the network. Deep networks usually have more than two layers of hidden neurons between the input and output layers [42]. The deep neural networks have enhanced the state-of-the-art accuracy in

[1] https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

image classification, object detection, speech recognition, language translation, and other areas considerably [34, 22].

Deep learning methods are based on learning representations (features) from data, such as text, images, or videos, rather than implementing task-specific algorithms [56]. Learning can either be unsupervised or supervised. However, most of the practical systems deploy supervised learning to leverage the benefits of deep learning [33, 50, 55, 20]. Supervised learning, in essence, means learning from labeled data. Andrew Ng [41] pointed out in Figure 2.2 shown below, the performance of deep learning methods increases with an increase in the amount of training data as opposed to traditional learning methods that saturate in performance; this characteristic makes the deep learning methods scalable.

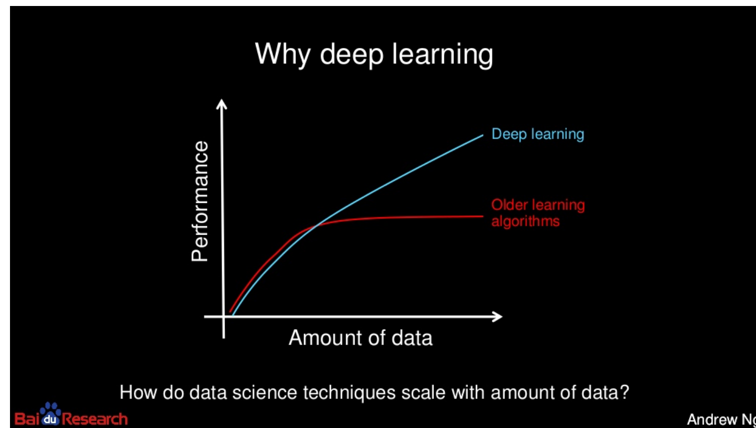


Figure 2.2: Comparison in performance of traditional learning methods vs. deep learning methods [41]

Another important aspect of deep learning is hierarchical feature learning. Feature learning is the automatic extraction of features from raw data. Since a deep neural network consists of multiple layers, it involves learning representations at multiple levels of complexity and different levels of abstractions. As mentioned earlier, the initial layers learn the low-level features and then send them over to the later layers. The following layers then gather high-level features based on previously learned lower level features. Learning features at the different level of abstractions and complexities facilitate deep learning models to learn complex functions without depending on human-crafted features [6, 22, 18].

The scalability and hierarchical feature learning aspects of deep neural networks make them a viable option for tasks such as image classification, object detection,

and speech recognition.

2.1.3 Convolutional Neural network (CNN)

Convolutional Neural Network (CNN) is the most common class of architectures used for deep learning. CNN works similarly as artificial neural network except it has a series of convolutional layers at the beginning. CNN is widely used for visual recognition tasks such as image classification, object detection, and speech recognition. CNN has become ubiquitous among researchers and computer vision community ever since AlexNet showcased outstanding performance at the ImageNet Challenge in 2012 [33].

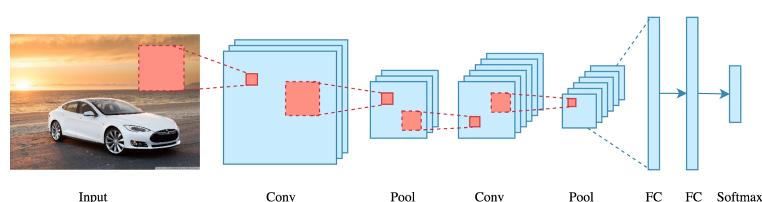


Figure 2.3: A Convolutional Neural Network (CNN)[2]

Figure 2.3 above shows the basic architecture of CNN. The main components of the CNN architecture are as follows:

- Convolution layers
- Pooling layers
- Fully connected layers
- Softmax function

Each input image is passed through multiple convolutional and pooling layers, followed by a couple of fully connected layers. Additionally, the softmax function is applied at the end to perform multiclass classification.

[2] https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Convolution Layers

Convolution layer is the first and primary building block of CNN. Convolution layers are made up of neurons which act as filters. The filters are sequentially slid over the entire image to create feature maps. This process of sliding is known as convolution [4]. Figures 2.4 and 2.5 illustrate the process of convolution on an input 6x6 feature map using a 3x3 filter (kernel). After sliding the filter across the entire input, the result is an output feature map of size 4x4.

3	2	5	1	8	6
1	4	6	4	7	3
8	9	1	5	3	7
2	4	5	2	9	2
3	1	6	1	5	8
9	3	7	3	2	5

(a) Input feature map

1	0	0
0	1	1
1	1	1

(b) Filter

Figure 2.4: 6x6 input feature map and 3x3 filter

3x1	2x0	5x0	1	8	6
1x0	4x1	6x1	4	7	3
8x1	9x1	1x1	5	3	7
2	4	5	2	9	2
3	1	6	1	5	8
9	3	7	3	2	5

(a) Convolution process

31	27	25	26
22	21	30	27
27	24	24	30
28	24	23	25

(b) Output feature map

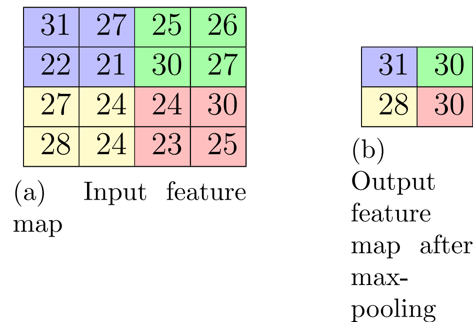
Figure 2.5: Convolution process of 6x6 input with 3x3 filter

As shown in the CNN architecture above, the first block is a block of convolution layers. Each layer works as a different filter and learns different feature maps from an input image or input feature map. For instances, given an image as input, if we have three convolution layers in a block. The output of each layer is a feature map for three different characteristics such as colors, edges, and shapes. Also, the output of each convolution block has three dimensions, height, width, and depth. Height and width correspond to height and width of feature maps, and depth is the number of convolution layers.

Pooling Layers

The pooling layers usually follow convolution layers. The pooling layer reduces the dimensionality of each feature map [42]. The layer takes each independent feature map, subsamples it and creates a compressed feature map. Although, the output of pooling layers is still three dimensional as of convolutional layers. Depth remains the same, only the height and width are condensed.

There are various types of pooling such as max-pooling, sum-pooling, and average-pooling. Max-pooling is the most popular among them [33, 50, 55, 20]. Max-pooling works by sliding a window over each feature map and selecting a max value in that window.



(a) Input feature map

(b) Output feature map after max-pooling

Figure 2.6: A function of max-pooling layer with 2x2 window and stride 2

Figure 2.6 above shows an example of max-pooling where the window size is 2x2 and the stride is 2. Stride length is by how many pixels do we move the filter or window at each step. The main advantage of pooling is that it helps in reducing the number of parameters which in effect reduces the training time.

Fully Connected (FC) Layers

Some fully connected layers are added after the convolution and pooling layers. A fully connected network is a network where all neurons are connected to every neuron in the neighboring layers. As mentioned above, the output of convolution + pooling layers is three dimensional, but fully connected layers consume one-dimensional vector as input. So, we flatten the final output and then send it as the input of the fully connected layers. FC layers then combine these feature maps to devise a model.

Softmax Function

The softmax function is one of the loss functions which can be added as the last layer of a CNN. The softmax function is useful for multiclass detection; it gives us the probability of predicting one class over all the other classes. If we use the softmax function for multiclass detection, the output of this function is a one-dimensional vector of probabilities of all the classes. So the class with the highest probability is the target class.

How to Train a Convolutional Neural Network (CNN)

Training a CNN requires a significant number of annotated images. A large number of images make it feasible for the network to learn various patterns (features) in images to make successful predictions. Before starting the training process, the labeled dataset of images is divided into three parts, training dataset, validation dataset, and test dataset. The training set is used for the actual training process to tune the weights in different layers and create a model that can be used for predictions. The purpose of the validation set is to perform hyperparameter tuning only. Also, the test set is used for the unbiased evaluation of the trained model.

During the training process, the network adjusts the filter weights by an algorithm known as backpropagation. Backpropagation consists of mainly four steps, the forward pass, calculating the loss function, the backward pass, and the weight update. At the beginning of a training process, all the weights are initialized randomly. In the first step of the forward pass, an input image from the training set is passed through the whole network to generate an output. The loss function is then calculated based on the predicted and actual output. Based on the loss function, we find out which weights are to be adjusted and by how much to reduce the loss function. This step is called the backward pass. Then, during the final stage of the weight update, we take all the weights and update them following the last action. All the weights are updated according to the equation given below:

$$
\begin{aligned}
&w = w_i - \eta \frac{dL}{dW}, \\
\text{where } &w = \text{New weight}, \; w_i = \text{Initial weight}, \\
&\eta = \text{Learning rate}, \; \frac{dL}{dW} = \text{Derivative of loss function}
\end{aligned}
$$

The η is a parameter known as the *learning rate*. The selection of the learning rate is an integral part of the training process and is chosen by the user [10]. Since the weights are initialized randomly in the beginning, the learning rate is usually kept higher [51, 58]. A higher learning rate means that it might take less time for the model to converge, that is, to minimize the loss function.

The four steps mentioned above constitute one iteration of learning. To minimize the loss function, which is the ultimate goal of the training process, we perform a fixed number of iterations of learning. The resulting model is then tested over the test dataset.

2.2 Image Classification Architectures

Image classification is the task of assigning a label to an image from a predefined set of classes. The image classification can be binary or multiclass. In binary classification, an input image either belongs to a given class or not. For instance, is there a pedestrian in an image or not? In the example, the pedestrian is a class. In multiclass classification, an input image can belong to one of the several classes. For instance, a multiclass classifier is designed to detect whether an image of a street belongs to a pedestrian, car, or bicycle class [39].

The classical machine learning approaches often need complex feature engineering. The traditional methods require extraction of handcrafted features using algorithms such as Scale Invariant Feature Transform (SIFT) [37], Speeded Up Robust Features (SURF) [5], and Histogram Oriented Gradient (HOG) [8]. Then, the extracted features are fed to the machine learning algorithms like Support Vector Machine (SVM) [54], Random Forest (RF) [35], and Principal Component Analysis (PCA) [11]. On the contrary, in Convolutional Neural Networks (CNN) features are learned automatically without any human intervention [2]. However as mentioned

before, the CNN is mostly based on supervised learning and requires a large number of annotated images for training a network.

In 2009, Deng et al. [9] introduced ImageNet—a new image database which consists of 14 million annotated images in more than 20,000 categories. The images are arranged according to the WordNet structures. The database led to the inception of a benchmark challenge, ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2010 [47]. The ImageNet Challenge is an annual competition in the field of object category classification and detection.

In 2012, Krizhevsky et al. [33] released AlexNet, a deep convolutional neural network trained on 1.2 million labeled images of the ImageNet database under 1,000 categories. AlexNet is trained for five to six days on two powerful NVIDIA GPUs and won the ImageNet Challenge by achieving a top-5 error rate of 15.3% or 84.7% accuracy. The top-5 error rate is the percentage of test images for which the correct class is not in the top 5 predicted classes. In the ImageNet Challenge 2012, AlexNet outperformed a non-CNN SIFT model, second best with a top-5 error rate of 26.2%, becoming state-of-the-art for the tasks related to image recognition. After 2012, ImageNet Challenge witnessed significant improvements in the development of deep learning networks for image recognition. GoogleNet (Inception-v1), a deep CNN proposed by Szegedy et al. [55] marked another important contribution in the field of object recognition by winning the ImageNet challenge in 2014 with a top-5 error rate of 6.7% or 93.3% accuracy. It was the first time a network exceeded 90% accuracy.

Simonyan et al. [50] published VGGNet, another deep network architecture that showcased excellent performance in the ImageNet Challenge 2014 by achieving a top-5 classification error rate of 7.3% or accuracy of 92.7%. It should be noted that the accuracy of the GoogleNet and VGGNet architectures has reached close to human accuracy, which is 94.9% on the ImageNet [47].

2.2.1 Residual Networks

As discussed earlier, one of the most notable achievements of VGGNet and Inception architectures are moving closer to the human error rate for object categorization on the ImageNet database. The human top-5 error rate is 5.1%, which is quite close to that of VGGNet and Inception, which are 7.3% and 6.7%, respectively [47]. However, the ResNet architecture proposed in the ImageNet Challenge 2015 attained a top-5 error rate of 3.57% improving upon the human error-rate on that task [20].

After the success of VGGNet and Inception architectures, the most common trend is to increase the depth of the deep networks to improve the accuracy. However, as observed the performance starts degrading with adding layers. There were two main reasons behind the decline, degradation, and vanishing gradient. Degradation is as the depth of the network increases, accuracy gets saturated and then starts deteriorating. Vanishing gradient, the problem where the gradient signal decreases exponentially as back propagated to the initial layers of the deep networks. Essentially, the error signal becomes too small by the time it reaches the early layers, and as a result, the learning in these layers is negligible. To mitigate the shortcomings resulting from increasing depth, He et al. [20] presented the Residual Networks or ResNet based on the concept of residual learning with skip connections.

The motivation behind the Residual Networks is the inception of the shortcut connections, that skips one or more layers in the network. Shortcut connections, also known as skip connections, merely perform identity mapping and add it to the outputs of the stacked layers. In a regular CNN, a layer computes a direct transformation from x to H(x). However, when the layers are stacked to form deeper networks, the problem of degradation arises. The authors proposed a fix in which they suggested to learn the difference between the two, H(x) and x. The difference is then added to the original input x. The difference is known as residual, and the corresponding learning is known as residual learning. The residual learning is attained by adding skip connections between one or more layers in the network as shown in Figure 2.7 below.

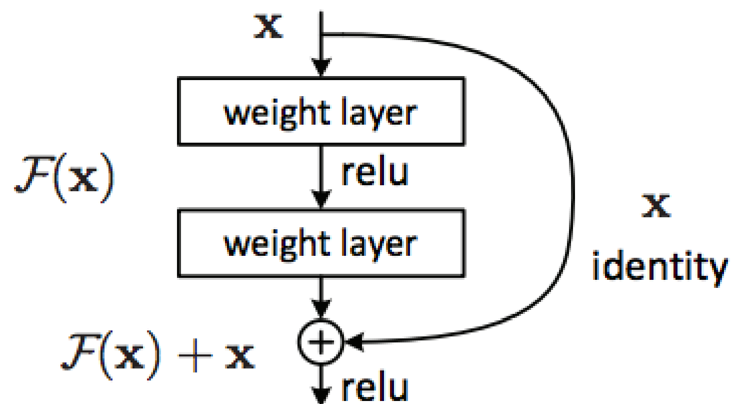


Figure 2.7: A residual learning block [20]

To summarize, the residual function is F(x) = H(x) − x and rather than learning H(x) straight away, the network learns H(x) = F(x) + x. The shortcut connections in ResNet architecture are similar to that present in highway networks [53, 52]. The only difference being, the shortcut connections in highway networks contain gating functions which are data-dependent [21]. On the contrary, the shortcut connections in Residual networks are data-independent and allow to pass most of the information from the initial layers to the final layers without degradation. Consequently, the skip connections also permit errors signals to travel directly through them. Moreover, shortcut identity connections do not add any extra parameter or increase computational complexity. To conclude, we can add several layers without facing issues like degradation and vanishing gradient. Therefore, the ResNet architecture that won the ImageNet 2015 Challenge comprises 152 layers trained on 60 million parameters in comparison to VGGNet which consists of 19 layers for 168 million parameters [20].

2.2.2 MobileNet

The state-of-the-art architectures described above are quite deep with up-to 152 layers and therefore require a considerable amount of memory and inference time. Also, most of the CNN needs high-end GPU devices for their training and inference. Moreover, as the network gets deeper and deeper, it demands more and more memory and computation, limiting their operation on even fewer devices. Due to the above reasons, it becomes difficult to run deep network models directly on mobile devices, which is an extensive market nowadays [29, 45]. The drawbacks of the deeper networks motivated the researchers to search for solutions appropriate for mobile and embedded vision applications.

A group of researchers at Google proposed MobileNet, a family of architectures which encapsulates the CNN that runs efficiently on mobile devices [24]. The authors claim that the accuracy of MobileNet is comparable to VGG16 although it is 32 times smaller than VGG16. The principle idea behind MobileNet is that it uses depthwise separable convolution to build quite small, low-latency deep neural networks.

In standard CNN, all the convolutional layers deploy regular convolution. Regular convolution means filtering all the input channels and combining them into a single output channel in one step as shown in Figure 2.8 below. However, in MobileNet architecture, the first layer still uses standard convolution while all the layers use depthwise separable convolution.

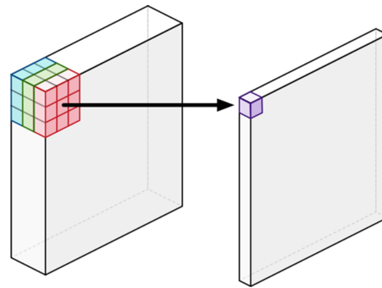
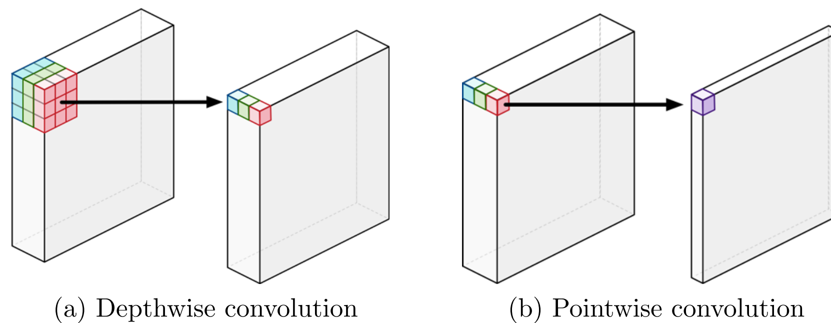


Figure 2.8: An example of regular convolution[3]

The depthwise separable convolution is factorized into two different operations of convolution: a depthwise convolution and a pointwise convolution. Unlike regular CNN, MobileNet filters the input channels in depthwise convolution and then combines them in pointwise convolution step. The Figure 2.9 below shows two different steps of the depthwise convolution and the pointwise convolution.



