MobileNet SSD

NUS National University of Singapore | ISS INSTITUTE OF SYSTEMS SCIENCE

Contents

1. Executive Summary

This report explores the methodology behind the popular lightweight object detector MobileNet SSD. MobileNet SSD is actually a combination of 2 separate algorithms published in separate papers. The MobileNet is a lightweight feature extractor meant for deployment in Mobile phones and has gone through a couple of iterations. The SSD on the other hand is an innovative object detector model that utilizes multiple feature maps to perform object detection. The original SSD was in fact trained on VGG and not MobileNet.

In this report we explore the use of MobileNet v2 combined with SSD to do object detections of everyday supermarket items. Manpower shortage together with an ageing working population are problems that many companies around the world are facing today. The problem of finding enough manpower to operate the increasing number of 24-hours supermarkets and increasing probability of payment errors at checkout due to the ageing population of cashiers are definitely problems that the supermarkets face today.

Our team hopes to solve both problems through the use of computer vision. An automated checkout Point of Sale system that utilizes a lightweight object detection algorithm. As such our team has gathered a collection of simple everyday items such as bananas and apples to train our object detector classifier.

The team faced numerous challenges and went through multiple iterations in not just the model creation process but also the data collection and pre-processing. This will all be further elaborated in the respective sections.

The team managed to obtain a validation loss of 0.43871.

2. Understanding MobileNet SSD

2.1. MobileNet

The MobileNet architecture is very similar to many of the feature extractor models, using a series of convolutional layers for the extraction (Howard, Zhu, & Chen, 2017). The unique feature of MobileNet is that instead of using a typical convolutional 2D layer, a combination of depth wise and point wise convolutions are used instead as shown in Fig 2.1a and 2.1b. This helps to greatly reduced the number of parameters required for the model. The entire MobileNet network simply consist of multiple blocks of these depth wise and point wise convolutions.

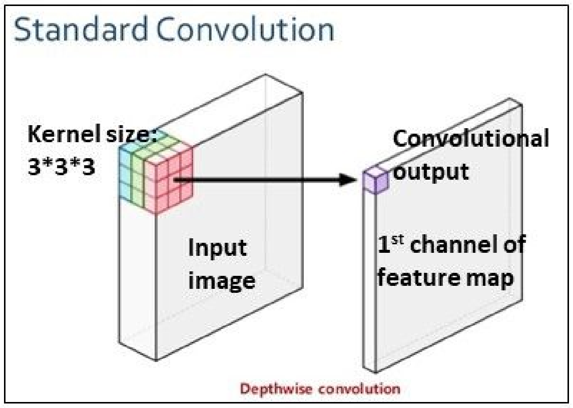


Fig 2.1a: Typical Convolution

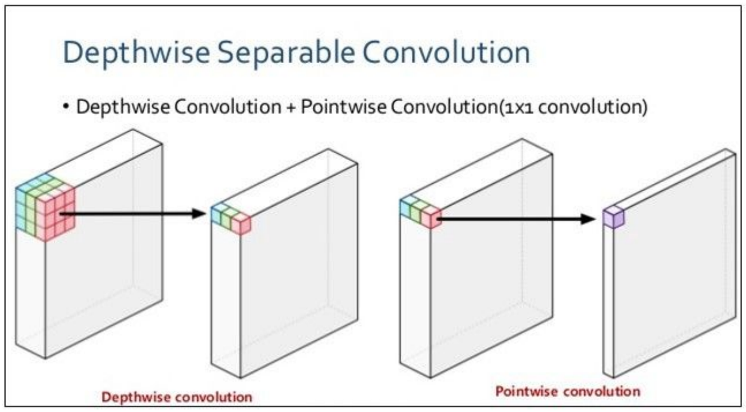
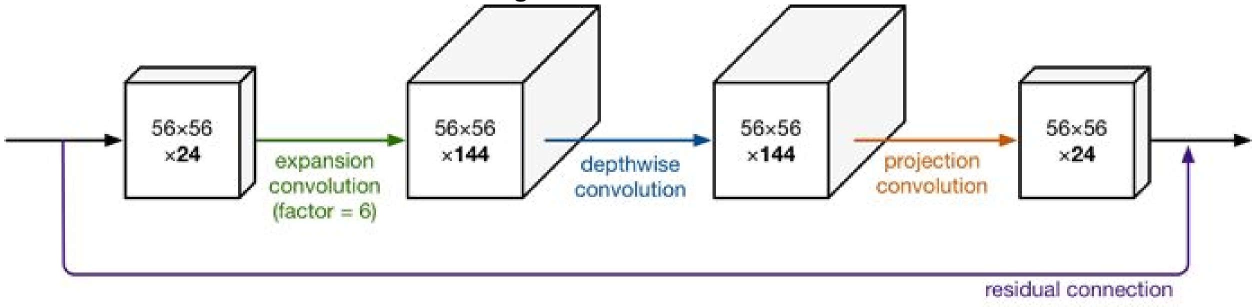

Depthwise Separable Convolution
• Depthwise Convolution + Pointwise Convolution(1x1 convolution)
Depthwise convolution
Pointwise convolution

Fig 2.1b: MobileNet Convolutions

The newer version 2 of MobileNet that we are using has 2 additional new features compared to version 1. The first being the additional residual connection that is added to the end of each block. The second being the bottleneck feature which the research papers claims to enhance the feature extraction capabilities of MobileNet for each block. In our project, we have utilized the pretrained version of MobileNet v2 from Keras with an input shape of 224 x 224 x 3.

Fig 2.1c: MobileNet v2



2.2. SSD

The original SSD architecture utilizes a total of 6 feature maps (38 x 38, 19 x 19, 10 x 10, 5 x 5, 3 x 3, 1 x 1) extracted from the VGG layers conv4_3, fc7 conv8_2, conv9_2, conv10_2, conv11_2 respectively. In order to replicate the equivalent feature maps and still use a pretrained MobileNet model from Keras (Keras only had pretrained MobileNet with input size of 224 x 224 x 3 and not 300 x 300 x 3). The team eventually extracted the following 6 feature maps from Keras pretrained MobileNet v2 28 x 28, 14 x 14, 7 x 7, 4 x 4, 2 x 2, 1 x 1.

With the 6 extracted feature maps, 3 separate sets of convolutions are further applied on each of the feature maps to produce 3 outputs. The first output is a 3-dimensional "Classes" array with a shape of [batch size, height of feature map * width of feature map * number of anchor boxes in each cell of feature map, number of classes] as seen in Fig 2.2a. In our model we are using 5 anchor boxes for each cell. As an illustration, for a feature map of 19 x 19, the size of the second dimension would be 19 x 19 x 5 = 1805.

The second output generates another 3-dimensional "Location" array of the calculated offsets for the anchor boxes. The first 2 dimensions are similar in shape with the Classes output, the third dimension consists of 4 offset coordinates for centroid x, centroid y, width of anchor box, height of anchor box.