(a) Depthwise convolution (b) Pointwise convolution

Figure 2.9: Two different steps of depthwise separable convolution[4]

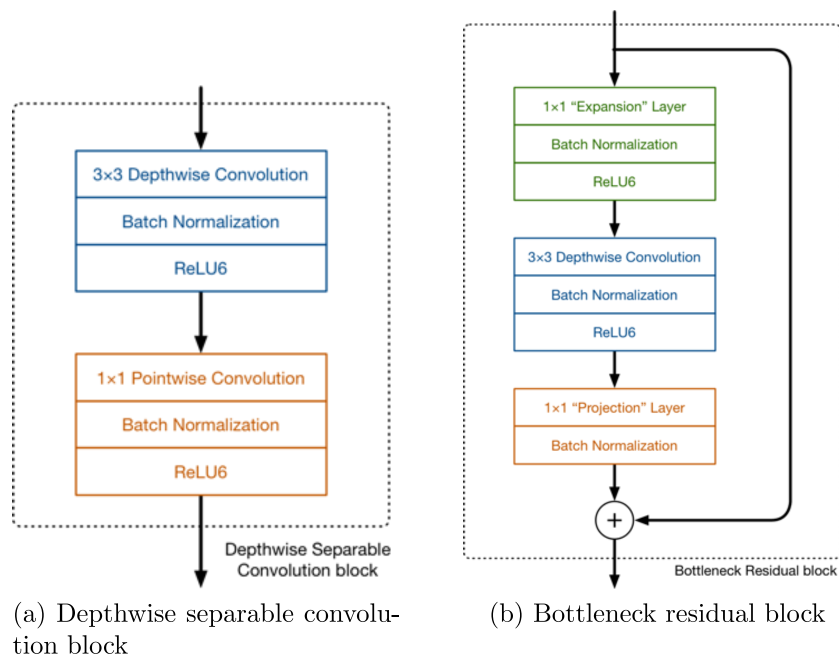
The factorization process described above makes it possible to reduce the computation and size of the model substantially. The full MobileNet version 1 (v1) network has 30 layers. Though the size of MobileNet is pretty small as compared to other deep networks, it further can be shrunk by tuning hyperparameters such as width multiplier and resolution multiplier. The hyperparameter tuning makes the networks smaller, and faster but at the expense of prediction accuracy [24].

[3] http://machinethink.net/blog/googles-mobile-net-architecture-on-iphone

[4] http://machinethink.net/blog/googles-mobile-net-architecture-on-iphone

MobileNet-v2

In 2018, a group of researchers at Google released version 2 (v2) of MobileNet [48], whose architecture is somewhat different from the one mentioned above. Instead of two convolution layers in one block as in version 1, version 2 now has three convolutional layers as shown below in Figure 2.10. The function of the depthwise convolution layer is the same as for filtering the input channels. Projection layer is similar to the pointwise convolution layer; however, the name and operation differ from the former one. Also, the first layer, i.e., the expansion layer is entirely new.



(a) Depthwise separable convolution block

(b) Bottleneck residual block

Figure 2.10: Block diagram of MobileNet-v1 and MobileNet-v2[5]

The principle contribution behind the block is the inverted residual with the linear bottleneck. The expansion layer takes a low-dimensional compressed input and expands it to a higher dimension based on an expansion factor. The depthwise convolution layer then filters the high-dimensional output of the previous layer as in version 1. Finally, the filtered output is projected back to a much lower number of dimensions. The projection layer is also known as linear bottleneck layer. Another important aspect is the residual connection. The motive behind the residual connection is similar to that of the shortcut connections in ResNets, that is, to propagate

[5] http://machinethink.net/blog/mobilenet-v2

the gradient back to the initial layers without degradation. However, the author has pointed out that the inverted design where shortcuts are inserted between the bottlenecks instead of expansion layers is considered more memory efficient [48].

The advantage of MobileNet-v2 over MobileNet-v1 is that the input fed to the tensors (connections between each convolution block) is of lower dimension. Low-dimension input means tensors now require comparatively less memory and computational resources but, having low-dimensional tensors can be worse too. Filtering a lower dimensional tensor might not give all the useful information. However, the expansion layer takes care of that by expanding the input fed by tensors before the filtering step. Figure 2.11 below compares MobileNet-v1 with MobileNet-v2 regarding accuracy and latency.

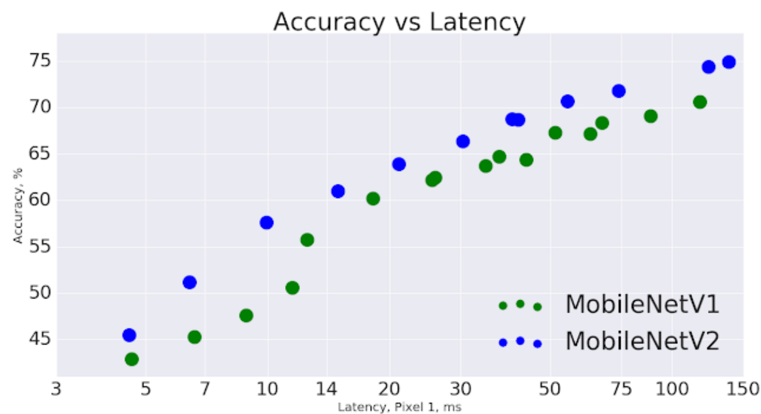
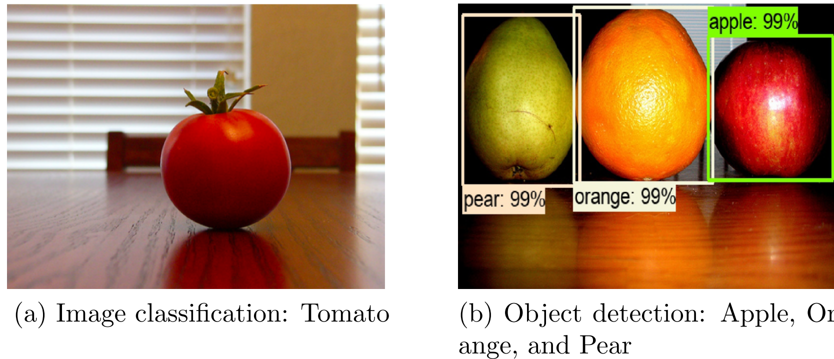


Figure 2.11: Comparison of MobileNet-v1 vs. v2[6]

2.3 Object Detection Techniques

As mentioned before in Chapter 1, object detection is not only about recognizing or classifying objects in an image, but also about localizing them and drawing bounding boxes around them. In other words, object detection is an extension of image recognition. For instance, on the left-hand side in Figure 2.12 below, image classification classifies it as belonging to one particular class (i.e., tomato), while on the right-hand side in Figure 2.12, object detection recognizes three objects apple, pear, and orange and localizes them by drawing bounding boxes around them.

[6]https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html



(a) Image classification: Tomato

(b) Object detection: Apple, Orange, and Pear

Figure 2.12: Difference between image classification and object detection

The object detection method mainly consists of three steps:

- In the first step, region proposals are generated from an input image. Region proposals, also known as regions of interest (RoI), are a large set of bounding boxes produced by scanning the whole input image.
- In the next step, visual features are extracted given the region proposals and these features help to detect objects in an image.
- In the final step of non-maximum suppression, highly-overlapping boxes are grouped into a single box.

Most of the modern object detection methods are based on the deep convolutional neural networks (CNN). Some of the most popular object detection models are Faster Region-based Convolutional Neural Network (Faster R-CNN), Region-based Fully Convolutional Network (R-FCN), and Multibox Single Shot Detector (SSD). Faster R-CNN is the successor of Fast R-CNN, and R-CNN is the predecessor of Fast R-CNN. In this section, we discuss the various object detection techniques along with their architectures.

2.3.1 Region-based Convolutional Neural Network (R-CNN)

In 2014, Girshick et al. [16] leveraged the advantages of the convolutional neural network to devise a network for object detection referred to as Region-based convolutional neural network or R-CNN. Training a network with R-CNN involves the following three steps.

- Firstly, around 2,000 region proposals are extracted by scanning an input image with an algorithm known as Selective Search. All the proposed regions are warped to the same size. Since the size of regions generated by selective search varies and CNN accepts a fixed size input, these regions are warped (scaled) to a fixed size required by the CNN.

- Secondly, features are computed for each of the proposed regions using an independent deep convolutional network.

- Lastly, based on the features derived by each CNN, multiple linear SVMs are deployed to classify regions and tighten bounding boxes.

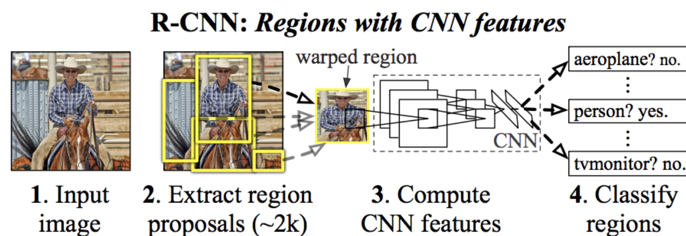


Figure 2.13: Region-based Convolutional Neural Network (R-CNN) [16]

Selective Search is an algorithm to generate region proposals [57]. The algorithm follows the three steps described below:

- At first, many small regions are generated using the fast method described by Felzenszwalb et al. [13]. Typically, at most one object resides in each of the generated regions.

- Next, apply a greedy algorithm to combine regions recursively.

- The regions produced in the second step are used to derive the final candidate object proposals.

To train a network with R-CNN required immense annotated datasets, but the datasets available at that time for object detection were scarce. To overcome the problem of scarcity of datasets, Girshick et al. [16] pre-trained the CNN described above with the ImageNet 2012 classification dataset that has only image-level annotations

(no bounding boxes data). Then, the network is fine-tuned for two different datasets, PASCAL VOC 2012 [12] dataset, and ImageNet 2013 dataset, with bounding boxes.

The results of the approach embraced above were quite positive. On PASCAL VOC 2012 dataset, R-CNN achieved mAP (Mean Average Precision) score of 62.4%, which is 22.0% more than the second best model. Similarly, on the ImageNet 2013 dataset, R-CNN achieved 31.4% mAP score, 7% more than the second best. Therefore, the approach adopted by the author, supervised pre-training on a large secondary dataset, followed by domain-specific fine-tuning improved the performance considerably [16].

2.3.2 Fast Region-based Convolutional Neural Network (Fast R-CNN)

As discussed above, the training process for R-CNN is multi-stage. First, a CNN is fine-tuned for each of the 2,000 region proposals; then multiple class-specific SVMs are used to classify objects followed by linear regressors to adjust bounding boxes. Also, the three-stage training makes the process time-consuming and expensive. Additionally, it takes 47 seconds with VGG16 (on one GPU) to detect objects for one test image, which makes R-CNN unsuitable for object detection in real-time applications. To improve upon the training process of R-CNN, Girshick [15] came up with a faster object detection algorithm in 2015 known as Fast R-CNN.

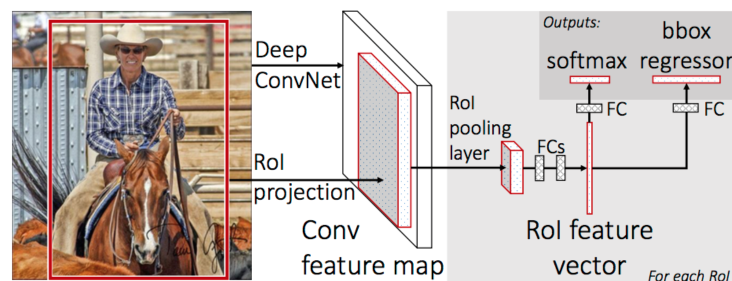


Figure 2.14: Fast region-based Convolutional Neural Network (Fast R-CNN) [15]

The architecture diagram of Fast R-CNN is shown above in Figure 2.14. The working of the network is explained below:

- In Fast R-CNN, an input image is fed to a single CNN with multiple convolutional layers to generate a convolution feature map.

- The regions of interest are still created with the Selective Search algorithm and provided as an input to Fast R-CNN. Each of the region proposals is warped and fed into a RoI pooling layer.
- RoI pooling layer produces a fixed-length feature vector from the feature map for each of the object proposals and passes it as an input to a series of fully connected layers.
- The fully connected layers are finally divided into two branches: the softmax classifier predicts the class of the proposed region and the output of regression provides the four offset values for adjusting the bounding boxes.

The advantages of Fast R-CNN over R-CNN are as follows:

- Fast R-CNN involves training only one CNN for an entire image instead of training multiple CNNs for all the region proposals generated for an image.
- Multiple class-specific SVMs are replaced by single softmax classifier, extending the neural network without training a different module.
- The above factors make the training single-stage and faster as compared to R-CNN. The testing speed also improves considerably. The author claims that the speed of Fast R-CNN for training the VGG16 network has improved nine times than the speed of R-CNN. Likewise, the test speed for Fast R-CNN has increased by 213 times as compared to that of R-CNN [15].
- Moreover, Fast R-CNN achieved the mAP score of 68.4% for the PASCAL VOC 2012 dataset improving upon the detection accuracy [15].

2.3.3 Faster Region-based Convolutional Neural Network (Faster R-CNN)

Fast R-CNN advanced the training and testing time, but the Selective Search algorithm used to generate region proposals is still one of the major bottlenecks of these networks. Selective Search is a slow and computationally expensive process to produce region proposals. Therefore, in 2015 Ren et al. [46] replaced the Selective Search with Region Proposal Network (RPN) in an object detection algorithm known as Faster R-CNN. Region proposal network, a deep fully convolutional network is used

to generate a fixed number of region proposals. Faster R-CNN unifies RPN with Fast R-CNN to detect objects.

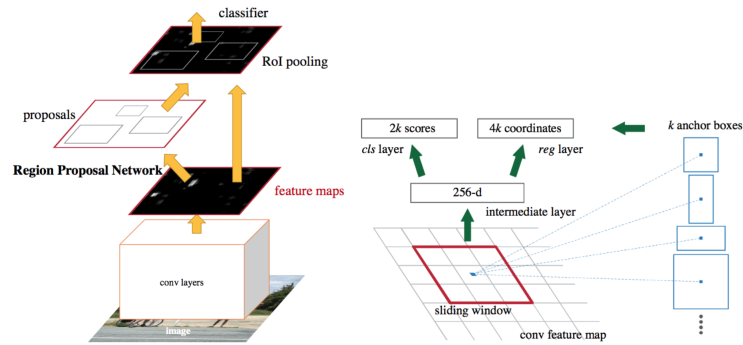


Figure 2.15: Faster region-based Convolutional Neural Network (Faster R-CNN) [46]

The network used to generate region proposals is shown in Figure 2.15 on the right-hand side, and the working is described below:

- Similar to Fast R-CNN, an input image is fed to a CNN to generate convolutional feature map.
- A filter/window of size nxn, is slid over the feature map generated by the last convolution layer and maps it to a lower-dimensional (256-d) feature vector.
- The output feature vector is connected to two fully connected layers, one for classification and one for regression. For each sliding window location, RPN generates a set of k region proposals with different aspect ratios. The k regions proposed by sliding window are known as anchor boxes. Each anchor box is associated with an objectness score of the box and four coordinates of the bounding box. Objectness score classify anchor boxes into object classes or background.

Since RPN generates a maximum of k regions for each location, the output of the classification layer is 2k scores (to predict the probability whether each region is an object or not) and the output of the regression layer is 4k (corresponding to the four coordinates of each region). After detecting all the anchor boxes, the relevant region proposals are selected by applying a threshold to the objectness score. In essence, the regions whose objectness score is above a certain threshold are chosen and passed as an input to Fast R-CNN detector along with convolutional feature map.

Since RPN shares the feature map generated by CNN, the region proposal step is almost free of cost. Secondly, Faster R-CNN achieved the mAP score of 75.9% for the PASCAL VOC 2012 dataset, around 7% more than that of Fast R-CNN. Lastly, in the ImageNet Challenge and the COCO 2015 competition, RPN and Faster R-CNN are the backbones of various approaches that won in these events [46].

2.3.4 Region-based Fully Convolutional Network (R-FCN)

Fast R-CNN and Faster R-CNN, the object detection algorithms discussed above share feature maps generated by CNN for object classification and region proposal. However, Fast R-CNN and Faster R-CNN still make use of fully connected layers, which are computed over and over again hundreds of times for each region proposal. The unshared computations make them slow and expensive as compared to the object detection algorithm proposed by Dai et al. [7] in 2016, which is mostly shared fully convolutional network. The algorithm is known as Region-based Fully Convolutional Network (R-FCN).

The principle idea behind the R-FCN is positive-sensitive score maps. The positive-sensitive score maps were introduced to compromise between translation-invariance in image classification vs. translation-variance in object detection. In other words, image classification is in-discriminative of location variance, whereas object detection wants to learn location variance. The score maps were used to address the dilemma between location invariance and location variance.

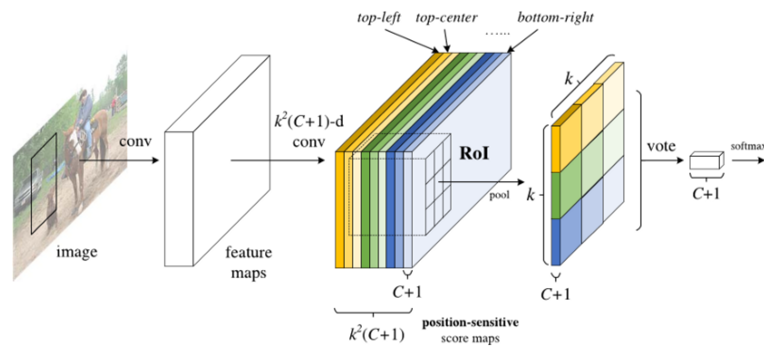


Figure 2.16: Region-based fully Convolutional Network (R-FCN) [7]

The architecture of R-FCN is shown in Figure 2.16 above. The working of R-FCN is discussed below:

- Similar to Fast R-CNN and Faster R-CNN, an input image is fed to a convolutional neural network to generate feature maps. Also, region proposals are generated using Region Proposal Network (RPN).
- Next, a fully convolutional layer is added which takes the features maps as an input and produces positive-sensitive score maps. Positive-sensitive score maps are the convolutional feature maps that have learned to detect specific location of each class object. There are a total of $k^2(C+1)$ score maps, $(C+1)$ corresponds to the number of classes + background, and k^2 represents the size of the grid used to divide an object. All the score maps constitute a score bank. For instance, there are eight classes in our dataset, and each class object is divided into nine sub-regions with a 3x3 grid. Then, there is a total of $3^2(8+1) = 81$ score maps.
- Then, each region of interest (RoI) is divided into the same kxk grid, i.e., each RoI has k^2 sub-regions (bins). For each bin, inspect the score bank to map it to the corresponding bin of one object. Average the score map values in the RoI region. Repeat this for all the sub-regions to create a vote for each class. This process of creating vote array by mapping RoI with score maps is known as position-sensitive RoI pooling.
- The process mentioned above is repeated for all the classes + background to produce vote for all the $(C+1)$s classes. Afterward, average the bin values for each class to get a score per class. A score vector is created to represent scores of all the classes.
- Lastly, apply the softmax classifier to classify the RoI based on $(C+1)$ score vector created in the previous step.

As evident from the architecture shown above, R-FCN is fully convolutional and maximizes the shared computation. Thereby, R-FCN is faster than Fast R-CNN and Faster R-CNN and achieves comparable performance [7].

2.3.5 Single-Shot Multibox Detector (SSD)

All the region-based object detection algorithms discussed work in two stages: first, generate region proposals and then classify those regions in a separate step. Single-Shot Detector, an object detection algorithm proposed by Liu et al. [38] in 2016

predicts the bounding box (region) and the class simultaneously in a single shot. Doing both the operations in one step makes SSD faster and a viable option for real-time detections.

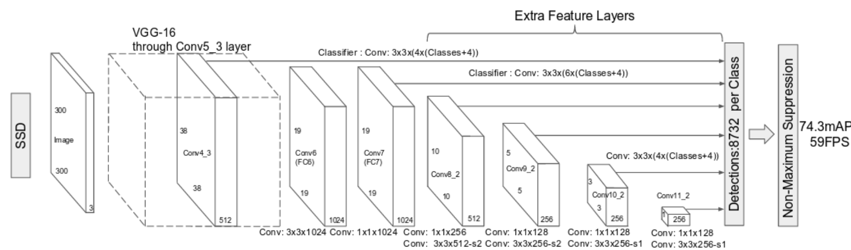


Figure 2.17: Single-Shot Detector (SSD) [38]

The architecture of SSD is shown in Figure 2.17. The working is described below:

- For the base network, the author has chosen the truncated version of VGG16, which is an image classification architecture. VGG16 is truncated to remove the classification part and extended with extra feature layers with sizes in decreasing order. An input image passes through all the convolution layers to generate multiple convolutional feature maps of different sizes.

- Then, for each of the feature maps produced above, anchor boxes are generated in a similar way as for Faster R-CNN. A convolution filter of size say, 3x3 is slid over each feature map to create anchor boxes at different scales and aspect ratios. Each predicted box has four coordinates and a vector of class probabilities.

- At the training time, all the predicted boxes are matched with the ground truth boxes based on IoU (Intersection over Union). As a result, all the boxes with IoU > 0.5 are considered as positives, and others as negatives.

- Since there are multiple feature maps of different sizes, the number of bounding boxes generated are quite large. Also, most of the anchor boxes are negatives.

- To overcome the shortcomings mentioned above, firstly non-maximum suppression is deployed at the end to combine the overlapping boxes. Secondly, the technique of Hard Negative Mining (HNM) is used to balance the negative and positive training examples. In training, only a subpart of negative examples is

used. The subpart is created by sorting all the negatives by the confidence loss and selecting the top ones such that the ratio of negatives to positives is 3:1.

To conclude, SSD predicts the bounding boxes and class probabilities in one shot, which makes it faster than the region-based algorithms discussed above. Besides, accuracy is comparable to Faster R-CNN. Moreover, the variable sizes of the feature maps assist in generating boxes of varying dimensions, which in turn helps detect objects of different sizes [38].

2.4 Open Source Frameworks for Object Detection

Open source frameworks for object detection, such as Google's Tensorflow Object Detection API, OpenCV's DNN library, and Microsoft Cognitive Toolkit (CNTK), make it possible to create our application-specific object detectors. The open source frameworks also provide pre-trained models for object detection which can be easily fine-tuned for our datasets.

The most interesting feature of these open source frameworks is that they provide pre-trained models to initiate the process of fine-tuning. This process of using a pre-trained model as a basis of fine-tuning is known as transfer learning. Microsoft Cognitive Toolkit [49], an open source framework for designing and training deep convolutional neural networks, provides several pre-trained models for image classification trained on the ImageNet database and the CIFAR-10 dataset [32]. Additionally, for object detection, CNTK presents two Fast R-CNN pre-trained models trained on PASCAL VOC 2007 and a grocery dataset.[7] Moreover, CNTK presents the recipes of how to create a custom object detector deploying Fast R-CNN or Faster R-CNN in conjunction with pre-trained image classification models. However, CNTK currently does not support ResNet as a base models for deploying object detectors.[8] In addition, CNTK does not even support SSD with MobileNet-v1/v2, an object detection model suitable for mobile devices.

Google's Tensorflow provides various pre-trained models for image classification trained on the ImageNet database and also presents several pre-trained models for ob-

[7] https://github.com/Microsoft/CNTK/blob/master/PretrainedModels/download_model.py

[8] https://docs.microsoft.com/en-us/cognitive-toolkit/Object-Detection-using-Fast-R-CNN#using-a-different-base-model

ject detection trained on datasets such as COCO, Kitti, and Open Images. Tensorflow Object Detection API supports object detection models such as SSD, Faster R-CNN, and R-FCN in association with image classifiers including MobileNet-v1, MobileNet-v2, Inception-v2, and ResNet-101. The main advantage of using Tensorflow is that it supports SSD model with MobileNet-v1/v2 which enables a user to create an object detector suitable for mobile devices and embedded vision applications.

Therefore, to create an object detector for a refrigerator, we have used Tensorflow Object Detection API due to its popularity, ease of use, and well-documented code base. The following section describes Tensorflow Object Detection API in detail.

2.4.1 Tensorflow Object Detection API

Google's Tensorflow Object Detection API is an open source framework for object detection; it is based on Tensorflow and allows a user to define, train and utilize the models for object detection. The Tensorflow open source software library, developed by Google Brain Team, utilizes data flow graphs for numerical computations [1]. Data flow graphs consist of two important components, nodes, and edges. Nodes represent the mathematical computations (operations), and edges represent tensors (multi-dimensional arrays) that flow between the nodes. Tensorflow is widely used to develop applications with deep learning [44, 17]. Another interesting feature of Tensorflow is that it supports a tool known as Tensorboard for in-depth visualization of models during the training process. Tensorboard provides a web interface for visualization of computational graphs and to understand how the parameters and performance change while training a model.

As mentioned earlier, Tensorflow Object Detection API provides multiple pre-trained models such as SSD model with MobileNet, Faster R-CNN model with ResNet, and R-FCN model with ResNet on different datasets. To fine-tune our specific object detectors, we can choose among the pre-trained models to initialize our training. The selection of pre-trained model is based on the purpose of our application. The API also gives an insight into speed and accuracy trade-offs of different object detection models [9].

The most important requisite of modern object detection applications is to achieve real-time speed. The models trained with SSD and R-FCN networks are pretty fast,

[9] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

but this speed comes at the cost of reduced accuracy. On the contrary, models trained with Faster R-CNN are more accurate but are expensive in time [25]. The speed-accuracy trade-off for different object detection models is shown in Figure 2.18 below:

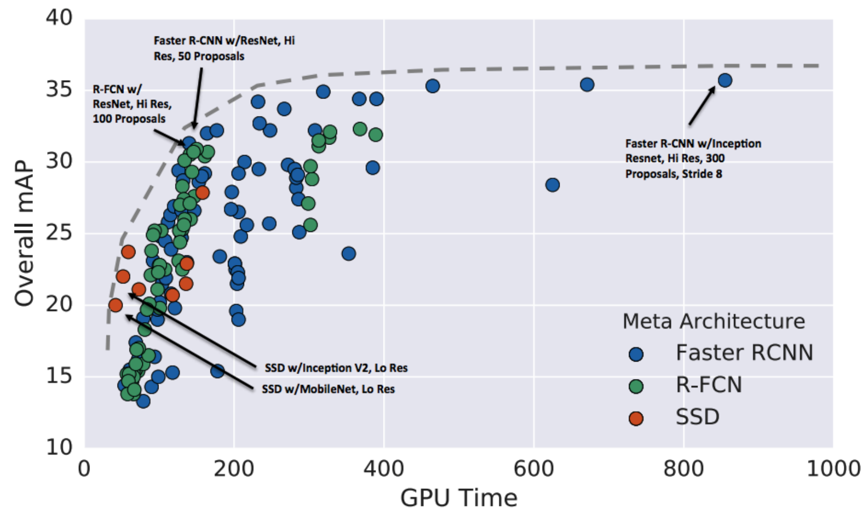


Figure 2.18: mAP vs. GPU time for different meta-architectures [25]

Figure 2.18 shows that Faster-RCNN model with Inception ResNet with 300 proposals is the most accurate of all; but also the most expensive. However, the models trained with SSD and R-FCN are faster but less accurate. Moreover, Faster R-CNN can be fast enough, if the regions of interest are limited as evident for Faster R-CNN with ResNet with 50 proposals.

2.5 Summary

This chapter described the related work in the field of computer vision for tasks related to visual computing such as image recognition and object detection. We introduced the concepts of an artificial neural network (ANN), deep neural network and deep learning, and convolutional neural network (CNN). We also described the method to train CNN as these networks are extensively used for the task of image classification and object detection. We explained in detail the contemporary image classification and object detection architectures along with their performance evolution. We then described the open source frameworks available for object detection

and speed-accuracy trade-off for various object detection models. In the next chapter, we discuss the implementation details of object detection models using Tensorflow Object Detection API.

Chapter 3

Implementation of Object Detection using Tensorflow

One of the main objectives of our thesis is to train three different object detectors and infer which one works best for detecting the eight classes of our interest. For this purpose, it is important to select appropriate models of object detection and also a dataset that can be used for training and evaluation purposes. This chapter starts with a description of the dataset used throughout our research along with the pre-processing required on that dataset. Next, we explain the fine-tuning of the three Tensorflow pre-trained models of object detection. Lastly, we describe the evaluation metrics used all through the thesis.

3.1 Data Collection and Pre-processing

Typically one of the significant challenging aspects of a deep CNN is to collect labeled examples to train image classifiers. In general, supervised learning demands thousands of annotated training examples. To meet the requirement of large training data and benchmark evaluation, the number of publicly available datasets are proliferating. In the context of computer vision, the available datasets contain large sets of images with additional information such as annotations, segmentation masks or other contextual data. Since there are several such datasets available for training and validation, it is essential to select the one which serves our purposes best. We selected the ImageNet database because it contains many images for each class along with bounding box data for some images. Moreover, the ImageNet database has been

used successfully for various applications of image classification which is an integral part of object detection [33, 50, 55, 20].

To train and evaluate our object detection models, we first extracted the image and bounding box datasets for eight classes namely, *Apple, Bell pepper, Cauliflower, Lemon, Orange, Pear, Tomato and Turnip.* Some of the issues we encountered with the extracted datasets are as follows:

- The extracted datasets consist of images of all sizes. However, we want to make sure that the images used for training and evaluation are not too small (i.e., at least 300 pixels).

- Another problem with the extracted datasets is that the bounding box data of each class is labeled for that particular class only and not all possible classes. For instance, in the Apple dataset, as depicted in Figure 3.1 below, is labeled for only Apple, and not for Orange and Pear. However, the requirement is to have all the images annotated with all the possible classes, in our case, we have eight classes.

- Additionally, the number of bounding box examples expected for each class is insufficient. For the training, validation, and evaluation purposes, we require at least 620 bounding box examples for each class. However, the ImageNet database contains lesser examples of each class.

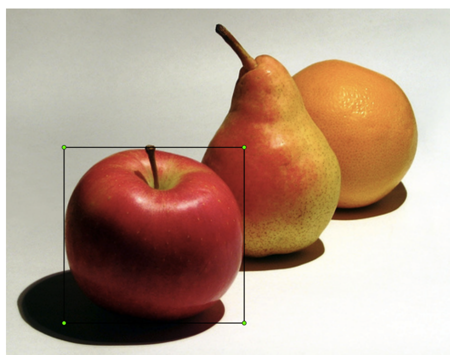


Figure 3.1: An image example from the Apple dataset of the ImageNet database

Keeping the above challenges in mind, we pre-processed the food datasets derived from the ImageNet database to create datasets suitable for training and validation purposes as follows :

- **Selection and Annotation of Datasets.** We selected and annotated around 620 examples for each class. The selection process is fairly simple but, the annotation part of manual labeling of images required a significant amount of time. To annotate the images, we used a tool called LabelImg,[1] which is a Python-based annotation tool for labeling graphical images. The annotations in the form of bounding box coordinates, and labels are saved as XML (Extensible Markup Language) files in PASCAL VOC (Visual Object Classes) or YOLO (You Only Look Once) format. The bounding box datasets of the ImageNet database are also in the form of XML files in PASCAL VOC format. Hence, we used the PASCAL VOC format to save the annotations using this tool.

- **Create training, validation, and test dataset.** After completing the annotation process, we divided each class dataset into three sets: training dataset, validation dataset, and test dataset. Each training dataset contains about 500 examples, and each validation and test dataset have around 60 samples. Then, the corresponding datasets for all the classes are merged—eight training datasets for eight classes. We then combined the eight training datasets into one training dataset with the training examples from all the classes. Similarly, we combined the validation and test datasets as well. Finally, we arrived at one training dataset with 4,019 examples, one validation dataset with 485 examples, and one test dataset with 488 examples.

- **Create TFRecords.** To fine-tune a pre-trained Tensorflow object detection model, the training and validation datasets are converted into a TFRecord format. TFRecord is a standard format, supported by Tensorflow for training object detection models.[2] Initially, the training and validation datasets are in XML files format; they are first converted into CSV (Comma Separated Values) file format, and then into a TFRecord format. The python command to convert a CSV file into a TFRecord file is given below[3]:

[1] https://github.com/tzutalin/labelImg
[2] https://www.tensorflow.org/api_guides/python/reading_data#file_formats
[3] https://github.com/datitran/raccoon_dataset/blob/master/generate_tfrecord.py

```
# Convert a training dataset csv file into training TFRecord
python generate_tfrecords.py
 --csv_input=path/to/train_labels.csv
 --output_path=path/to/train.record
```

Tensorflow models are pre-trained for multiple datasets such as COCO, Kitti, and Open Images. We selected three pre-trained models trained on the COCO dataset to fine-tune them with the images from the ImageNet database. The fine-tuning is vital for our application because the pre-trained models trained on the COCO dataset are only able to detect two out of eight classes, and even for those two classes, the performance is not great as demonstrated in Section 4.1.

3.2 Tensorflow Object Detection API for Fine-tuned Model

As stated earlier, the pre-trained models for object detection are not good at detecting all the eight classes required for our research. So, there is a need to fine-tune the pre-trained models with our dataset to be able to detect those eight classes effectively. The reason for fine-tuning a pre-trained model is that if a model is trained from scratch with randomly initialized weights, it might take weeks or months for the model to converge. Even then, the accuracy might not be good. This process of fine-tuning a pre-trained model is known as transfer learning, and it takes comparatively little time to train a model with transfer learning.

We selected three pre-trained models for object detection namely, SSD with MobileNet-v2, Faster R-CNN with ResNet-101, and R-FCN with ResNet-101. The three models are selected based on the literature review of object detection techniques described in Chapter 2. According to the discussion, among all the models SSD models exhibit best performance, while Faster R-CNN is the most accurate. The following section explains the fine-tuning of the three pre-trained models.

3.2.1 Object Detection with SSD-MobileNet-v2

The pre-trained SSD-MobileNet-v2 model (ssd_mobilenet_v2_coco) is fine-tuned with the training and validation datasets created earlier in Section 3.1. To begin, we are

using the provided configuration file (ssd_mobilenet_v2_coco.config) as the basis of our fine-tuning. The configuration file requires a checkpoint file to initiate the fine-tuning process which is already provided by Tensorflow for SSD-MobileNet-v2.[4] The configuration file also requires training record and test record file locations. Training record is a TFRecord file created for training dataset, and a test record is a file created for validation dataset. Another requisite of the configuration file is the location of an object label map file. The structure of an object label map file is shown in the following Figure 3.2, and ends with '.pbtxt' extension.

```
item {
name: "apple"
id: 1
display_name: "apple"
}
item {
name: "bell pepper"
id: 2
display_name: "bell pepper"
}
.
.
```

Figure 3.2: A sample of object label map file

The training record and test record file locations are added along with the object label map file as shown in Figure 3.3 below. The configuration file is also updated with the number of classes used for our model training. In our application, we have a total of eight food classes.

[4] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

```
train_input_reader: {
tf_record_input_reader {
input_path: "data/train.record"
}
label_map_path: "data/obj-det.pbtxt"
}

eval_config: {
num_examples: 485
num_visualizations: 50
max_evals: 485
}

eval_input_reader: {
tf_record_input_reader {
input_path: "data/test.record"
}
label_map_path: "data/obj-det.pbtxt"
shuffle: false
num_readers: 1
}
```

Figure 3.3: A sample of updated configuration file

Other important parameters mentioned in the configuration file include:

- number of layers: 6
- aspect ratios: 0.33, 0.5, 1.0, 2.0, and 3.0
- learning rate: initial value is 0.004 with a decay factor of 0.95 after 800,720 steps (iterations) of training
- data augmentation options: random_horizontal_flip and ssd_random_crop
- batch size: 24

After updating the configuration file according to our dataset, the Python commands to run the training and evaluation script are presented below:

```
# Command to train a model
python train.py --logtostderr
 --train_dir=path/to/train-directory
 --pipeline_config_path=path/to/config-file

# Command to evaluate a model while training
python eval.py --logtostderr
 --checkpoint_dir=path/to/train-directory
 --pipeline_config_path=path/to/config-file
 --eval_dir=path/to/eval-directory
```

The SSD model is fine-tuned for 80,000 steps as shown in Figures 3.4 and 3.5. Figure 3.4 shows the decline in total loss throughout the process of fine-tuning. The total loss mainly comprises classification loss and localization loss. Classification loss is based on how good an object is classified; localization loss is based on how well a correctly classified object is localized. The total loss values are added to the training log files after a particular interval of time. The log files are then passed as input to Tensorboard, a visualization tool to plot the computational graphs. All the plots presented below are captured from Tensorboard visualizations. The Python command to visualize log files on Tensorboard is given below[5]:

```
# Pass the path of directory of log files as a paramter
tensorboard --logdir=path/to/log-directory
```

As observed in Figure 3.4, the total loss value rapidly decreases since we have started our training from a pre-trained checkpoint file instead of starting from scratch. The total loss values fluctuate but exhibit reducing behavior overall. It should be noted that the values plotted in faded orange are the actual values of total loss, while the darker orange is obtained after smoothing of 0.6.