And for the third output, another 3-dimensional "Prior Box" array with the actual coordinates and variances of the anchor boxes. The first and second dimension remains similar to output 1 and 2. The third dimension contains the 4 coordinates (centroid x, centroid y, width of anchor box, height of anchor box) + 4 variances. In the original SSD research paper, these 4 variance coordinates were not mentioned but for many of the example codes that tried to replicate the SSD architecture, these 4 variances were added in. The variances were applied to the width and height of the anchor boxes to replicate a gaussian scaling ratio. For our project, we have decided to set the variance = 1 (essentially not adding any scaling effects to the boxes).

The derivation of the anchor boxes and how they are used in the calculations will be further discussed in the sections below. Sections of the codes used in our model can be found in Appendix 7.1 MobileNet SSD

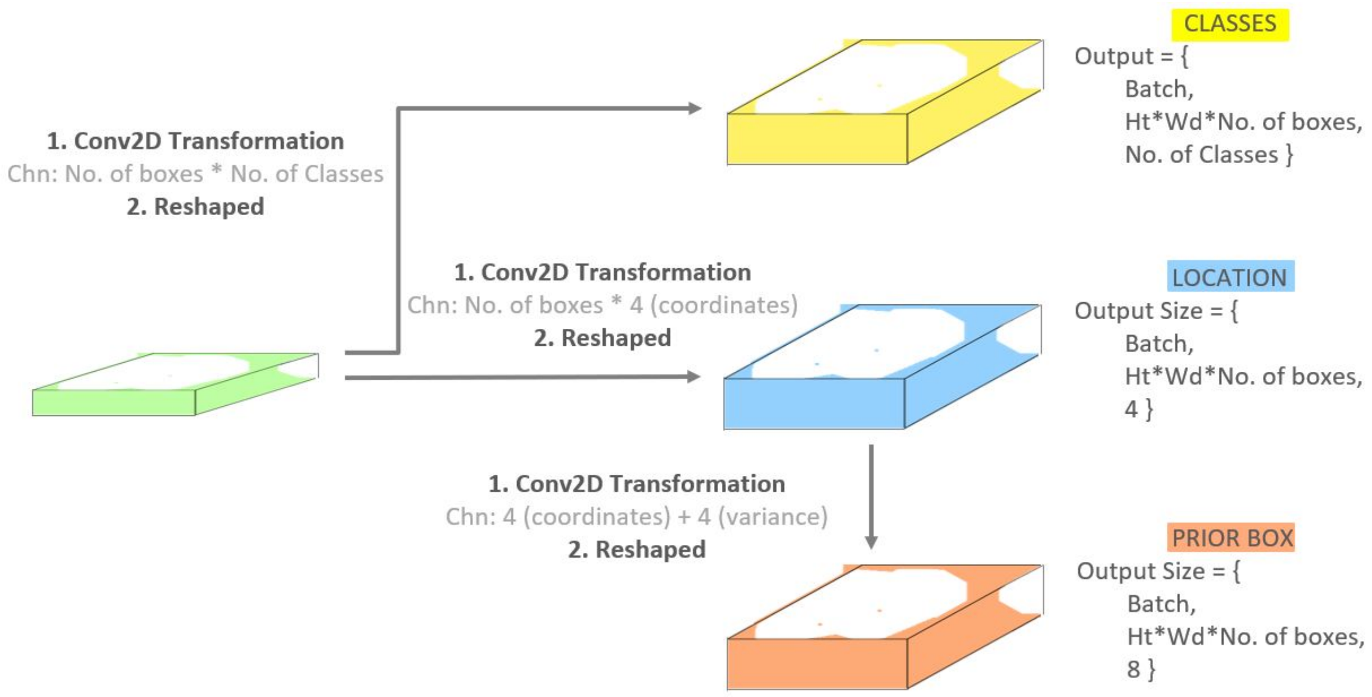


Fig 2.2a: SSD Extraction

The 3 outputs are eventually concatenated to form one big 3-dimensional array as shown in Fig 2.2b. The is the final output that is produced by the MobileNet SSD model.

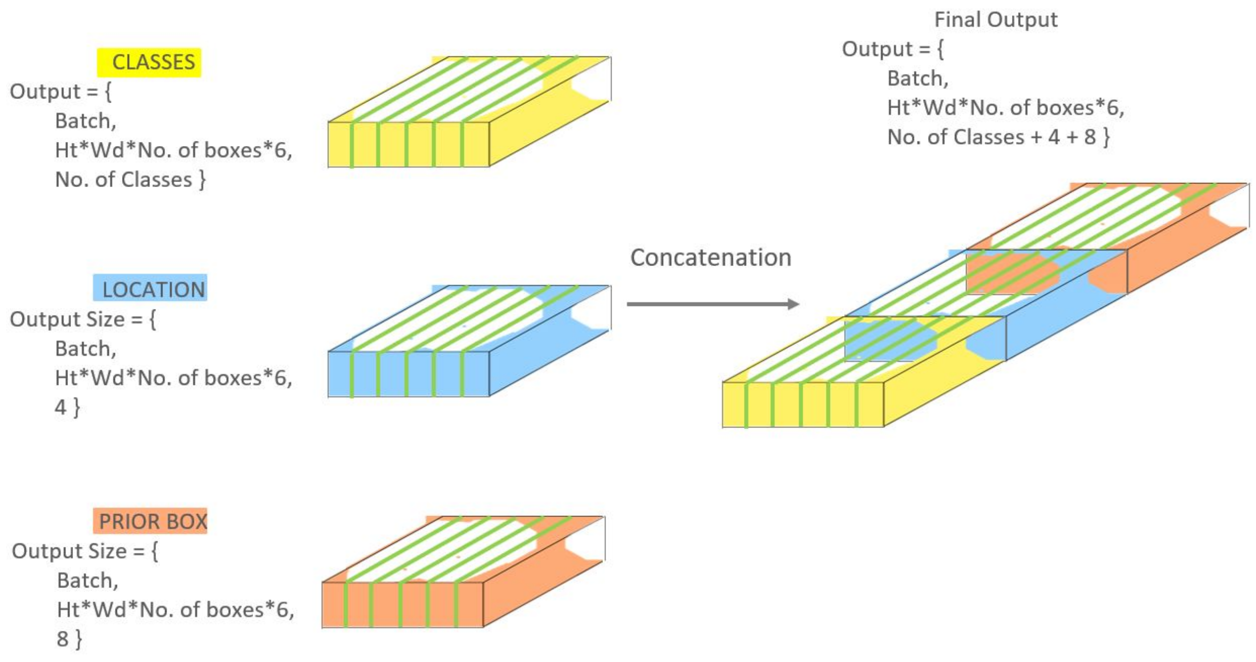


Fig 2.2b: MobileNet SSD Output from the 6 feature maps extracted

2.3. Anchor Boxes

The intuition in creating the SSD is to create a detector on one single shot on objects. The use of fixed grid detectors at multiple scales and at different aspect ratios in SSD is the main idea that powers this detector that sets it apart from the R-CNN.