[5]https://www.tensorflow.org/guide/summaries_and_tensorboard

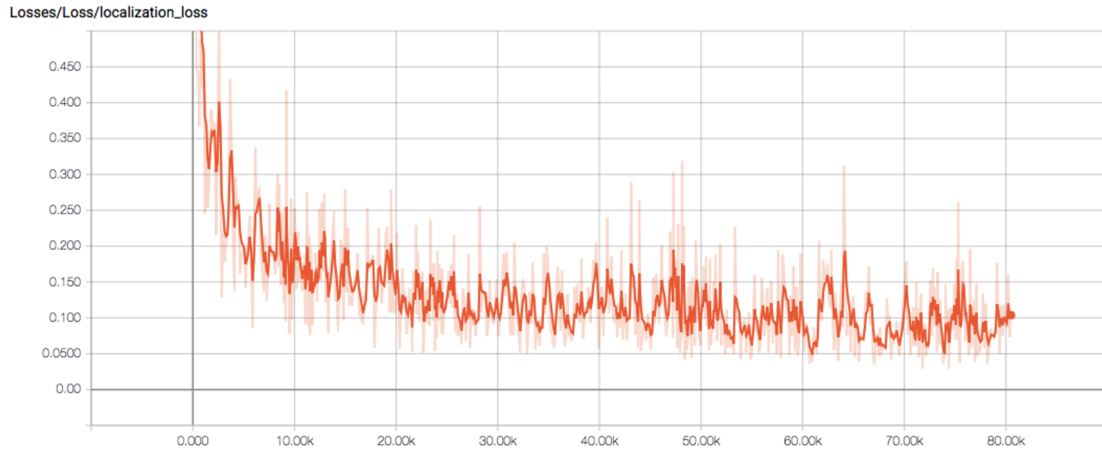


Figure 3.4: Decline of total loss when fine-tuning SSD-MobileNet-v2

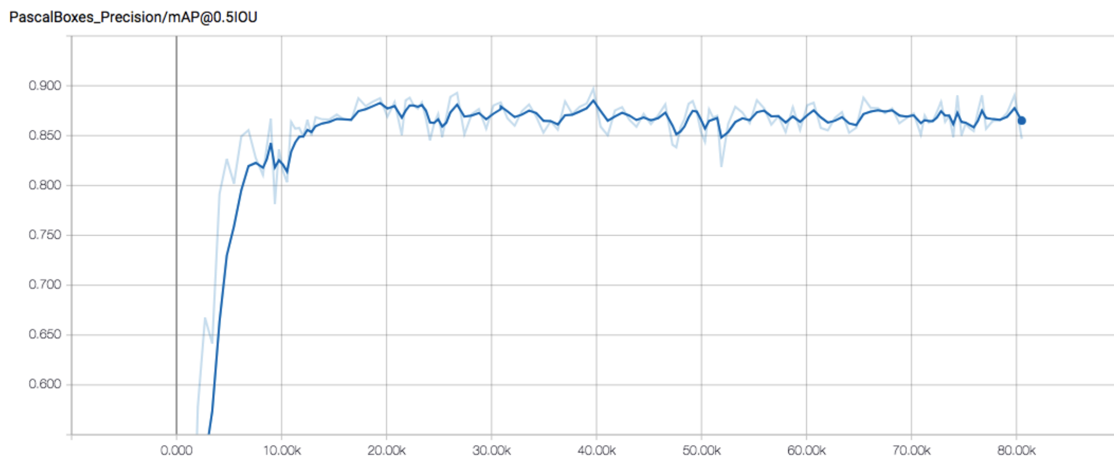


Figure 3.5: Development of overall mAP when fine-tuning SSD-MobileNet-v2

Figure 3.5 shows the development of overall mAP at 0.5 IoU for 80,000 steps. The mAP values are evaluated for the validation dataset at 0.5 IoU. As evident from Figure 3.5, mAP witnesses a tremendous increase until 85.5% in around 6,800 steps. After that, the mAP increases slightly more and reaches closer to 88% in 19,000 steps. After 19,000 steps, the mAP value remains almost stable with minor fluctuations. Since the mAP values leveled out after 19,000 steps, the model is exported at different steps of training and tested on the test dataset. The Python command to export the inference graph is given below:

```
# Export inference graph in the output directory
python export_inference_graph.py
 --input_type image_tensor
 --pipeline_config_path path/to/pipeline-config-file
 --trained_checkpoint_prefix path/to/train/model.ckpt-xxxx
 --output_directory path/to/output-directory
```

Changing the parameters in the configuration file to fine-tune the model

The SSD model is fine-tuned with the provided configuration file earlier. However, a few parameters can be modified in the configuration file to verify if there is a different configuration setting that works better in our case. For this purpose, we implemented the following three minor changes:

1. Increased the initial learning rate from 0.004 to 0.005. The learning rate is one of the most important hyperparameters that assists in the convergence of models. We modified the value of the learning rate to check if the model converges at an earlier stage and with better performance.

2. Decreased the initial learning rate from 0.004 to 0.003. The learning rate is decreased to make sure it is not set higher initially. The learning rate is reduced to make sure that we are not missing any local minima and the model converges properly.

3. Fine-tuned the model with only three different aspect ratios for anchor boxes, 0.5, 1.0 and 2.0 as opposed to the original configuration file which has five aspect ratios. This will generate fewer anchor boxes, and since all eight objects are mostly circular, we might not need aspect ratios of 0.33 and 3.0.

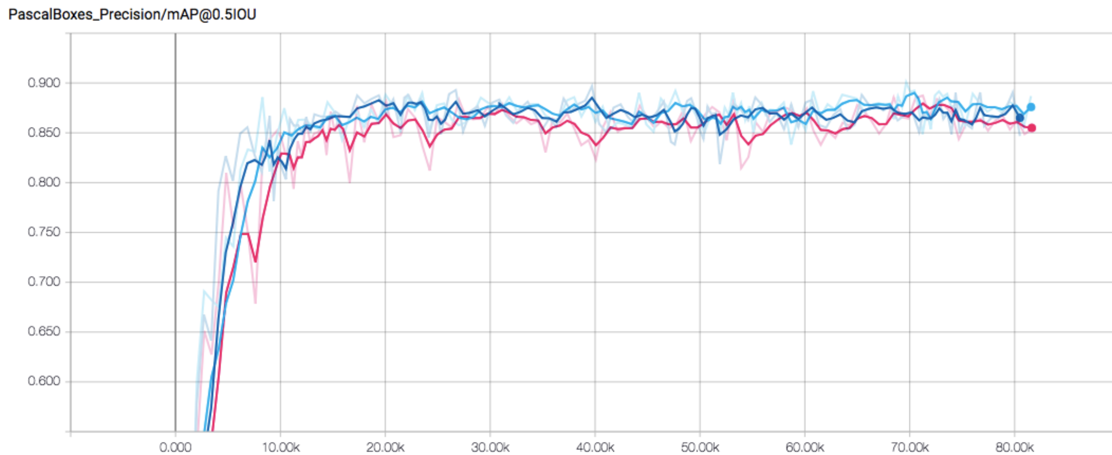


Figure 3.6: Development of overall mAP when fine-tuning SSD-MobileNet-v2 (blue= with learning rate 0.004 (the default one), light blue= with learning rate 0.003, pink= with learning rate 0.005)

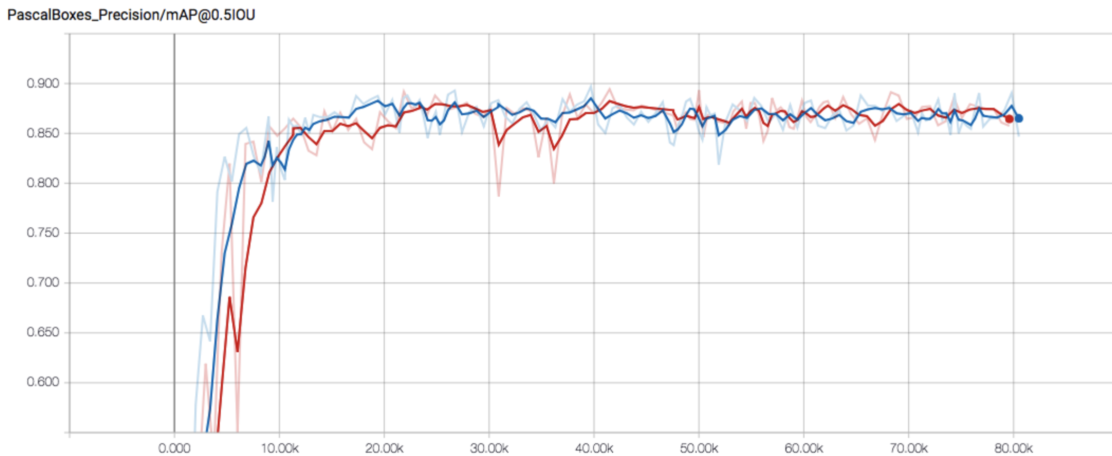


Figure 3.7: Development of overall mAP when fine-tuning SSD-MobileNet-v2 (blue= with default aspect ratios, red= with modified aspect ratios)

As observed from Figures 3.6 and 3.7, the mAP development is almost similar for all the configurations. However, when tested on the test dataset, the model fine-tuned with the provided configuration file outperforms the other configuration variations (cf. Section 4.2).

3.2.2 Object Detection with FRCNN-ResNet-101

In this section, the pre-trained model of FRCNN -ResNet-101 (faster_rcnn_resnet101_coco) is fine-tuned with the training and validation datasets created before. We selected the provided configuration file as a basis similar to fine-tuning of SSD-MobileNet-v2. In the same way, we utilized the checkpoint file to start the process, added locations of training and test record, and object label map, and updated the number of classes.

Other important parameters mentioned in the configuration file of FRCNN-ResNet-101 are as follows:

- scales: 0.25, 0.5, 1.0, and 2.0
- aspect ratios: 0.5, 1.0, and 2.0
- first stage maximum proposals: 300
- learning rate: initial value is 0.0003, reduce to 0.00003 after step 900,000 and further reduce to 0.000003 at step 1,200,000
- data augmentation options: random_horizontal_flip
- batch size: 1

The Faster R-CNN model is fine-tuned for 100,000 steps as shown in Figures 3.8 and 3.9. Figure 3.8 shows the downward trend of total loss throughout the training process. The total loss values show a fluctuating trend as the batch size is 1, and a loss for each iteration can be slightly different from the previous iteration. However, the key point is that the loss values decreased overall during the course of training.

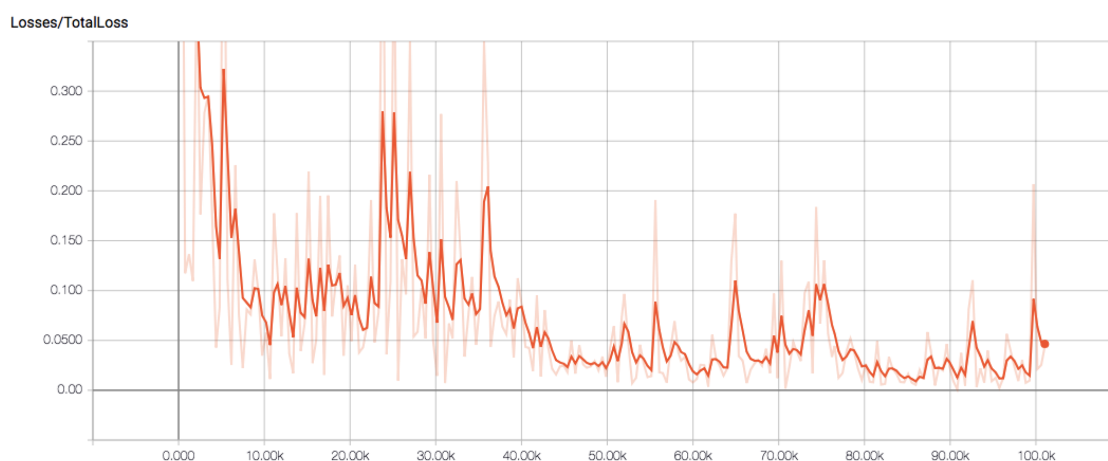


Figure 3.8: Decline of total loss when fine-tuning FRCNN-ResNet-101

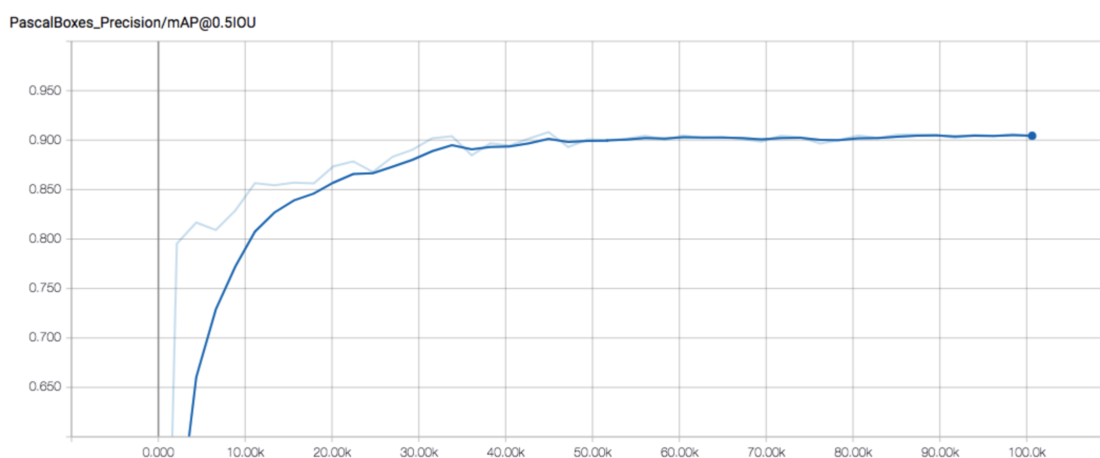


Figure 3.9: Development of overall mAP when fine-tuning FRCNN-ResNet-101

Similarly, Figure 3.9 shows the development of overall mAP at 0.5 IoU for 100,000 steps. As shown in Figure 3.9, mAP values increase until 90.4% in 33,820 steps and remain close to 90%, maximum being 90.8% at around step 45,000. The fine-tuned model is exported at different steps of training for evaluation on the test dataset.

Changing the parameters in the configuration file to fine-tune the model

The Faster R-CNN model is fine-tuned with the provided configuration file in the earlier section. However, similar to the fine-tuning of the SSD model, we experimented

with other configuration settings. For the fine-tuning of Faster R-CNN model, we modified the following parameters:

1. Updated the aspect ratios to 0.33, 0.5, 1.0, 2.0, and 3.0, while the default configuration file, has only three aspect ratios.

2. Reduced the number of proposals from 300 to 100. The number of proposals is reduced to check if on decreasing the proposals, the time of inference also decreases while achieving the comparable performance.

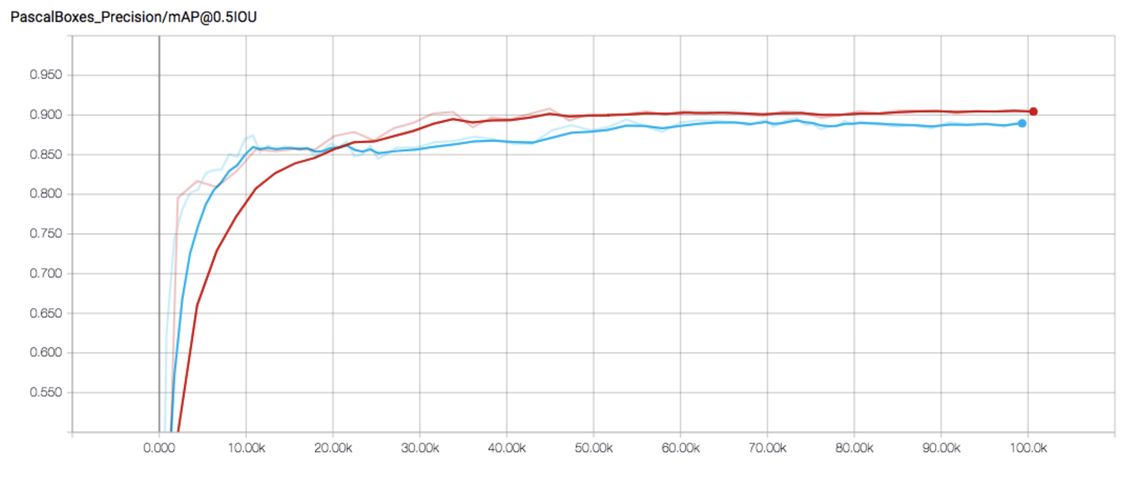


Figure 3.10: Development of overall mAP when fine-tuning FRCNN-ResNet-101 (red= with default aspect ratios, light blue= with modified aspect ratios)

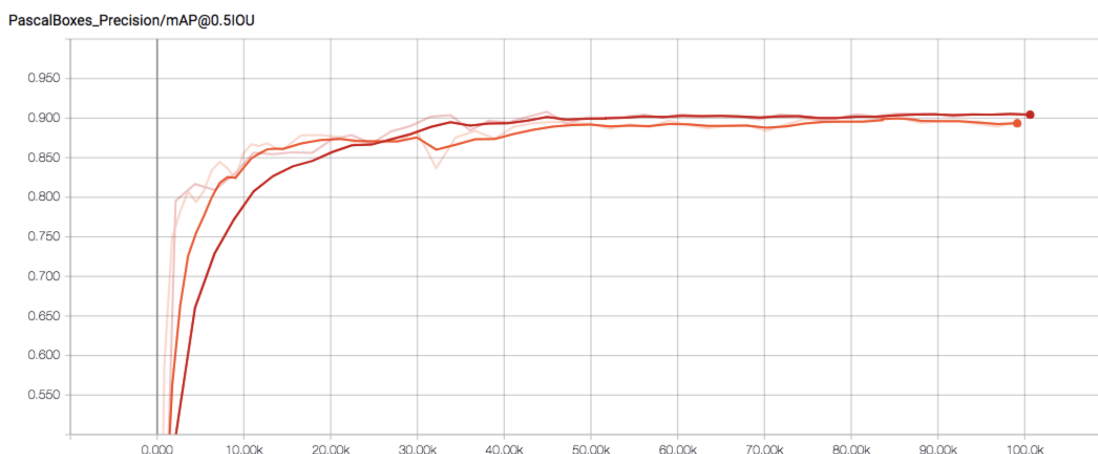


Figure 3.11: Development of overall mAP when fine-tuning FRCNN-ResNet-101 (red= with 300 number of proposals (the default one), orange= with 100 number of proposals)

As observed in Figures 3.10 and 3.11, throughout the training, the mAP values for the other two configuration settings are slightly less than those of the default configuration. Additionally, when tested on the test data, the model fine-tuned with the provided configuration file surpasses the other configuration variations.

3.2.3 Object Detection with RFCN-ResNet-101

This section discusses the fine-tuning of the RFCN-ResNet-101 (rfcn_resnet101_coco) pre-trained model. Like, the fine-tuning of SSD and Faster R-CNN model, R-FCN also uses the provided configuration file as a basis for fine-tuning. Similarly, we are using the checkpoint file to initiate the process, added locations of training and test record, and object label map, and updated the number of classes.

Other important parameters mentioned in the configuration file of RFCN-ResNet-101 are as follows:

- scales: 0.25, 0.5, 1.0, and 2.0
- aspect ratios: 0.5, 1.0, and 2.0
- first stage maximum proposals: 300
- learning rate: initial value is 0.0003, reduce to 0.00003 after step 900,000 and further reduce to 0.000003 at step 1,200,000

- data augmentation options: random_horizontal_flip
- batch size: 1

Similar to Faster R-CNN, the R-FCN model is also trained for 100,000 steps as depicted in Figures 3.12 and 3.13. Figure 3.12 shows the decay of total loss throughout the fine-tuning. The total loss showcases an overall reduction trend with minor fluctuations.

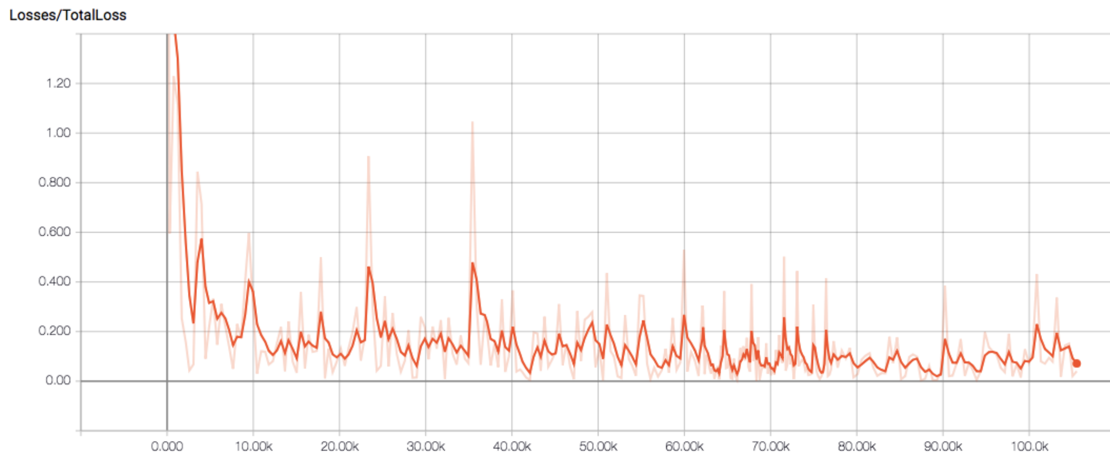


Figure 3.12: Decline of total loss when fine-tuning RFCN-ResNet-101

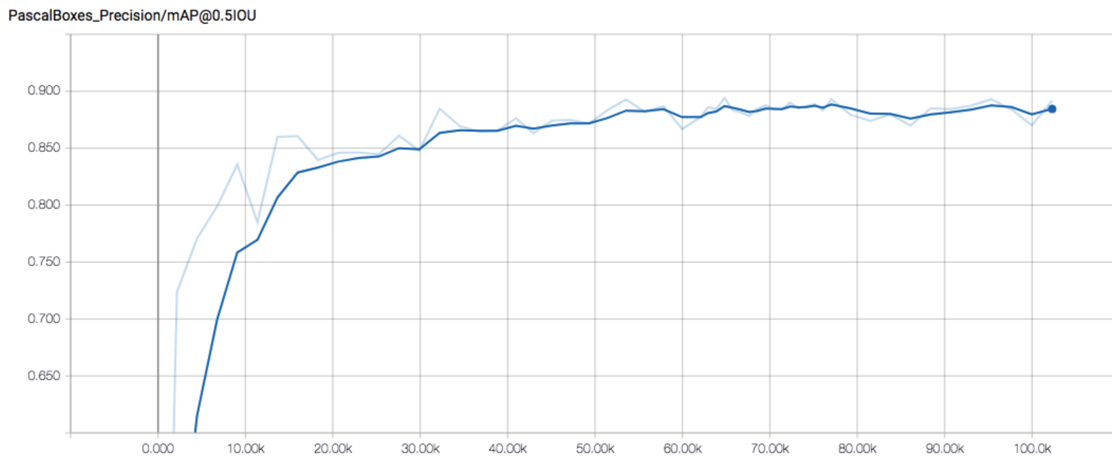


Figure 3.13: Development of overall mAP when fine-tuning RFCN-ResNet-101

Similarly, Figure 3.13 shows the development of overall mAP at 0.5 IoU for 100,000 steps. As shown in Figure 3.13, mAP values increase until 88.4% in 32,220 steps and remain close to 88.5%, maximum being 89.4% at around step 64,800. The fine-tuned model is exported at different steps of training for evaluation on the test dataset.

Changing the parameters in the configuration file to fine-tune the model

In the previous section, the R-FCN model is fine-tuned with the default configuration file. However, similar to the fine-tuning of SSD and Faster R-CNN model, we can modify the configuration settings to get better results. For the fine-tuning of R-FCN model, we changed the following parameters:

1. Updated the aspect ratios to 0.33, 0.5, 1.0, 2.0, and 3.0, whereas the default configuration file, has only three aspect ratios.
2. Reduced the number of proposals from 300 to 100.

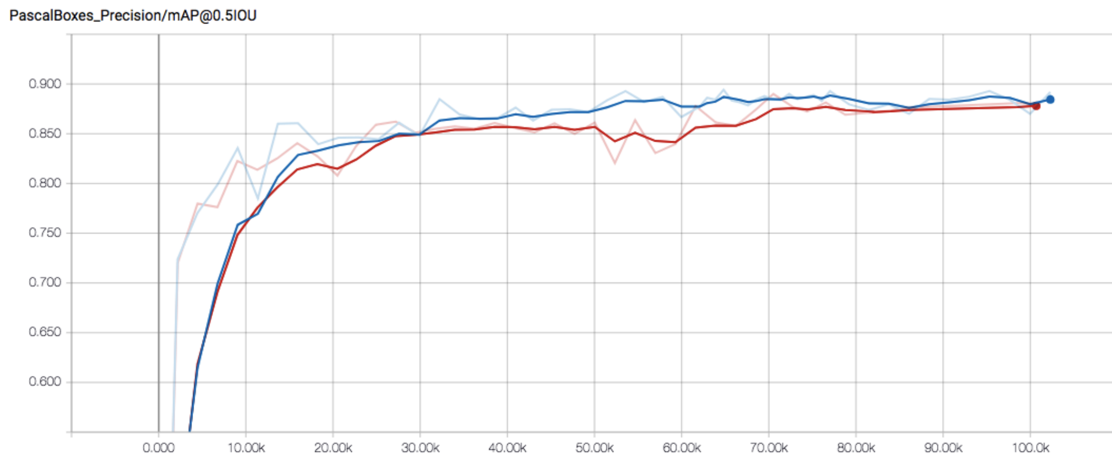


Figure 3.14: Development of overall mAP when fine-tuning RFCN-ResNet-101 (blue= with default aspect ratios, red= with modified aspect ratios)

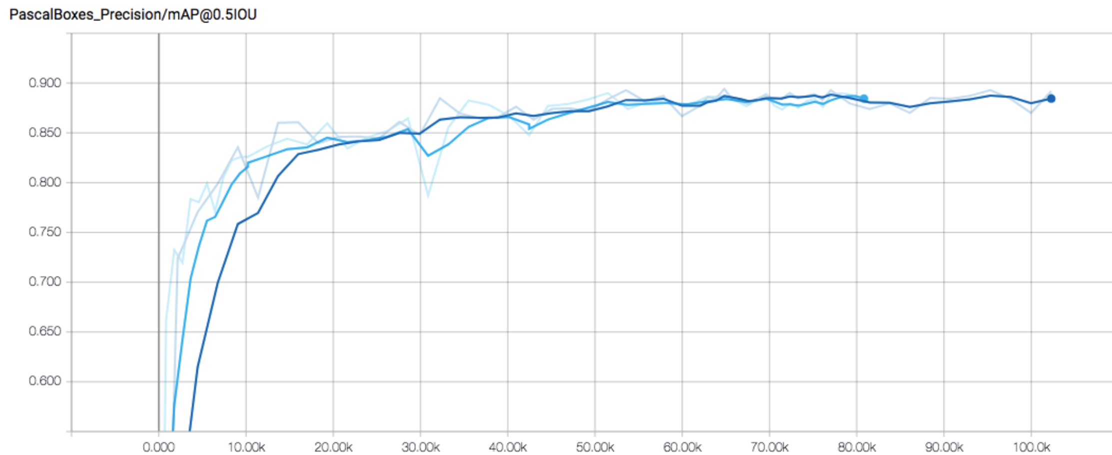


Figure 3.15: Development of overall mAP when fine-tuning RFCN-ResNet-101 (blue= with 300 number of proposals (the default one), light blue= with 100 number of proposals)

As observed in Figures 3.14 and 3.15, throughout the training, the mAP values for the other two configuration settings are slightly less than those of the default configuration. Moreover, when tested on the test dataset, the model fine-tuned with the default configuration file surpasses the other configuration variations.

3.3 Evaluation Metrics

The evaluation metrics used to evaluate performance in this thesis are Precision, Recall, Precision-Recall curves (PR curves), and Mean Average Precision (mAP). Precision, Recall, PR curves are relevant measures for binary classification. However, in this thesis, we are concerned with eight classes. So, precision, recall, and PR curves are calculated for each class, separately as opposed to all the other classes.

3.3.1 Precision

Precision measures how many instances are relevant out of all the retrieved instances.

$$
\begin{aligned}
Precision &= \frac{TruePositive}{TruePositive + FalsePositive} \\
&= \frac{TruePositive}{TotalPredictedPositive}
\end{aligned}
$$

3.3.2 Recall

Recall measures the ratio of retrieved instances that are relevant over all the relevant instances.

$$
\begin{aligned}
Recall &= \frac{TruePositive}{TruePositive + FalseNegative} \\
&= \frac{TruePositive}{TotalActualPositive}
\end{aligned}
$$

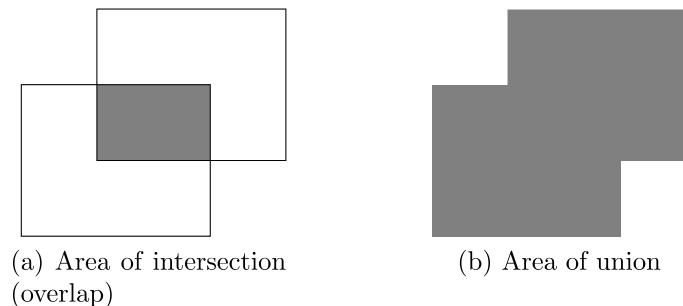
Usually, to evaluate the quality of object detection models, it is essential to consider both precision and recall. However, when precision improves, recall declines and vice-versa. For instance, a model with a higher threshold for detection generates fewer false positives, but it also retrieves fewer true positives. That means, precision increases with increasing threshold while recall decreases. On the contrary, if the model is too sensitive, it detects most of the relevant instances (true positives), but it also ends up producing many false positives. The higher model sensitivity makes precision lower, and recall higher. Therefore, it is important to find a balance between precision and recall depending on the application. In our application, it is important to have fewer false positives and consequently higher precision.

3.3.3 Intersection Over Union (IoU)

IoU is a useful measure which assists in evaluating the performance of object detectors. IoU signifies the similarity between the predicted region (bounding box) and ground truth. In other words, IoU indicates how good our prediction is as compared to the ground truth. IoU is calculated based on the formula given below:

$$IoU = \frac{AreaofIntersection(Overlap)}{AreaofUnion}$$

Given two bounding boxes, a predicted bounding box, and a ground truth bounding box, an area of overlap and an area of union are estimated as shown in Figure 3.16 below:



(a) Area of intersection (overlap)

(b) Area of union

Figure 3.16: Areas of intersection and union

3.3.4 Precision-Recall Curves

The Precision-Recall (PR) curves are drawn to highlight the relationship between precision and recall at different threshold values (different from IoU threshold values). As mentioned earlier, the PR curves are a good measure for binary classification. However, our work is concerned with object detection with multiple classes. To draw PR curves, we plot the precision and recall values for each class against the rest of the classes for different scores (threshold values) at 0.5 IoU.

For the task of object detection, the evaluation of true positives, false positives, and false negatives vary from application to application. The evaluation metrics in Tensorflow Object Detection API is based on PASCAL VOC 2010 detection metrics [12]. According to VOC 2010 metrics, a predicted box is considered as true positive only when the IoU of the predicted box is over 0.5 or 50% of the ground truth box. Additionally, each object is associated with only one bounding box. If there are multiple predictions for a single object, only one of them is considered as

a true positive, and all the other detections are false positives. Also, if any of the objects are not predicted, they are considered false negatives.

3.3.5 Mean Average Precision (mAP)

Tensorflow Object Detection API uses mAP as an evaluation protocol to measure and compare the accuracy of object detection models.[6] According to PASCAL metrics, mAP is calculated as the mean of average precision (AP) of all the object classes. Average precision is the area under the PR curve for each object class. The following section describes how to calculate AP for each class.

Calculating AP

In object detection, first the predicted bounding boxes (for all the images) are sorted according to their confidence score, and the precision and recall values are calculated for each confidence score. Afterwards, a PR curve is plotted for each of these precision-recall values at different scores. As pointed out above, AP summarizes the area under the PR curve in one number. AP is the averaged precision across all recall values between 0 and 1.

However, the PASCAL VOC metric is based on interpolated average precision. Interpolated average precision is defined as "the mean precision at a set of eleven equally spaced recall levels [12]." AP is calculated based on the formula given below:

$$AP = \frac{1}{11} \sum_{r \epsilon \{0, 0.1, 0.2.., 1\}} p_{interp}(r)$$

In interpolated AP, precision at each recall level corresponds to the maximum precision across all the recall values higher than recall level. Precision at recall is calculated as follows:

[6] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/evaluation_protocols.md

$$p_{interp}(r) = \max_{\tilde{r}:\tilde{r} \geq r} p(\tilde{r}),$$

$$\text{where } p(\tilde{r}) = \text{Precision measured at recall } \tilde{\text{r}},$$
$$\tilde{r} = \text{Recall values higher than recall level r}$$

3.4 Summary

This chapter discussed the implementation of object detection using Tensorflow. We first described the technique to collect the food datasets from the ImageNet database and pre-process them according to our requirements. Next, we described in detail the method to fine-tune the pre-trained models available in Tensorflow Object Detection API. For fine-tuning, we selected three object detection models including SSD-MobileNet-v2, FRCNN-ResNet-101, and RFCN-ResNet-101. We also discussed the fine-tuning of these three models with different configuration settings. Finally, we explained the evaluation metrics used in this thesis to evaluate our object detection models. The next chapter discusses the evaluation of object detection models based on the evaluation metrics described in this chapter.

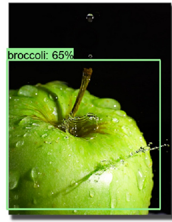
Chapter 4

Results and Discussion

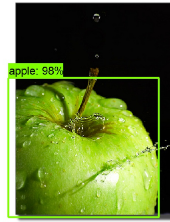
This chapter presents a comparison in performance of the pre-trained models and the fine-tuned models based on precision, recall, and mAP (cf. Section 3.3). The models are evaluated on a separate test dataset for an unbiased evaluation. Additionally, we are also comparing the performance of fine-tuned models with different configuration settings. Moreover, we are analyzing the performance of three fine-tuned models in terms of speed and accuracy and discuss the most relevant model (out of the three) for our application.

4.1 Pre-trained vs. Fine-tuned Models

As stated in earlier chapters, models pre-trained on the COCO dataset do not perform well for the eight food classes, *Apple, Bell pepper, Cauliflower, Lemon, Orange, Pear, Tomato and Turnip*. In Figure 4.1, the pre-trained model is not able to detect Apple correctly, although Apple is an object class in the COCO dataset. Besides, in Figure 4.2, the pre-trained model detects Tomato as Apple, and Vase (in pink color bounding box). On the contrary, the fine-tuned model detects Apple and Tomato correctly in Figures 4.1 and 4.2, respectively.