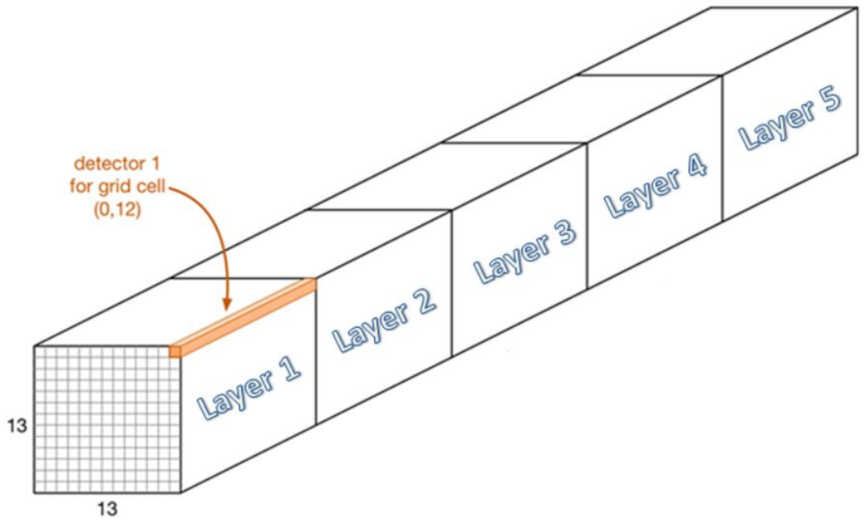


Figure 2.3.1 Feature Map Detector illustration

With reference to figure 2.3.1 , the input image in layer 1 is split into grids of 13 by 13 cells. Where the detectors are placed in the middle of each cell, this illustrates that each detector is responsible for detecting objects at a different location.

The number of cells in the subsequent layers is usually reduced evenly per layer to 1 by 1 cell in the last layer. Each detector at different layers will have a specific task in detecting objects from a “microscopic” view (layer 1) to the “macroscopic” view (layer 5) at different locations in the image.

The size of each detector is calculated with “scales” which is the size of the detector with respect to the input image. In the original paper, the largest scale and the minimum scale are predefined and interim scales are calculated based on the formula below.

$$s_k = s_{\text{min}} + \frac{s_{\text{max}} - s_{\text{min}}}{m - 1}(k - 1), \quad k \in [1, m]$$

Where m is the total number of layers and k is the layer of interest, we can calculate the interim scales of detector which is uniformly distributed through the layers. The max and min scale in the original paper is defined as 0.9 and 0.2 which shows that the detectors of the last layer is responsible for detection for up to 90% (last layer) of the input image and 20% (first layer) per grid cell with respect to the input image.

To further enhance the performance and to make better predictions of the SSD detectors, the original paper proposes using multiple detectors at different aspect ratios in each individual grid cell. The intuition behind this is to have detectors which are specialized in detecting objects of different shapes.

$$a_r \in \{1, 2, 3, \tfrac{1}{2}, \tfrac{1}{3}\} \quad (w_k^a = s_k\sqrt{a_r}) \quad (h_k^a = s_k/\sqrt{a_r})$$

With the given formula, each aspect ratio provides the width and the height of detector boxes at different scales. The original paper provided a set of aspect ratios given {1,2,3,0.5,0.33}. The numbers can be redefined based on the detection tasks of the SSD. As per explanation, the aspect ratios should be as close to the general natural orientations of the object classes that is to be detected where different clustering methods can be used to obtain the optimum aspect ratios for different object detection task. The original paper also provided calculation of a special anchor box which is a geometric mean of the current scale and the next scale. This provides the detector with a detector box size that is intermittent between the current and the next scale.

$$s'_k = \sqrt{s_k s_{k+1}}$$

Detectors at different sizes can be visualized in Figure 2.3.2 below. Where the aspect ratios 1 and Sp. is specialized in detecting square objects and <1 and >1 is specialized to detect tall and long objects respectively. Each set of detectors is then packed into each single grid cells at different aspect ratios shown in Figure 2.3.3.

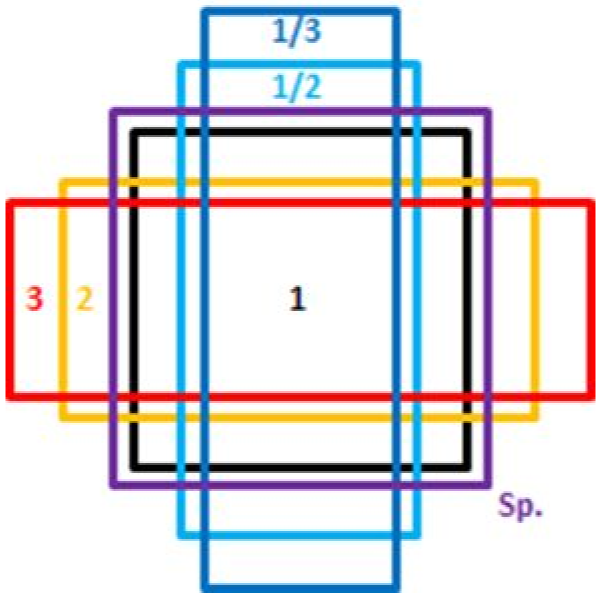


Figure 2.3.2 Anchor Boxes at Different Aspect Ratios

Given the defined set of bounding boxes per grid, it is required to tabulate the positioning of each bounding box to the layers. With the given input size and the number of grid cells on each layer, the width and height of each step size can be determined by dividing the cells with respect to the input shape. The step sizes will then be tiled throughout to find the centroids to each grid cell for detection. Therefore the total number of detections per class can be calculated by multiplying the total number of grid cells in each layer with the number of aspect ratio per cell.

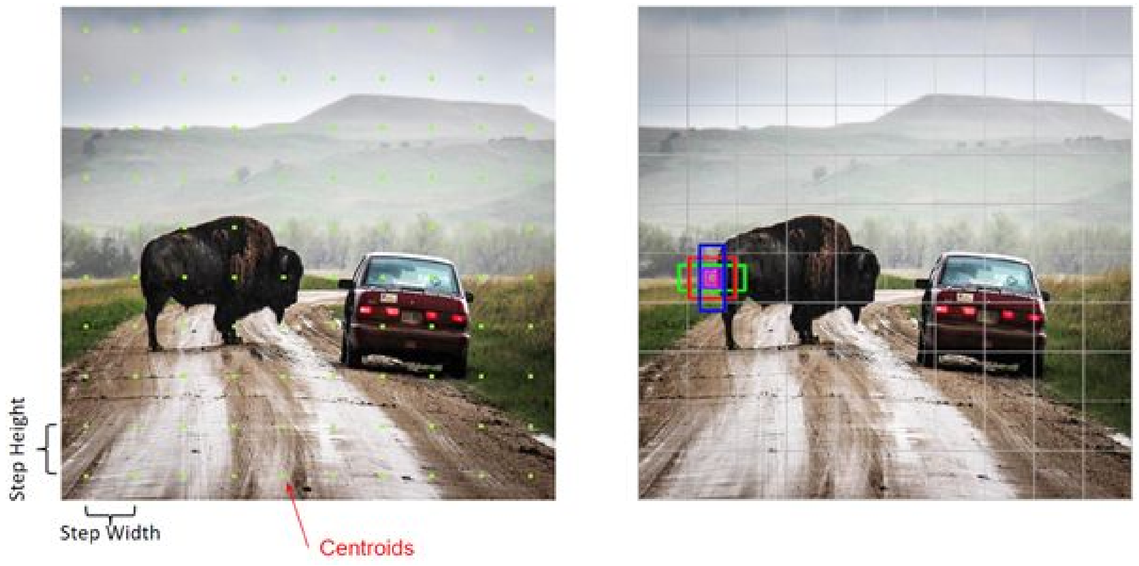


Figure 2.3.3 Anchor Box placement on Image

As proposed by the paper, a set of variances has been vaguely described. To the best of our understanding, the variances denote a Gaussian distribution of adjustments for the anchor boxes. This means that there are no 2 images that can be labelled in the same way therefore allowing a slight “allowable limits” of offsets during a prediction. This is catered to enable robustness while training the SSD to “differences” in the labelled ground truth.