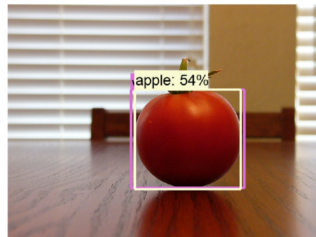


(a) Pre-trained model

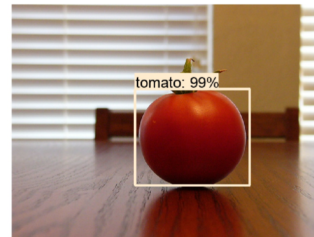


(b) Fine-tuned model

Figure 4.1: Detections for class Apple (present in pre-trained models)



(a) Pre-trained model



(b) Fine-tuned model

Figure 4.2: Detections for class Tomato (not present in pre-trained models)

In this section, we are comparing the performance of pre-trained models with their fine-tuned counterparts on the basis of precision, recall, and mAP (Mean Average Precision). The precision, recall, and mAP values are evaluated on the test dataset with 488 examples of eight classes. It should be noted that all the results shown below are calculated on the test dataset only.

4.1.1 SSD-MobileNet-v2: Pre-trained and Fine-tuned Model

The SSD model pre-trained on the COCO dataset is trained for two classes out of the eight classes of our interest, they are Apple and Orange. Therefore, we compared the performance of the pre-trained model with the fine-tuned model for these two classes for ten different values of IoU. The fine-tuned model used here is trained for 50,000 steps. Figures 4.3 and 4.4 illustrate the difference in performance of the two models for the two classes Apple and Orange, respectively.

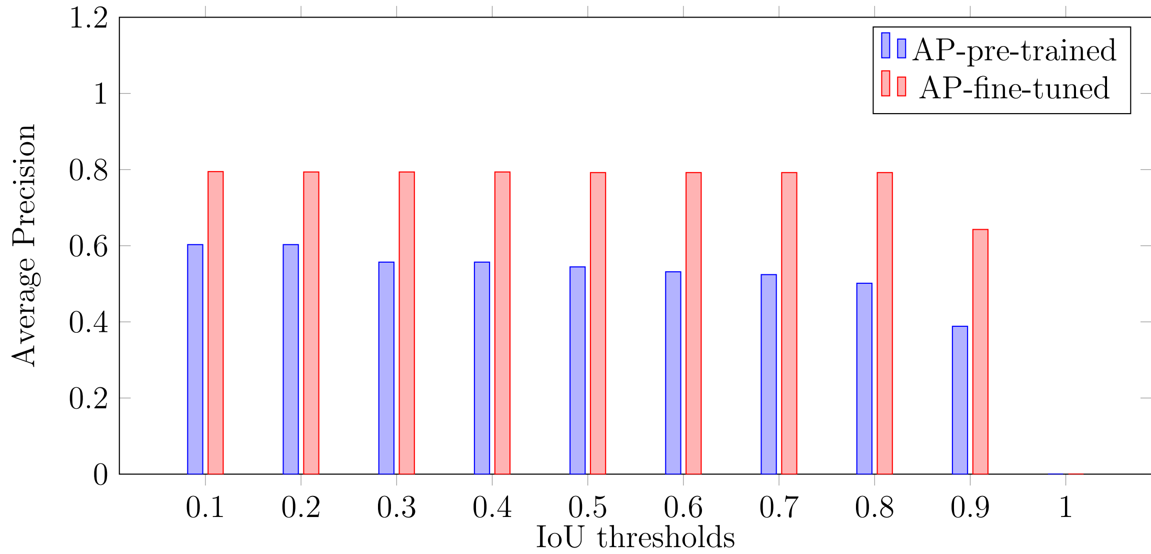


Figure 4.3: SSD: Performance of pre-trained model vs. fine-tuned model for class Apple

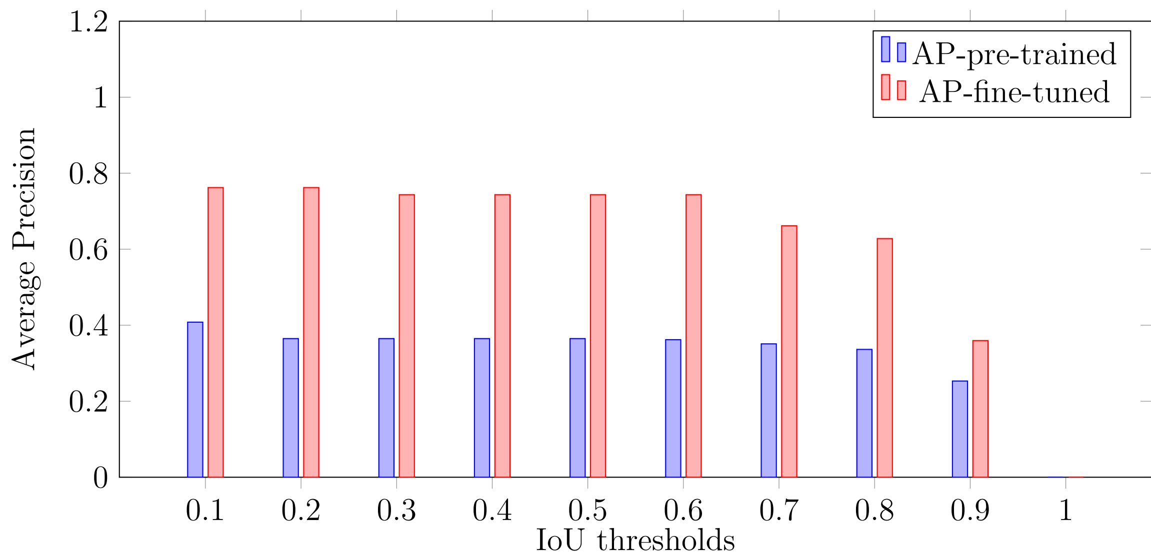


Figure 4.4: SSD: Performance of pre-trained model vs. fine-tuned model for class Orange

Figures 4.3 and 4.4 display the average precision (AP) of the pre-trained and fine-tuned models for various values of IoU threshold. However, as stated in Section 3.3.4, Tensorflow Object Detection API follows the PASCAL VOC 2010 metrics, where a predicted bounding box is considered as true positive (TP) only when IoU ≥ 0.5 with respect to ground truth bounding box. So, the other results presented in this section

are based on the PASCAL VOC 2010 metrics (i.e., they are calculated at 0.5 IoU).

As presented in Chapter 3, the fine-tuned model for SSD-MobileNet-v2 witnessed a mAP score of 85.5% at iteration 6,800. The mAP score further increased to 88% around step 19,000. After 19,000 steps of training, the mAP was fairly constant with minor variations. So, to evaluate the fine-tuned model on the test dataset, we exported the inference graph at approximately 20,000, 50,000 and 80,000 steps. The three steps are selected such that:

- 20,000 is selected based on the fact that the mAP reaches significantly closer to maximum mAP value of 88% at step 19,000.
- 50,000 is decided as the SSD-Mobilenet-v2 fine-tuned model performs best on the test data for the model exported at this step.
- 80,000 is picked to make sure that the model trained with 50,000 steps is good enough and does not improve further with training.

Tables 4.1-4.3 present the recall, precision, and average precision (AP) values for the pre-trained model and the fine-tuned model at 20,000, 50,000 and 80,000 steps of training for eight food classes. Table 4.3 also highlights the mAP value for the pre-trained model and the fine-tuned model at three different steps.

Table 4.1: SSD: Precision values for eight classes for pre-trained and fine-tuned models at different steps of training

	Precision			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	0.3643	0.8868	0.8793	0.8060
Bell pepper	-	0.6395	0.8772	0.7536
Cauliflower	-	0.9333	0.9333	0.9815
Lemon	-	0.7941	0.8889	0.7778
Orange	0.43	1.00	0.8772	1.00
Pear	-	0.98	0.9286	1.00
Tomato	-	0.8148	0.9423	0.9792
Turnip	-	1.00	1.00	1.00
Average	**0.3972**	**0.8811**	**0.9159**	**0.9123**

Table 4.2: SSD: Recall values for eight classes for pre-trained and fine-tuned models at different steps of training

	Recall			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	0.7833	0.7833	0.85	0.90
Bell pepper	-	0.8871	0.8065	0.8387
Cauliflower	-	0.9180	0.9180	0.8689
Lemon	-	0.45	0.6667	0.4667
Orange	0.6935	0.5968	0.8065	0.5645
Pear	-	0.8167	0.8667	0.7833
Tomato	-	0.7097	0.7903	0.7581
Turnip	-	0.70	0.7833	0.70
Average	**0.7384**	**0.7327**	**0.8110**	**0.7350**

Table 4.3: SSD: AP values for eight classes, and mAP values for pre-trained and fine-tuned models at different steps of training

	Average Precision (AP) in %			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	54.44	70.05	79.21	72.25
Bell pepper	-	75.01	81.11	76.94
Cauliflower	-	90.91	89.18	81.82
Lemon	-	38.80	62.47	39.81
Orange	36.49	54.55	74.35	54.55
Pear	-	81.82	79.67	72.73
Tomato	-	65.22	72.36	72.53
Turnip	-	63.64	72.73	63.64
mAP	**45.47**	**67.50**	**76.39**	**66.78**

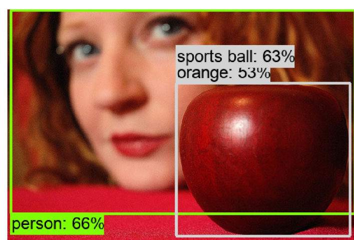
As pointed out earlier, the fine-tuned model trained until 50,000 steps performs best among all the models. The precision is 0.9159, recall is 0.8110, and mAP is 76.39% for SSD-MobileNet-v2 trained up-to 50,000 steps. As evident from Table 4.3, the mAP score decreases for model fine-tuned up-to 80,000 steps. This depreciation

is due to the fact that our trained model starts over fitting the training data. Besides, the mAP score of the fine-tuned model is over 30% more than that of the pre-trained model. Moreover, Figure 4.5 compares the detections of the pre-trained model and the fine-tuned model for a class not present in the COCO dataset. Similarly, Figures 4.6 and 4.7 highlight the differences in detections of the two models for a class present in the COCO dataset. **Note:** The COCO dataset is a dataset used for training of pre-trained models.

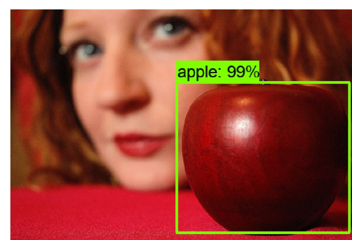
(a) Pre-trained model

(b) Fine-tuned model

Figure 4.5: Detections of SSD models for class Pear (not present in pre-trained model)

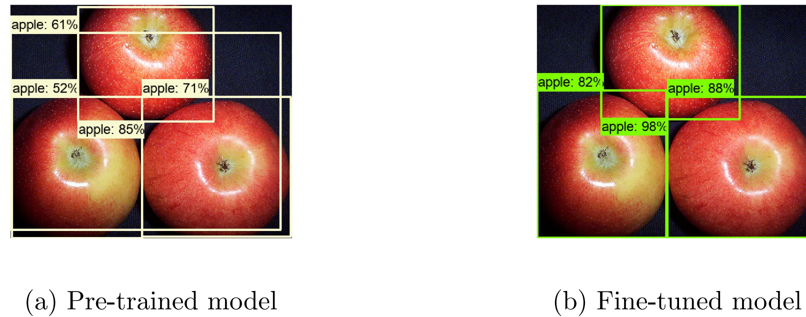


(a) Pre-trained model



(b) Fine-tuned model

Figure 4.6: Detections of SSD models for class Apple (present in pre-trained model)



(a) Pre-trained model (b) Fine-tuned model

Figure 4.7: Detections of SSD models for class Apple (present in pre-trained model)

As observed in Figure 4.6, the pre-trained model is not able to detect Apple while Apple is a class of the COCO dataset. Moreover, in Figure 4.7, the pre-trained model detects four apples instead of three and with a lesser confidence score as compared to its fine-tuned counterpart.

4.1.2 FRCNN-ResNet-101: Pre-trained and Fine-tuned Model

Like SSD model, we also compared the mAP values of the Faster R-CNN pre-trained model and the fine-tuned model for classes Apple and Orange. The fine-tuned model used here is also trained up-to 50,000 steps. Figures 4.8 and 4.9 compare the performance of the two models for the two classes.

The results displayed below are calculated for different IoU values. However, the other results presented in this section are based on the PASCAL VOC 2010 metrics (i.e., they are evaluated at 0.5 IoU). Additionally, to evaluate the fine-tuned model on the test dataset, the inference graph is exported at approximately 20,000, 50,000 and 80,000 steps similar to SSD model.

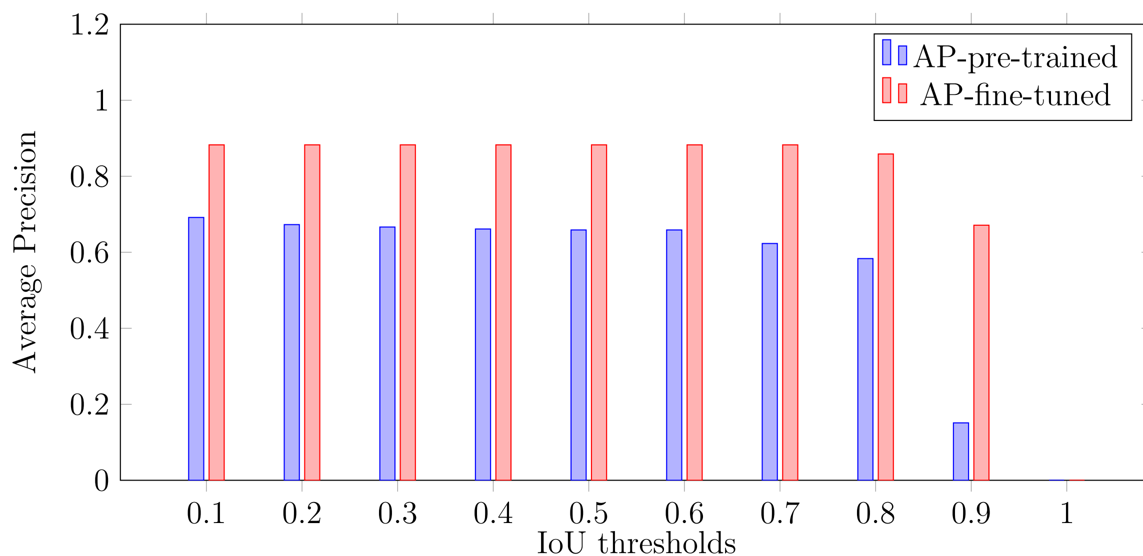


Figure 4.8: FRCNN: Performance of pre-trained model vs. fine-tuned model for class Apple

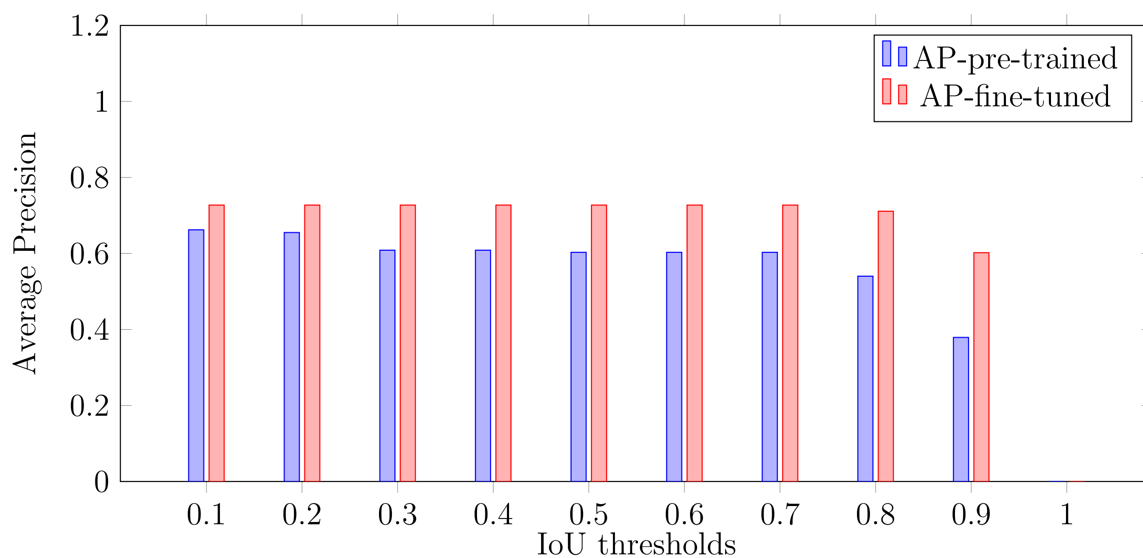


Figure 4.9: FRCNN: Performance of pre-trained model vs. fine-tuned model for class Orange

In the same way, Tables 4.4-4.6 display the recall, precision, and average precision (AP) values for the pre-trained model and fine-tuned model at different steps.

Table 4.4: FRCNN: Precision values for eight classes for pre-trained and fine-tuned models at different steps of training

	Precision			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	0.2764	0.7532	0.8529	0.7089
Bell pepper	-	0.6461	0.6941	0.7160
Cauliflower	-	0.8636	0.8906	0.8796
Lemon	-	0.7231	0.8214	0.7833
Orange	0.3972	0.96	0.9091	0.9091
Pear	-	0.8333	0.8689	0.7368
Tomato	-	0.8033	0.7639	0.7342
Turnip	-	0.9091	0.8235	0.8615
Average	**0.3368**	**0.8115**	**0.8281**	**0.7912**

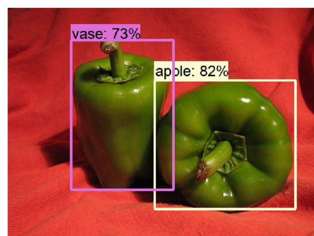
Table 4.5: FRCNN: Recall values for eight classes for pre-trained and fine-tuned models at different steps of training

	Recall			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	0.9167	0.9667	0.9667	0.9333
Bell pepper	-	0.8871	0.9516	0.9355
Cauliflower	-	0.9344	0.9344	0.9344
Lemon	-	0.7833	0.7667	0.7833
Orange	0.9032	0.7742	0.8065	0.8065
Pear	-	0.9167	0.8833	0.9333
Tomato	-	0.7903	0.8871	0.9355
Turnip	-	0.8333	0.9333	0.9333
Average	**0.9100**	**0.8608**	**0.8912**	**0.8994**

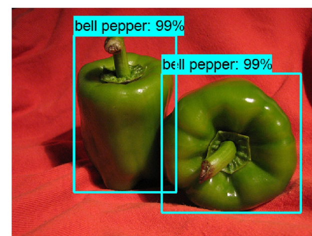
Table 4.6: FRCNN: AP values for eight classes, and mAP values for pre-trained and fine-tuned models at different steps of training

	Average Precision (AP) in %			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	65.87	85.81	88.26	81.71
Bell pepper	-	78.58	85.07	83.84
Cauliflower	-	87.06	87.03	86.21
Lemon	-	71.10	71.45	66.81
Orange	60.30	71.94	72.73	81.14
Pear	-	87.75	80.21	86.12
Tomato	-	71.15	78.75	80.37
Turnip	-	79.84	90.43	89.29
mAP	**63.09**	**79.15**	**81.74**	**81.94**

Table 4.6 shows that the mAP value is slightly more at 80,000 steps as compared to 50,000 steps. However, in Table 4.4 the average value of precision is considerably better for 50,000 steps. Therefore, the fine-tuned model at step 50,000 is considered as the best model for Faster R-CNN. Also, the mAP score of the fine-tuned model is over 18% more than that of the pre-trained model. Additionally, Figures 4.10 and 4.11 compare the detections of the pre-trained model with the fine-tuned model.



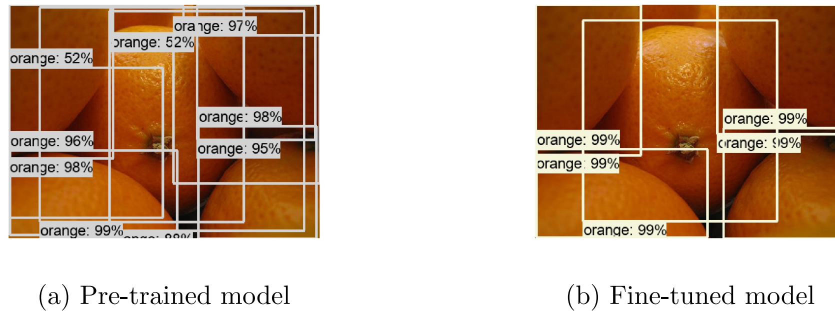
(a) Pre-trained model



(b) Fine-tuned model

Figure 4.10: Detections of Faster R-CNN models for class Bell Pepper (not present in pre-trained model)

In Figure 4.10, the pre-trained model detects Bell Peppers as Vase, and Apple as it



(a) Pre-trained model (b) Fine-tuned model

Figure 4.11: Detections of Faster R-CNN models for class Orange (present in pre-trained model)

is not present in the COCO dataset. Moreover, in Figure 4.11 which has five oranges, the pre-trained model detects more than five oranges and with a lower confidence score. On the contrary, the fine-tuned model detects five oranges, all with 99% confidence.

4.1.3 RFCN-ResNet-101: Pre-trained and Fine-tuned Model

Like SSD and Faster R-CNN model, we also compared the mAP values of the R-FCN pre-trained model with its fine-tuned counterpart for classes Apple and Orange. The fine-tuned model used here is trained for around 50,000 steps. Figures 4.12 and 4.13 compare the performance of the two models for the two classes.

The results presented below are calculated for different IoU values from 0.1 to 1.0. However, the other results showcased in this section are based on the PASCAL VOC 2010 metrics (i.e., they are evaluated at 0.5 IoU).

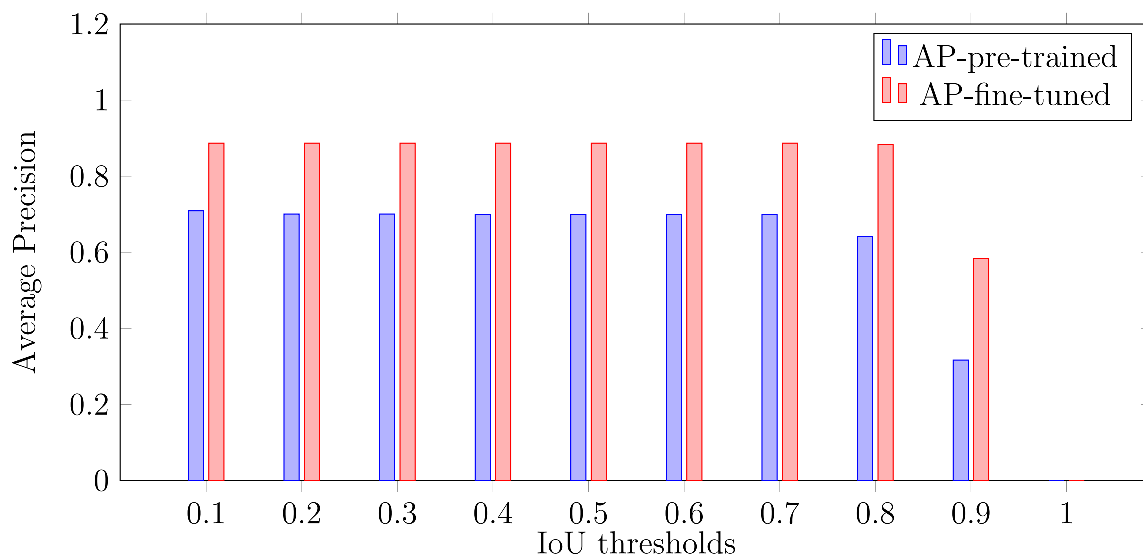


Figure 4.12: RFCN: Performance of pre-trained model vs. fine-tuned model for class Apple

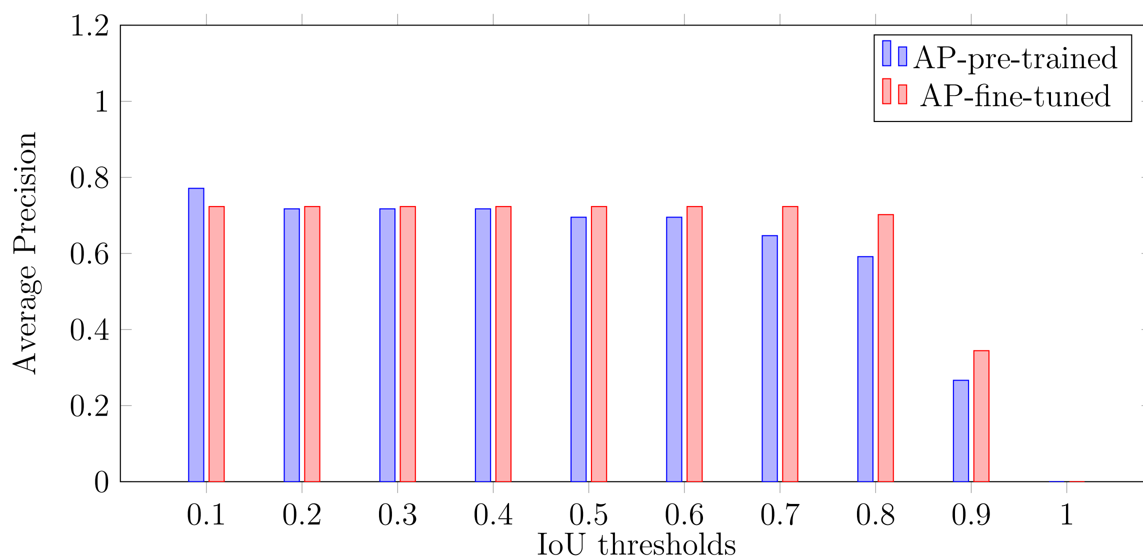


Figure 4.13: RFCN: Performance of pre-trained model vs. fine-tuned model for class Orange

Likewise, Tables 4.7-4.9 display the recall, precision, and average precision (AP) values for the pre-trained model and the fine-tuned model at different steps of training.

Table 4.7: RFCN: Precision values for eight classes for pre-trained and fine-tuned models at different steps of training

	Precision			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	0.3580	0.5413	0.6667	0.7160
Bell pepper	-	0.5895	0.5196	0.6374
Cauliflower	-	0.8594	0.6667	0.8769
Lemon	-	0.6912	0.7015	0.7619
Orange	0.4583	0.8596	0.7833	0.6757
Pear	-	0.6548	0.7051	0.5914
Tomato	-	0.8246	0.7714	0.7368
Turnip	-	0.9057	0.7826	0.8182
Average	**0.4082**	**0.7408**	**0.7043**	**0.7268**

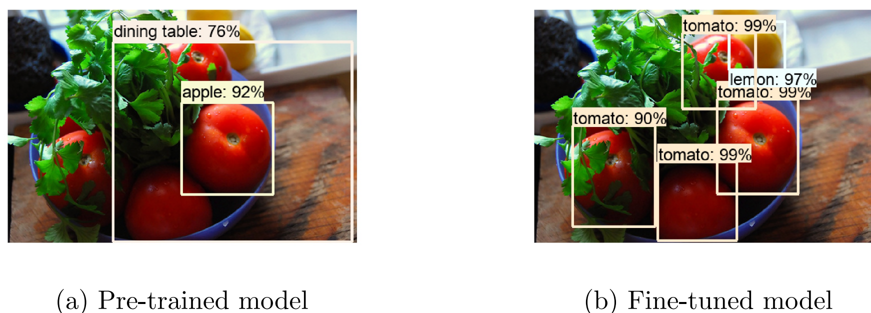
Table 4.8: RFCN: Recall values for eight classes for pre-trained and fine-tuned models at different steps of training

	Recall			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	0.9667	0.9833	0.9667	0.9667
Bell pepper	-	0.9032	0.8548	0.9355
Cauliflower	-	0.9016	0.9180	0.9344
Lemon	-	0.7833	0.7833	0.80
Orange	0.8871	0.7903	0.7581	0.8065
Pear	-	0.9167	0.9167	0.9167
Tomato	-	0.7581	0.8710	0.9032
Turnip	-	0.80	0.90	0.90
Average	**0.9269**	**0.8546**	**0.8711**	**0.8954**

Table 4.9: RFCN: AP values for eight classes, and mAP values for pre-trained and fine-tuned models at different steps of training

	Average Precision (AP) in %			
Classes	**Pre-trained**	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	69.89	80.26	88.68	88.04
Bell pepper	-	84.11	72.68	75.99
Cauliflower	-	90.24	85.69	87.72
Lemon	-	70.66	68.71	69.39
Orange	69.52	72.33	72.33	69.17
Pear	-	83.71	85.90	79.50
Tomato	-	69.50	78.25	78.98
Turnip	-	72.73	90.43	89.26
mAP	**69.71**	**77.94**	**80.33**	**79.76**

Table 4.9 shows that the mAP value at 50,000 steps is the highest among all the models. Moreover, the mAP score of the fine-tuned model is about 11% more than that of the pre-trained model. Moreover, Figures 4.14 and 4.15 compare the detections of the pre-trained model with the fine-tuned model.



(a) Pre-trained model

(b) Fine-tuned model

Figure 4.14: Detections of R-FCN models for classes Tomato and Lemon (not present in pre-trained model)

In Figure 4.14 there are four tomatoes and one lemon, the pre-trained model detects only one tomato as an apple. On the contrary, the fine-tuned model detects four tomatoes and a lemon correctly. Besides, the pre-trained model detects all three

(a) Pre-trained model (b) Fine-tuned model

Figure 4.15: Detections of R-FCN models for class Apple (present in pre-trained model)

apples in Figure 4.15 but with lesser confidence scores.

4.2 Fine-tuned Models with Different Configuration Settings

This section provides insights into the difference in performance of fine-tuned models with different configuration settings on the test dataset. The fine-tuned models are compared and analyzed on the basis of mAP values at various steps of training.

4.2.1 SSD-MobileNet-v2 Fine-tuned Models

As described in Section 3.2.1, we used four different configuration settings for the fine-tuning of the SSD model. The four different settings are as follows:

1. The provided configuration file settings
2. Increase initial learning rate to 0.005
3. Decrease initial learning rate to 0.003
4. Use three aspect ratios instead of five as in the default configuration file

Tables 4.10-4.13 present the average precision (AP) values of eight classes at different steps of training for four different configuration settings mentioned above.

Table 4.10: SSD: AP values for eight classes, and mAP values for fine-tuned model with the default configuration at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	70.05	79.21	72.25
Bell pepper	75.01	81.11	76.94
Cauliflower	90.91	89.18	81.82
Lemon	38.80	62.47	39.81
Orange	54.55	74.35	54.55
Pear	81.82	79.67	72.73
Tomato	65.22	72.36	72.53
Turnip	63.64	72.73	63.64
mAP	**67.50**	**76.39**	**66.78**

Table 4.11: SSD: AP values for eight classes, and mAP values for fine-tuned model with learning rate: 0.005 at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	68.50	60.53	68.45
Bell pepper	81.17	72.53	78.58
Cauliflower	81.07	81.82	81.64
Lemon	60.85	42.09	50.12
Orange	71.80	63.64	61.33
Pear	81.80	90.43	71.07
Tomato	68.82	66.75	63.22
Turnip	61.89	66.18	70.77
mAP	**71.90**	**68.00**	**68.15**

Table 4.12: SSD: AP values for eight classes, and mAP values for fine-tuned model with learning rate: 0.003 at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	76.91	66.23	74.97
Bell pepper	81.64	72.17	75.19
Cauliflower	89.00	89.76	79.81
Lemon	47.15	46.31	48.77
Orange	61.78	67.57	70.28
Pear	80.15	81.45	81.82
Tomato	72.15	66.77	65.48
Turnip	68.63	66.23	72.52
mAP	**72.18**	**69.56**	**71.11**

Table 4.13: SSD: AP values for eight classes, and mAP values for fine-tuned model with three aspect ratios at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	66.40	72.73	81.26
Bell pepper	80.79	79.70	79.13
Cauliflower	81.64	81.82	81.26
Lemon	58.24	51.71	47.73
Orange	72.13	72.15	63.64
Pear	81.82	77.18	80.21
Tomato	72.21	67.48	68.42
Turnip	80.87	63.64	69.52
mAP	**74.26**	**70.80**	**71.40**

As observed from the four tables, the SSD model trained with the default configuration file performs the best with a mAP score of 76.39%.

4.2.2 FRCNN-ResNet-101 Fine-tuned Models

Similarly for Faster R-CNN, as discussed in Section 3.2.2, we have used three different configuration settings for model fine-tuning. The three settings are as follows:

1. The provided configuration file settings

2. Use five aspect ratios instead of three as in the default configuration file

3. Reduce number of proposals to 100

Tables 4.14, 4.15, and 4.16 display the results for the provided configuration file, file with the updated aspect ratios, and the file with reduced number of proposals, respectively.

Table 4.14: FRCNN: AP values for eight classes, and mAP values for fine-tuned model with the default configuration at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	85.81	88.26	81.71
Bell pepper	78.58	85.07	83.84
Cauliflower	87.06	87.03	86.21
Lemon	71.10	71.45	66.81
Orange	71.94	72.73	81.14
Pear	87.75	80.21	86.12
Tomato	71.15	78.75	80.37
Turnip	79.84	90.43	89.29
mAP	**79.15**	**81.74**	**81.94**

Table 4.15: FRCNN: AP values for eight classes, and mAP values for fine-tuned model with five aspect ratios at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	79.21	79.21	86.84
Bell pepper	75.27	75.27	78.84
Cauliflower	81.98	81.98	83.46
Lemon	67.38	67.38	76.42
Orange	72.33	72.33	68.09
Pear	85.19	85.19	87.11
Tomato	70.91	70.91	75.73
Turnip	90.58	90.58	90.91
mAP	**77.86**	**77.86**	**80.93**

Table 4.16: FRCNN: AP values for eight classes, and mAP values for fine-tuned model with 100 proposals at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	80.92	84.65	86.23
Bell pepper	71.60	77.15	77.27
Cauliflower	86.98	84.75	86.36
Lemon	67.70	69.58	76.87
Orange	77.68	69.57	69.57
Pear	83.92	87.16	86.91
Tomato	80.16	80.72	81.65
Turnip	90.44	89.07	89.07
mAP	**79.93**	**80.33**	**81.74**

As evident from the three tables, the fine-tuned model trained with the default configuration outperforms the other two configuration settings.

4.2.3 RFCN-ResNet-101 Fine-tuned Models

Similar to SSD and Faster R-CNN model, we used three different configuration settings for fine-tuning of R-FCN model. The three configuration settings are as follows:

1. The provided configuration file settings
2. Use five aspect ratios instead of three as in the default configuration file
3. Reduce number of proposals to 100

Tables 4.17-4.19 present the results for the three configuration settings at different steps of training.

Table 4.17: RFCN: AP values for eight classes, and mAP values for fine-tuned model with the default configuration at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	80.26	88.68	88.04
Bell pepper	84.11	72.68	75.99
Cauliflower	90.24	85.69	87.72
Lemon	70.66	68.71	69.39
Orange	72.33	72.33	69.17
Pear	83.71	85.90	79.50
Tomato	69.50	78.25	78.98
Turnip	72.73	90.43	89.26
mAP	**77.94**	**80.33**	**79.76**

Table 4.18: RFCN: AP values for eight classes, and mAP values for fine-tuned model with five aspect ratios at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	78.54	81.53	81.40
Bell pepper	73.58	76.23	75.36
Cauliflower	84.81	83.91	81.97
Lemon	75.10	67.47	74.66
Orange	70.46	74.33	72.68
Pear	75.06	74.61	76.74
Tomato	70.50	58.53	64.34
Turnip	81.45	89.44	85.45
mAP	**76.19**	**75.76**	**76.58**

Table 4.19: RFCN: AP values for eight classes, and mAP values for fine-tuned model with 100 proposals at different steps of training

	Average Precision (AP) in %		
Classes	**Fine-tuned (20k)**	**Fine-tuned (50k)**	**Fine-tuned (80k)**
Apple	89.04	88.99	81.14
Bell pepper	75.62	82.85	72.17
Cauliflower	88.67	86.87	87.72
Lemon	71.27	70.00	74.92
Orange	70.77	71.10	72.33
Pear	85.98	83.81	85.03
Tomato	72.34	80.09	77.73
Turnip	81.82	88.80	90.91
mAP	**79.44**	**81.56**	**80.24**

As evident from the three tables, the fine-tuned model trained with the default configuration achieves the highest mAP value among all the configuration settings.