Detailed reference and steps on the creation of anchor boxes are described in the “Anchor Box” class in our codes.

2.4. Encoding Ground Truth

The Ground truth consists of a set of bounding boxes for each image where each bounding box corresponds to a class ID and a set of four parameters representing the corners of the object bounding box. These ground truth labels need to be transformed into a shape that matches the output of the model.

2.4.1. Matching Strategy

All the anchor boxes need to be assigned to either an object class or the background class. The criterion for matching anchor boxes to ground truth is Intersection-over-Union (IOU) which is also known as the Jaccard Similarity. The steps to achieve this have been described in detail below:

Best match

The first step is to find the anchor box that has maximum IOU for each ground truth box. This will ensure that every ground truth box will have at least one anchor box associated with it.

Positive IOU threshold

Thereafter, a thresholding step is applied where every anchor box with a similarity greater than the positive IOU threshold (0.5) is assigned to the corresponding object class. This ensures that not just the best match but all good matches are assigned to the anchor box. The remaining anchor boxes are assigned to the background class.

Maximum Negative IOU

Lastly, a second maximum negative IOU threshold (0.3) is used such that all anchor boxes with similarity measure between the maximum negative IOU threshold and the positive IOU threshold are assigned no class label. This is a critical step in matching anchor boxes as this ensures there is a significant difference between a foreground object class sample and a background class sample for the network to reliably learn the difference between objects and background.

2.5. Loss Function

The SSD loss function is a weighted sum of two components – localization loss and classification loss, which have been both described in more detail below:

- Localisation Loss: The localization loss is smooth L1 loss between the parameters of the predicted bounding box and the ground truth box of all the positive class predictions.
- Classification Loss: The classification loss is the softmax loss over multiple class confidence. The classification loss of all the object class predictions are included in the computation of the total loss. However negative or the background class loss is only included for certain training samples. The top-k background class losses are included such that the ratio of negative training samples to the positive training samples in a batch does not exceed 3. This aforementioned technique is useful to deal with the highly imbalanced set of training samples where the number of background class samples are higher than foreground object class samples.

The hyperparameter α is the weight of the localization loss which was set to 1 after cross-validation.

3. The Dataset

3.1. Data Collection

In the initial stage of the project, the team went about collecting images for 2 classes of object, “Milk”, “Fruits”. The dataset for “Milk” images was a standard carton box shape. However, for the images in the “Fruits” class were significantly more varied. Ranging from apples to oranges to grapes, this class contained images of fruits of various color, size and texture. Our initial model trained on this class label had difficulty identifying the “Fruit” class due to the varied dataset. This will be explained further in the model evaluation portion.

The team decided to reclassify the images and train on “Bananas” and “Apples” instead. For the data collection, the team manually collected more than XX images of each class at the supermarket. To ensure that the model was robust to background noise, the team made an extensive effort to vary the perspectives, environment and lighting when taking photos of the objects. Extra effort was also made to ensure that the dataset had images where part of the object to be detected was occluded. This is to ensure that the model has sufficient varied environments to train on.

3.2. Data Labelling

The images taken were labeled using a free open source labeling tool called LabelImg. The outputs of the labelling exercise produced an XML file with the class labels and coordinates of the boxes. A python script is subsequently used to process the XML files into a readable CSV format and eventually fed into the model for training.

To ensure consistency during the labelling process, the team made extra effort to ensure that all the object classes are labelled on the same basis. For example, the boundary of an “Apple” class would include the stem on the apple and not just the apple itself. This is to ensure data consistency and that the model is training on an accurate set of data.

The team faced a couple of issues during the imaging labelling process due to a bug using the LabelImg software. This will be further discussed section 4.1

3.3. Data Augmentation

Deep networks requires large amount of training data to achieve good performance. In order to build a good image classifier using limited training data, our team used image augmentation to boost the performance of our networks. Image augmentation creates artificial images for training by combining different ways of image processing with geometric transformation, such as random rotation, shears and image transformations like adding noise, blurs and varying contrasts.

Applying geometric transformation will also affect the coordinates of the labelled bounding boxes. Coordinates for each bounding boxes will have to be re-calculated with respect to the augmentation performed.

We created an augmented image generator from scratch using the imgaug library to generate batches of image data with real-time augmentation. imgaug has native support for bounding boxes and augmentations. Details to perform Image augmentation can be referenced in the function “image_augmentation” in our codes.

Although most of the data sets were readily available in machine readable formats (i.e. comma-

4. Model Evaluation

4.1. Iteration 1: The issue with LabelImg

When the team first started training the model, the validation loss continued to increase with each epoch. The team spent a significant amount of effort trying to debug every step of the preparation process right from the labelling of the images. The team discovered that there was a known bug with the LabelImg software (LabelImg Issue, 2018). The images were not labelled in the correct orientation.

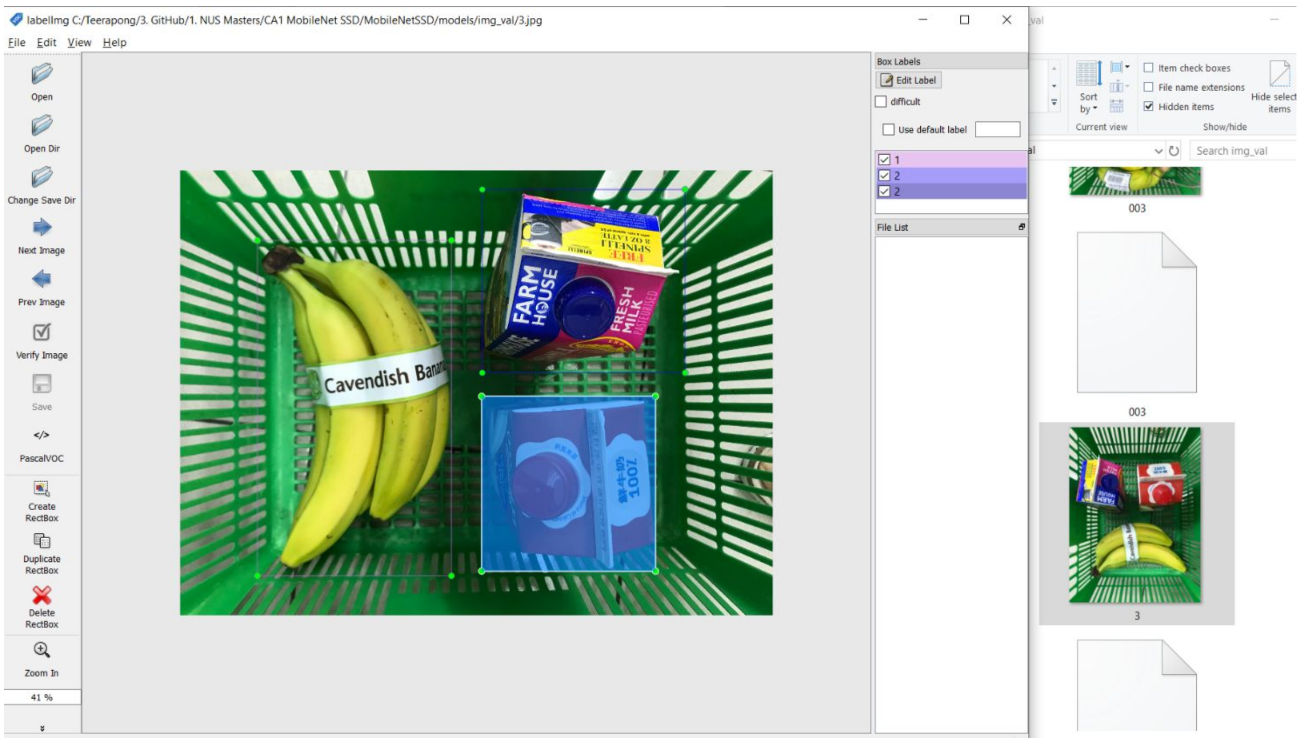


Fig 4.1a: LabelImg Issue

As shown in Fig 4.1a, the original image on the right in windows explorer is showing an upright orientation. When the same image is opened in LabelImg on the left, the image is rotated. Even though the coordinates of the image are labelled correctly with respect to the orientation shown in LabelImg, these coordinates are not translated to the correct orientation shown in windows explorer when it is saved in XML file (OpenCV essentially reads the orientation in windows explorer but uses the coordinates in the XML file which is based on the wrong orientation in LabelImg). This is due to the "orientation" tags on the image taken by the different cameras/mobile phones resulting in some of the images labelled having the correct coordinates and some with incorrect coordinates depending on the "orientation" tag. As of date, LabelImg has not solved this bug.

Because of the incorrect coordinates in most of the training and validation dataset, the model was not able to converge. The team had to manually match the orientation of the images in windows explorer vs LabelImg to get the dataset corrected.

4.2. Iteration 2: Training without pre-trained Keras MobileNet

The team initially started with creating the layers of MobileNet and training from scratch. Even though the training and validation loss was decreasing with every epoch, the loss was still too high. After 500 epochs of training, the validation loss only decreased from 100+ to around 40+. The team was not able to get a consistent loss and accuracy despite the high number of epoch training. This is likely due to the fact that we were trying to train and optimise millions of parameters within the MobileNet layers with just a few hundred images.

The team then decided to use the pre-trained MobileNet Layers from Keras instead. These layers were essentially frozen while the later layers extracted from MobileNet (conv4_3, fc7 conv8_2, conv9_2, conv10_2, conv11_2) were trained. The advantage of transfer learning is that it allowed the use of an optimized feature extractor trained on a significantly much larger training set. Just by changing the MobileNet to the pre-trained model alone, the validation loss on the first epoch was only 5. A significantly improvement vs 500 epoch untrained MobileNet.

4.3. Iteration 3: A Milky Situation

With the label image and pre-trained model issue resolved, another key challenge that the team faced was with the detection accuracy of the model we have trained. The model was able to detect "Milk" categories quite accurately (Fig 4.3a) but it was also always predicting the "Milk" class regardless of the objects in the test image (Fig 4.3b). In our original labelling methodology, we labelled our objects into 2 classes, Milk and Fruits.

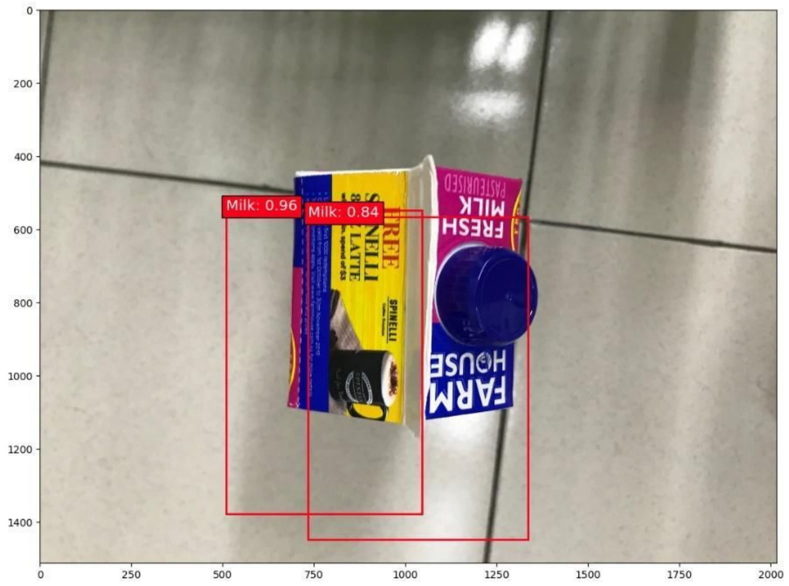


Fig 4.3a: Milk Class Detection

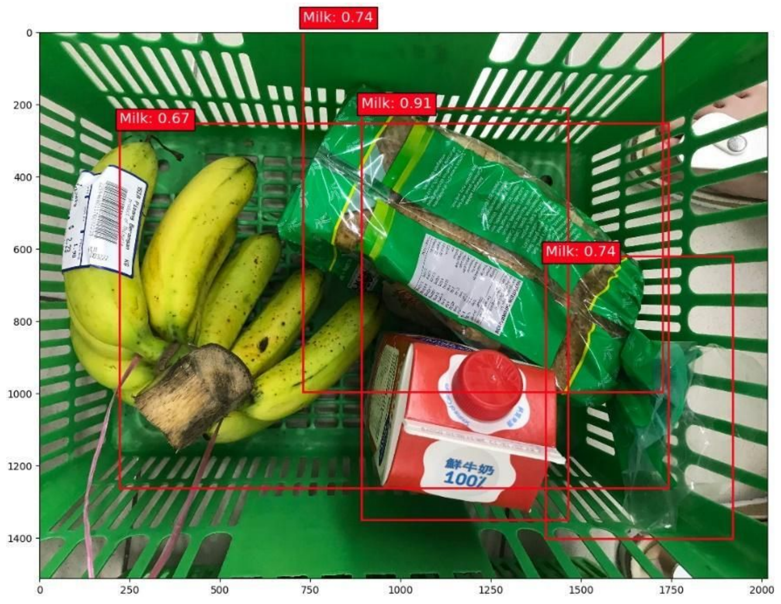


Fig 4.3b: Milk Class Detection Issue

For the Milk category, the contour, shape of the milk cartons were pretty consistent even though we took the images from different angles across different brands of milk. However for the "Fruits" class, our dataset contained a varied set of fruits with different colors and shapes ranging from apples to bananas to oranges to grapes. As such, using only a few hundred images to train the extremely varied "Fruit" class resulted in the model not being able to accurately identify the various types of fruits in the "Fruit" class.