4.3 Speed-accuracy Trade-off of Different Object Detection Models

This section demonstrates the trade-off in speed and accuracy of the three fine-tuned models. As described in Section 2.4.1, it is an important and interesting goal of an object detector to achieve real-time speed while maintaining accuracy. Therefore, in this section, we compare the mAP values and the GPU speeds of the fine-tuned models of SSD-MobileNet-v2, FRCNN-ResNet-101, and RFCN-ResNet-101. Figure 4.16 displays the mAP value and speed of the fine-tuned models with different configuration settings.

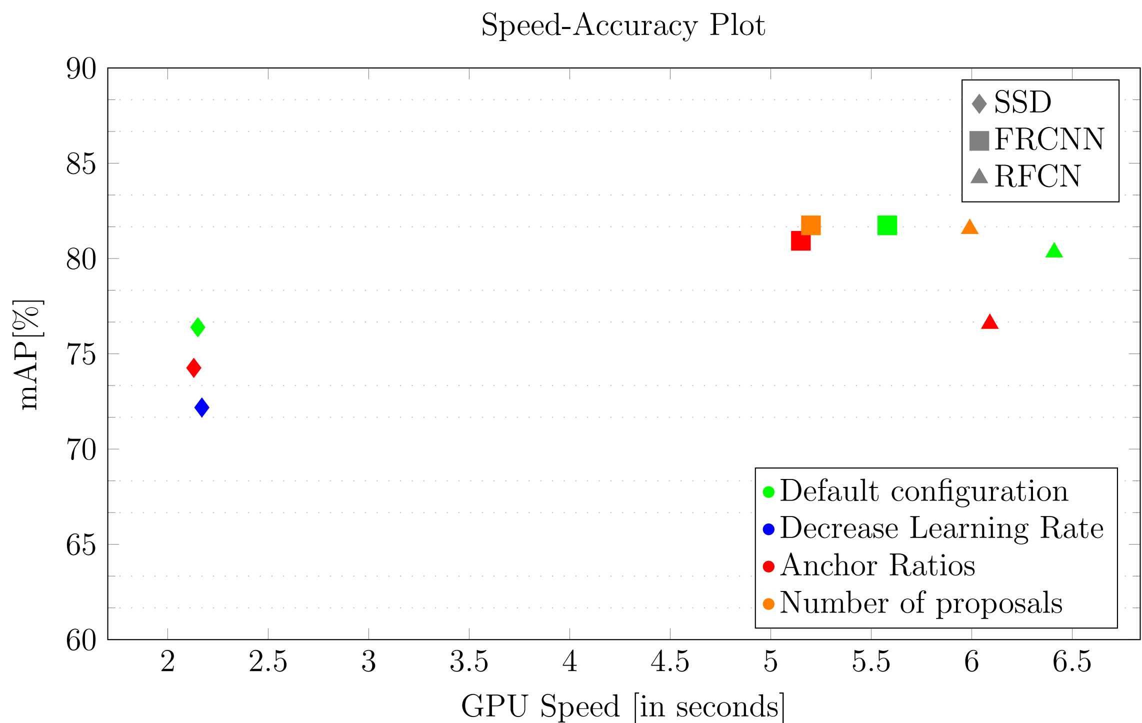


Figure 4.16: Speed-accuracy comparison of the three models

Figure 4.16 compares the accuracy and time of inference of the fine-tuned models of SSD-MobileNet-v2, FRCNN-ResNet-101, and RFCN-ResNet-101. As stated in Section 2.4.1, the time of inference of a SSD model is the least among all the models, i.e., SSD model is the fastest and takes around two seconds for inference on one Tesla P100-PCIe-12GB GPU. Besides, the performance of Faster R-CNN is the highest with a mAP score of 81.74%. Moreover, for Faster R-CNN, we observe the reduction

in time of inference on reducing the number of proposals.

4.4 Discussion

This section discusses the performance results of the object detections models presented in the previous sections. Table 4.20 summaries the results of the pre-trained and fine-tuned models from Section 4.1. Similarly, Table 4.21 presents the summary of the results of the fine-tuned models with different configurations from Section 4.2.

Table 4.20: Results of pre-trained models vs. fine-tuned models

	mAP in %	
Models	**Pre-trained model**	**Fine-tuned model**
SSD-MobileNet-v2	45.47	76.39
FRCNN-ResNet-101	63.09	81.74
RFCN-ResNet-101	69.71	80.33

Table 4.21: Results of fine-tuned models with different configuration settings

	mAP in %			
Models	**Default configuration**	**Decrease learning rate**	**Updated aspect ratios**	**Updated number of proposals**
SSD-MobileNet-v2	76.39	72.18	74.26	-
FRCNN-ResNet-101	81.74	-	80.93	81.74
RFCN-ResNet-101	80.33	-	76.58	81.56

As gleaned from Table 4.20, the fine-tuned models outperform the pre-trained models in all the three cases. For SSD, the mAP value of the fine-tuned model is over 30% better than that of the pre-trained model. Likewise, for Faster R-CNN and R-FCN, it is 18% and 10% more than their respective pre-trained models. Moreover,

we can conclude from Table 4.21 that the default configuration in all three cases generates higher (or similar as in case of Faster R-CNN) mAP values. Therefore, to compare speed and accuracy, we focus on the three models fine-tuned with their default configurations. Table 4.22 displays the speed and accuracy results from Section 4.3.

Table 4.22: Results of the three fine-tuned models

	Speed-accuracy trade-off	
Models	**GPU time in seconds**	**mAP in %**
SSD-MobileNet-v2	2.15	76.39
FRCNN-ResNet-101	5.58	81.74
RFCN-ResNet-101	6.41	80.33

As pointed out earlier, the time of inference for the SSD model is the least, i.e., the SSD model is the fastest among all the models. Moreover as expected, the mAP value is the highest for Faster R-CNN which makes it the most accurate of all. Additionally, Faster R-CNN is approximately 5% more accurate than the SSD model. However, in terms of speed, the SSD model is around 61% faster than the Faster R-CNN model.

To summarize, the speed of an object detector is an essential aspect for real-time applications. Therefore, to achieve our objective of detecting objects in a refrigerator, we select SSD-MobileNet-v2 as an excellent model for our application.

4.5 Summary

In this chapter, we discussed the evaluation of object detection models on the basis of precision, recall, and mAP. We discussed the difference in performance of the pre-trained model and fine-tuned model on the test dataset. We also presented the comparison of the pre-trained and fine-tuned model for the two classes present in both of their training datasets. We also discussed the comparison in performance of the fine-tuned models with different configuration settings. We compared and analyzed the three fine-tuned models on the basis of speed and accuracy. We concluded the chapter with a discussion on the most suitable model for our application.

Chapter 5

Conclusions

This chapter summarizes this thesis, outlines the contributions and concludes with the insights into future work to improve object detection in refrigerators.

5.1 Summary

Object detection is a technology related to computer vision for detecting multiple objects in an image. Convolutional Neural Networks (CNN), deployed as image classifiers, are widely used for the realization of object detection models. The main objective of our thesis is to deploy an object detection model to recognize and localize objects in a refrigerator. While a smart refrigerator also utilizes object detectors to detect objects inside them, they often work on a closed platform. Therefore, this thesis aims to test different object detection models as a) a pre-trained model and b) a fine-tuned model to infer a model suitable technique for our objective of detecting objects in a refrigerator.

In this thesis, we explained the function of CNN and how they are being used for the task of image classification. We discussed the evolution of object detection models from the relatively slow R-CNN to the latest optimized models. This evolution is not due to CNN itself; instead, it is because of how we use CNN, where we place CNN and the computations that take place around CNN. R-CNN is slow since it entails several independent phases like RoI (region of interest) generation, convolution process of each region, computing fully connected layers, and finally classification and regression step. However, in the current models, these phases are progressively being combined with CNN itself. In recent years, the speed of object detection models has become

more critical and increased more drastically than accuracy (mAP).

We discussed the open source frameworks such as Google's Tensorflow Object Detection API and Microsoft Cognitive Toolkit (CNTK) available for object detection. We have used Tensorflow Object Detection API for its easy-to-use approach and well-documented code base. Tensorflow Object Detection API has several object detection models pre-trained on publicly available datasets. However, they are not suitable for detecting the objects relevant in our application. Therefore, we fine-tuned the pre-trained models, trained initially on the COCO dataset, for our dataset. To create our dataset, we extracted eight food class datasets from the ImageNet database and pre-processed them according to our requirements. After pre-processing, we divided each of these eight datasets into training, validation, and test datasets. We then combined the corresponding datasets from each class into one training dataset, one validation, and one test dataset. We used the training and validation datasets for the fine-tuning of pre-trained models, and test dataset for unbiased evaluation.

We selected the three pre-trained models namely, SSD-MobileNet-v2, FRCNN-ResNet-101, and RFCN-ResNet-101 for the process of fine-tuning. Each of the pre-trained models is fine-tuned with the provided (default) configuration setting and other different configuration settings. While training, we exported the inference graph of the fine-tuned models at different steps of training for evaluation on the test dataset.

We evaluated and compared the pre-trained models and the fine-tuned models on the basis of precision, recall, and mAP. The fine-tuned models are evaluated at different steps of training such as 20,000, 50,000 and 80,000 steps. We also compared fine-tuned models with different configuration settings based on mAP. Lastly, we compared and analyzed the three object detection models on the basis of speed and accuracy to select a viable model to detect objects in a refrigerator.

5.2 Contributions

This section summarizes the main contributions of this thesis:

- **Method to collect and pre-process the food datasets.** One of the main contributions of this thesis is to gather and pre-process the datasets to create training, validation, and test datasets. The training and validation datasets are required to train an object detection model. Moreover, test dataset is required for an unbiased evaluation of these models. We collected eight food datasets

from the ImageNet database and pre-processed manually using an annotation tool known as LabelImg. The method to collect and pre-process the datasets is described in Chapter 3.

- **Survey of pre-trained object detection models.** We conducted a literature review of the state-of-the-art image classification and object detection architectures. We selected three pre-trained object detection models based on that review. The selected pre-trained models are SSD-MobileNet-v2, FRCNN-ResNet-101, and RFCN-ResNet-101. The state-of-the-art image classification and object detection techniques are described in Chapter 2.

- **Implementation of fine-tuned models using transfer learning.** To improve upon the performance further, we have fine-tuned the three selected pre-trained models based on the training and validation datasets created earlier (Contribution 1). We have fine-tuned the models using Tensorflow Object Detection API. The implementation steps are described in Chapter 3.

- **Selection of a suitable model for detecting food classes.** We evaluated the pre-trained models with their fine-tuned counterparts to draw a comparison between them on the basis of precision, recall, and mAP. The three different fine-tuned models are also compared and analyzed based on speed and accuracy. The fine-tuned model of SSD-MobileNet-v2 is selected as an optimal model for our application. These different evaluations are described in Chapter 4.

5.3 Future Work

This section discusses selected ideas for future work that can further enhance the accuracy and usefulness of object detection:

- Currently, our application focuses on only eight food classes namely *Apple, Bell pepper, Cauliflower, Lemon, Orange, Pear, Tomato and Turnip*. However, there are still many other objects in our refrigerators that are not included in our application. Therefore, we can further extend the number of classes by augmenting our training dataset with examples of other food classes and fine-tuning a model for all of them.

- In our current implementation, object detection is based on an inference graph exported from fine-tuning a model. However, to improve the performance further, we can explore ways to implement context-sensitive object detector. To illustrate, we can leverage the contextual information of a whole image to make the detections more precise. For instance, suppose an image depicts the scene of a refrigerator, and our object detector detects potato in that image. Based on the image context, i.e., it is an image of a refrigerator, we can infer that detected object could not be potato since potato is not usually kept inside a refrigerator.

- As pointed out earlier in Chapter 1, we can further improve the accuracy of our detector by incorporating the grocery context and personal context of a user. For example, we can acquire the grocery context of a user from the grocery shopping receipt. Also, suppose our detector detects an object with two overlapping bounding boxes for orange and lemon. Based on the grocery context and assuming the user does not have lemons and oranges in her refrigerator initially, we might argue that the object is orange and not lemon. Another way would be to utilize the personal context of a user like allergies of a user. Suppose, a user is allergic to eggs, and if our object detector detects eggs, we might argue that user could not have eggs in her refrigerator since she is allergic to them.

- Our object detector can be integrated with a recipe recommender system, like CAPRecipes [26], to supplement refrigerator context. CAPRecipes exploits users' personal context, temporal context and kitchen context to recommend personalized recipes. The kitchen context comprises of attributes of available ingredients in terms of their quantity and expiry dates. The refrigerator is an integral part of the kitchen and in the same way, refrigerator context is an important component of kitchen context. Therefore, we can utilize the refrigerator context (ingredients available in the refrigerator) to recommend recipes to users.

- In the same way, we can associate our object detector with Foodie [3], a text and voice-enabled conversational kitchen assistant. Foodie is customized to understand and assist with food-related topics. One of the major goals of Foodie is to recommend recipes to users based on their personal, physical, and linguistic context. Similar to CAPRecipes, we can also provide refrigerator

context to Foodie for recommending recipes.

- We can also integrate our object detector with SmartGrocer [27], a context-aware personalized grocery system to provide personalized coupons to users. SmartGrocer creates a grocery list based on the recipe recommendations from CAPRecipes. However, SmartGrocer could not track ingredients used by other recipes not recommended by CAPRecipes. Using our detector, we can also create a grocery list by comparing each image with the previous one to get an idea of the missing ingredients and adding them to the grocery list. We can then provide this grocery list to SmartGrocer to account for ingredients used by other recipes.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016. 31

[2] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Mahmudul Hasan, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches. *arXiv preprint arXiv:1803.01164*, 2018. 15

[3] Prashanti Priya Angara. Towards a Deeper Understanding of Current Conversational Frameworks through the Design and Development of a Cognitive Agent. Master's thesis, Department of Computer Science, University of Victoria, 2018. 82

[4] Ben Athiwaratkun and Keegan Kang. Feature representation in convolutional neural networks. *arXiv preprint arXiv:1507.02313*, 2015. 12

[5] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European Conference on Computer Vision*, pages 404–417. Springer, 2006. 15

[6] Yoshua Bengio et al. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009. 10

[7] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in Neural Information Processing Systems*, pages 379–387, 2016. xi, 1, 27, 28

[8] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 886–893. IEEE, 2005. 15

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009. 2, 16

[10] Cicero dos Santos and Maira Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings 25th International Conference on Computational Linguistics, COLING 2014: Technical Papers*, pages 69–78, 2014. 15

[11] George H Dunteman. *Principal Components Analysis*. Number 69. Sage, 1989. 15

[12] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The Pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010. 24, 52, 53

[13] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167–181, 2004. 23

[14] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)*, 2013. 2

[15] Ross Girshick. Fast r-cnn. In *Proceedings IEEE International Conference on Computer Vision*, pages 1440–1448. IEEE, 2015. x, 24, 25

[16] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587. IEEE, 2014. x, 22, 23, 24

[17] Peter Goldsborough. A tour of Tensorflow. *arXiv preprint arXiv:1610.01178*, 2016. 31

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 1. MIT press Cambridge, 2016. 10

[19] Chunhui Gu, Chen Sun, David A Ross, Carl Vondrick, Caroline Pantofaru, Yeqing Li, Sudheendra Vijayanarasimhan, George Toderici, Susanna Ricco, Rahul Sukthankar, et al. AVA: A video dataset of spatio-temporally localized atomic visual actions. *CoRR, abs/1705.08421*, 4, 2017. 2

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. IEEE, 2016. x, 1, 2, 10, 13, 16, 17, 18, 35

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. 18

[22] Robert D. Hof. Deep Learning: With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart. *MIT Technology Review*, 2018. 10

[23] Shin Hoo-Chang, Holger R Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M Summers. Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning. *IEEE Transactions on Medical Imaging*, 35(5):1285, 2016. 3

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 18, 19

[25] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, volume 4. IEEE, 2017. xi, 32

[26] Harshit Jain. *CAPRECIPES: A context-aware personalized recipes recommender for healthy and smart living.* PhD thesis, Department of Computer Science, University of Victoria, 2018. 82

[27] Roshni Jain. *SmartGrocer: A context-aware personalized grocery system.* PhD thesis, Department of Computer Science, University of Victoria, 2018. 83

[28] Fares Jalled and Ilia Voronkov. Object Detection Using Image Processing. *arXiv preprint arXiv:1611.07791*, 2016. 8

[29] Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. Flattened convolutional neural networks for feedforward acceleration. *arXiv preprint arXiv:1412.5474*, 2014. 18

[30] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732. IEEE, 2014. 3

[31] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017. 2

[32] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 30

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. 1, 2, 8, 10, 11, 13, 16, 35

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436, 2015. 10

[35] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomForest. *R News*, 2(3):18–22, 2002. 15

[36] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014. 2

[37] Tony Lindeberg. *Scale invariant feature transform*, volume 7. 2012. QC 20120524. 15

[38] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016. xi, 2, 28, 29, 30

[39] Tien-Dung Mai, Thanh Duc Ngo, Duy-Dinh Le, Duc Anh Duong, Kiem Hoang, and Shinichi Satoh. Efficient large-scale multi-class image classification by learning balanced trees. *Computer Vision and Image Understanding*, 156:151–161, 2017. 15

[40] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Book Company, 1997. 8

[41] Andrew Ng. What data scientists should know about deep learning. *Extract Data Conference*, 2015. x, 10

[42] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. 9, 13

[43] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 1717–1724. IEEE, 2014. 3

[44] Ladislav Rampasek and Anna Goldenberg. Tensorflow: Biology's gateway to deep learning? *Cell Systems*, 2(1):12–14, 2016. 31

[45] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016. 18

[46] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015. x, 1, 25, 26, 27

[47] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 16

[48] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. IEEE, 2018. 20, 21

[49] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016. 30

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1, 2, 10, 13, 16, 35

[51] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 15

[52] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in Neural Information Processing Systems*, pages 2377–2385, 2015. 18

[53] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015. 18

[54] Ingo Steinwart and Andreas Christmann. *Support Vector Machines*. Springer Science & Business Media, 2008. 15

[55] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings IEEE Conference on*

Computer Vision and Pattern Recognition, pages 1–9. IEEE, 2015. 1, 2, 10, 13, 16, 35

[56] Srinivas TK and Ramakrishnan Viswanathan. Cognitive Computing: The Next Stage in Human/Machine Coevolution. Technical report, Cognizant White Paper, 2017. 10

[57] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013. 23

[58] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012. 15