As such, the team went through the trouble or reorganizing and relabeling our image dataset. The team decided to redefine our dataset classes to just "Apples" and "Bananas". And this eventually led to the final iteration of our model discussed in the next section.

4.4. Iteration 4: The Finale

In the final iteration of the model a total of 1000 epochs of training was done. The validation loss was 0.43871 and converged at its local optimum at 385 epochs. It is observed that the saved model performed badly due to overfitting as it can only provide predictions of confidence scores less than 0.1. We backtracked and observed that the model performed better on the weights with higher validation score of 1.3606 as it was able to predict the classes more accurately than the previous few iterations.

The selected weights predicts banana accurately(Fig 4.4a, 4.4b) on most instances but did not perform as well for apples (Fig 4.4c, 4.4d).

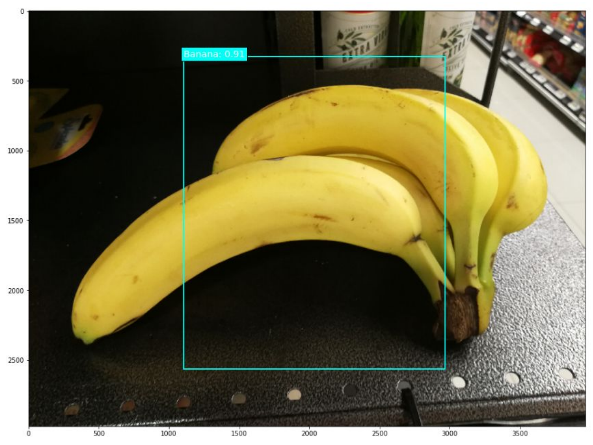


Fig 4.4a: Correct Class Detection

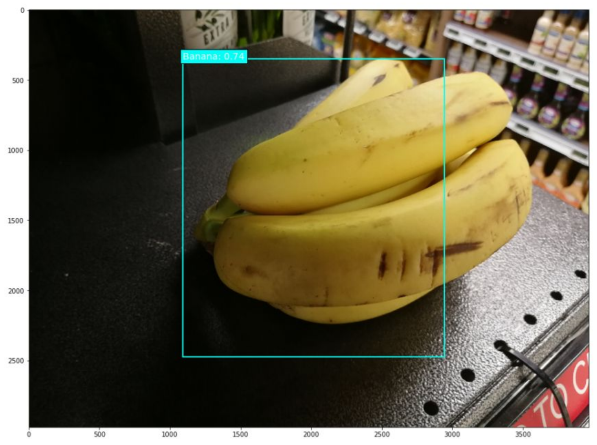


Fig 4.4b: Correct Class Detection

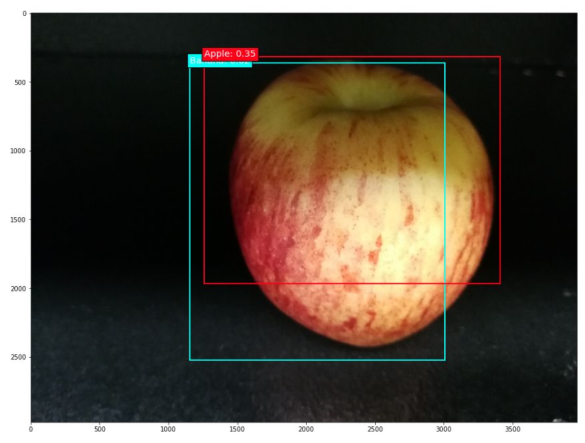


Fig 4.4c: Low Probability Detection

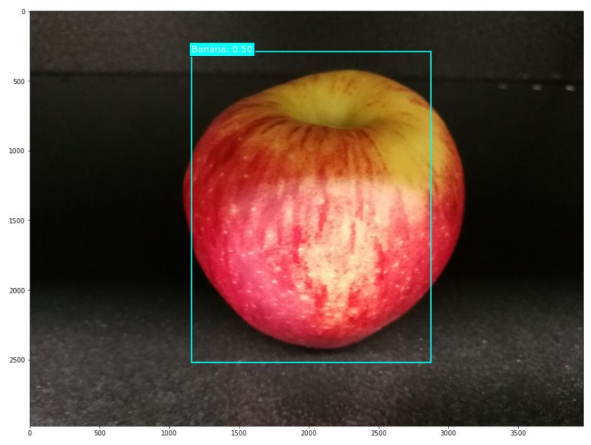


Fig 4.4d False Negative

As seen from the results, the model was able to better predict the classes when the background is clear from background noise (i.e. white/black background) and the object is not occluded in the image. Whenever the object is placed in a complicated background, there often leads to multiple false detection within the image itself (Fig 4.4d). And the probability of detected class is also very low. The model is clearly still far from perfect and being able to correctly identify the correct classes in an image. Given the limited dataset per classes (As compared to millions to images in typical image database), the team was satisfied with the current iteration of the model. The team feels that given more time, there are definitely opportunities to further improve on the model which will be discussed in the next section of the report.

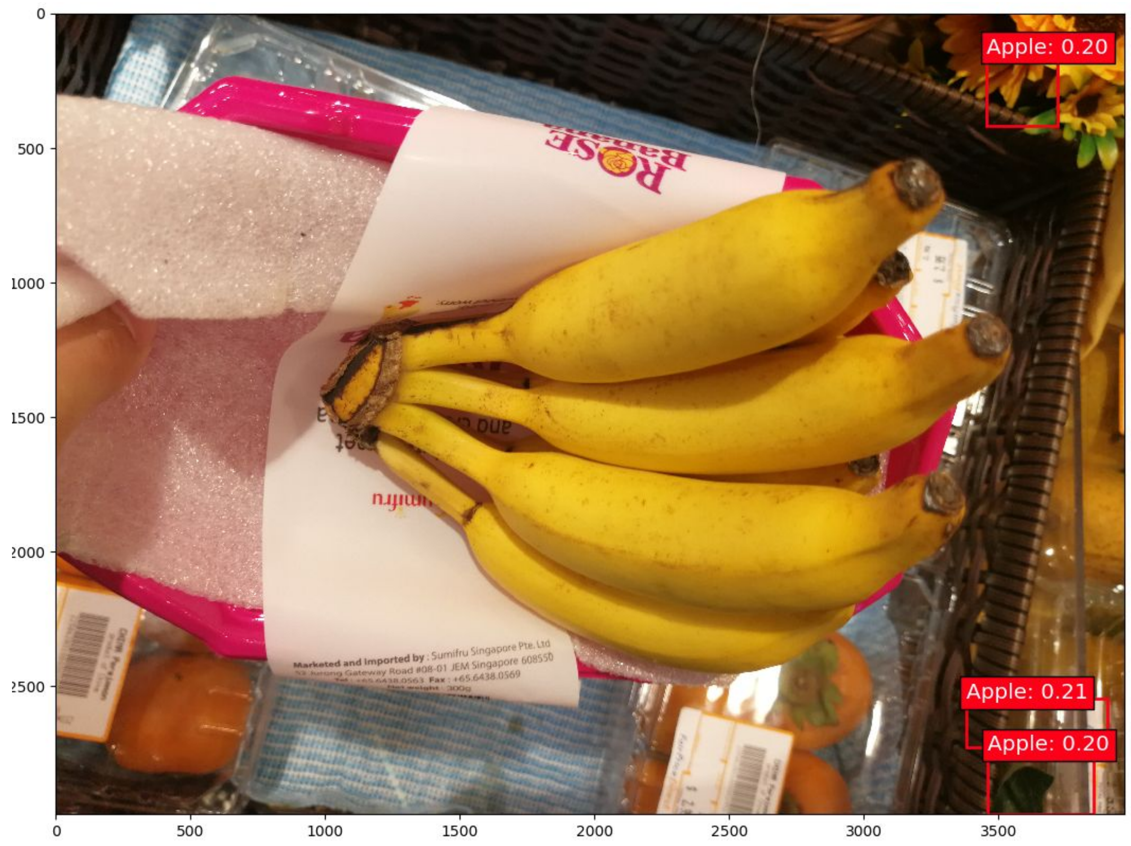
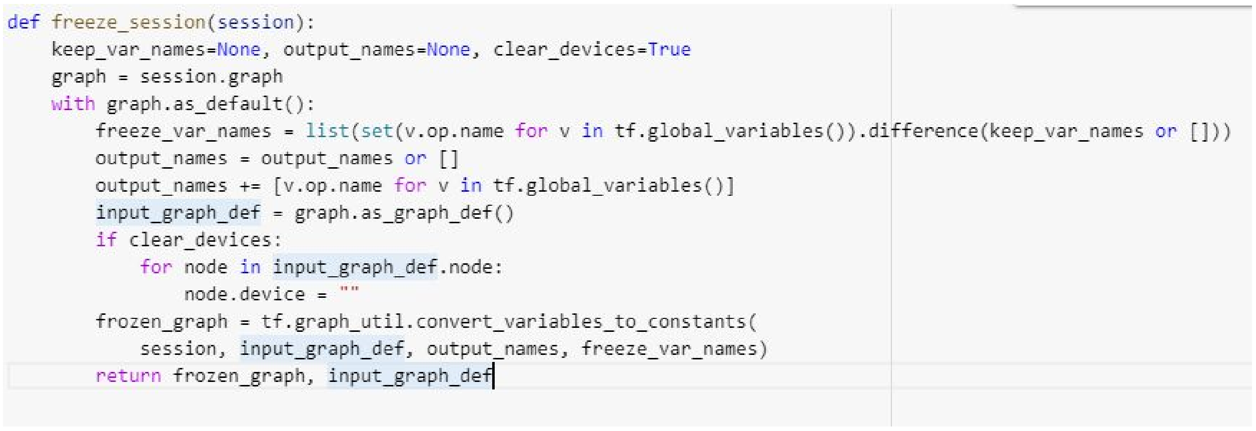


Fig 4.4d: False Negatives with Complicated Background

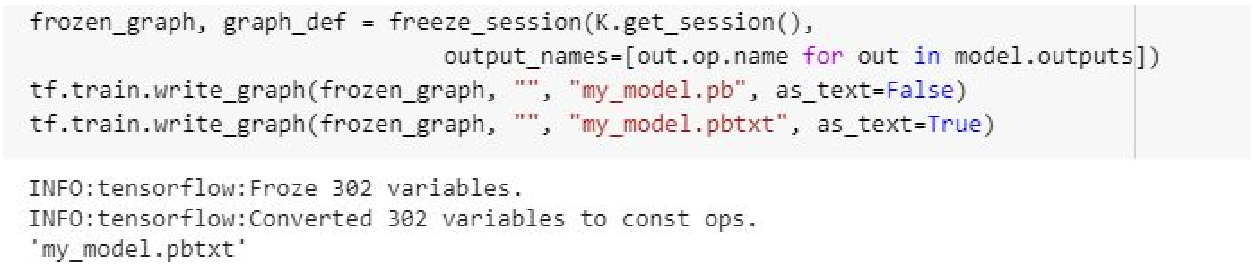
5. Model Conversion

The SSD object detection model was trained in Keras using the TensorFlow backend and saved in the HDF5 format. However, to make the model usable in OpenCV using the functionality in the *dnn* module, it needs to be converted into the protobuf format. The *dnn.readNetFromTensorflow()* API expects two arguments, a binary protobuf (.pb) description of the network architecture and a config file in the protobuf text format (.pbtxt) that contains the text graph definition. While Keras does not define any APIs for this model conversion, TensorFlow provides the functionality to save the model using *tf.Saver* and write graph definition using the *tf.train.write_graph* APIs. The tensorflow graph needs to be frozen with all its variables converted to constants (Figure 5.1) before writing it to protobuf files. The Keras session required for this function can be retrieved using *K.get_session()* where *K* is the variable for the imported Keras TensorFow backend. This frozen graph along with the graph definition retrieved from the session can then be written into protobuf format that is readable by the Open CV *dnn* module (Figure 5.2).



```
def freeze_session(session):
    keep_var_names=None, output_names=None, clear_devices=True
    graph = session.graph
    with graph.as_default():
        freeze_var_names = list(set(v.op.name for v in tf.global_variables()).difference(keep_var_names or []))
        output_names = output_names or []
        output_names += [v.op.name for v in tf.global_variables()]
        input_graph_def = graph.as_graph_def()
        if clear_devices:
            for node in input_graph_def.node:
                node.device = ""
        frozen_graph = tf.graph_util.convert_variables_to_constants(
            session, input_graph_def, output_names, freeze_var_names)
        return frozen_graph, input_graph_def
```

Figure 5.1 Function to freeze the tensorflow graph loaded in the current session



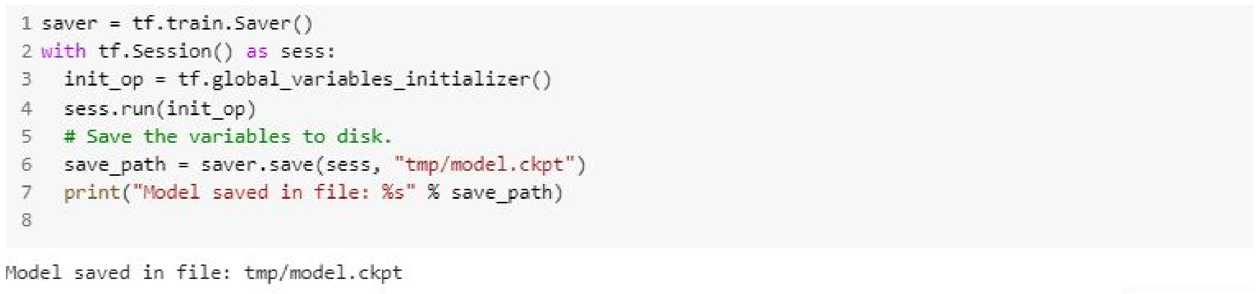
```
frozen_graph, graph_def = freeze_session(K.get_session(),
                              output_names=[out.op.name for out in model.outputs])
tf.train.write_graph(frozen_graph, "", "my_model.pb", as_text=False)
tf.train.write_graph(frozen_graph, "", "my_model.pbtxt", as_text=True)
```

```
INFO:tensorflow:Froze 302 variables.
INFO:tensorflow:Converted 302 variables to const ops.
'my_model.pbtxt'
```

Figure 5.2 Write the frozen inference graph and graph definition in protobuf format

The saved model can then be loaded into OpenCV using the *dnn.readNetFromTensorflow()* API. Though the protobuf text file contains the network architecture in the text format, it fails at this load step with the error that the config file cannot be parsed.

Another attempt is to convert the keras model into tensorflow checkpoint and meta file then reconvert back to prototext and protobuf file shown in Figure 5.3 and Figure 5.4. Both files has been successfully created. Once again the files failed while parsing the 2 files into OpenCV API.

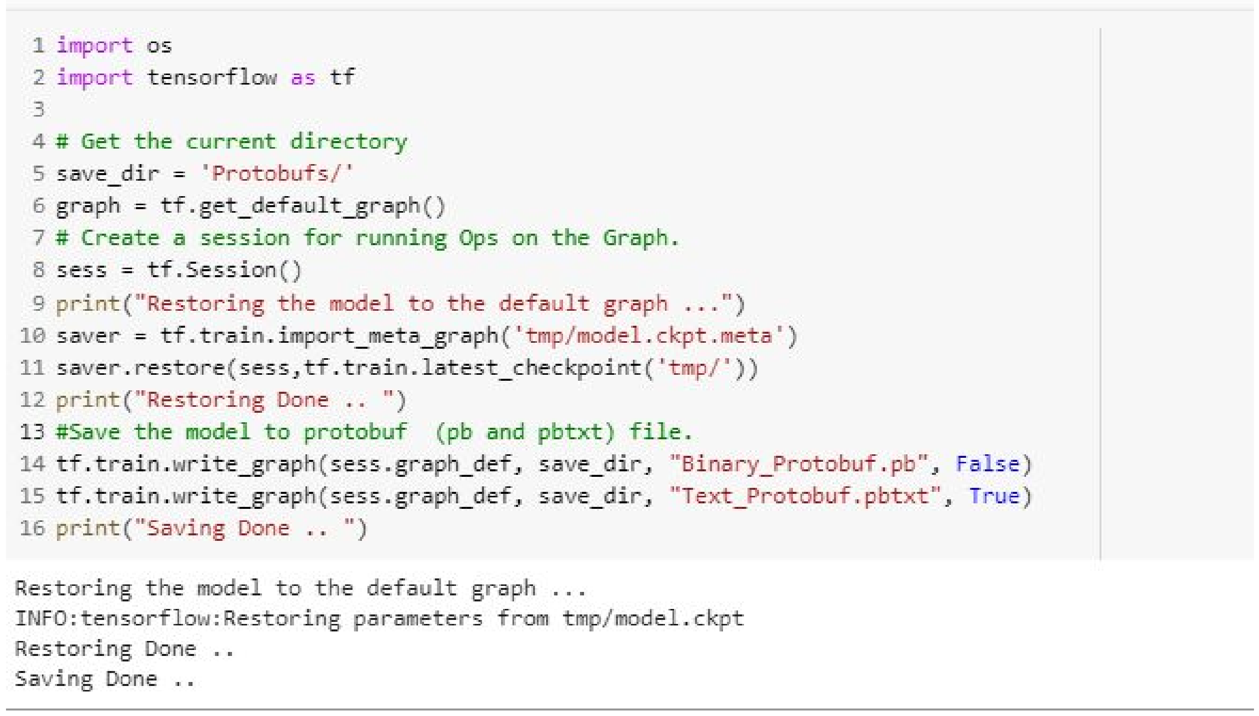


```python
saver = tf.train.Saver()
with tf.Session() as sess:
  init_op = tf.global_variables_initializer()
  sess.run(init_op)
  # Save the variables to disk.
  save_path = saver.save(sess, "tmp/model.ckpt")
  print("Model saved in file: %s" % save_path)

```

```
Model saved in file: tmp/model.ckpt
```

Figure 5.3 Converting model into ckpt files



```python
import os
import tensorflow as tf

# Get the current directory
save_dir = 'Protobufs/'
graph = tf.get_default_graph()
# Create a session for running Ops on the Graph.
sess = tf.Session()
print("Restoring the model to the default graph ...")
saver = tf.train.import_meta_graph('tmp/model.ckpt.meta')
saver.restore(sess,tf.train.latest_checkpoint('tmp/'))
print("Restoring Done .. ")
#Save the model to protobuf  (pb and pbtxt) file.
tf.train.write_graph(sess.graph_def, save_dir, "Binary_Protobuf.pb", False)
tf.train.write_graph(sess.graph_def, save_dir, "Text_Protobuf.pbtxt", True)
print("Saving Done .. ")
```

```
Restoring the model to the default graph ...
INFO:tensorflow:Restoring parameters from tmp/model.ckpt
Restoring Done ..
Saving Done ..
```

Figure 5.4 Converting ckpt model files to .pb and .pbtxt files

After several rounds of troubleshooting, we believe this is likely because of custom Keras layers used in the SSD implementation which do not allow for the straightforward conversion of model formats. The protobuf files as well as the code for conversion is available along with other submission artifacts for reference.

6. Areas of Opportunities

Despite moving through multiple iterations of the model and training on over 2000+ epochs, the team feels that there are still pockets of opportunities to further improve the performance.

Getting more quality data

Data has proven to be a significant factor when it comes to improving the performance of the model. As shown in our earlier iterations, have a class of extremely varied fruits with different contours, colors, textures led to the model only being able to identify Milk classes. As a start, the classes defined should be specific so that the contours and textures of the objects within each class is similar (and we don't have the luxury of training a few million images over a few million epochs).

Next, as a start more images with less noisy background should be taken. We have seen in our last iteration that objects with noisy backgrounds often end up with multiple false negatives. Increasing the number of images with clean and relatively less noisy background should help improve the accuracy of the model.

7. References

Howard, A., Zhu, M., & Chen, B. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision*. Retrieved from Arxiv: https://arxiv.org/pdf/1704.04861.pdf

LabelImg Issue. (2018). Retrieved from Github: https://github.com/tzutalin/labelImg/issues/198

8. Appendix

8.1. Functions List

Section: Bounding Box Utilities

```
def convert_coordinates(tensor, start_index, conversion):
Convert coordinates for axis-aligned 2D boxes between two coordinate formats.
```

```
def intersection_area(boxes1, boxes2, coords='centroids', mode='outer_product'):
Computes the intersection areas of two sets of axis-aligned 2D rectangular boxes.
```

```
def iou(boxes1, boxes2, coords='centroids', mode='outer_product'):
Computes the intersection-over-union similarity (also known as Jaccard similarity)
of two sets of axis-aligned 2D rectangular boxes.
```

```
def match_bipartite_greedy(weight_matrix):
Returns a bipartite matching according to the given weight matrix.
```

```
def match_multi(weight_matrix, threshold):
Matches all elements along the second axis of `weight_matrix` to their best
matches along the first axis subject to the constraint that the weight of a match
must be greater than or equal to `threshold` in order to produce a match.
```

Section [Class]: SSDInputEncoder

```
Transform ground truth labels from list of ndarrays into the shape required
by the model
```

```
def generate_anchor_boxes_for_layer(self,
                                    feature_map_size,
                                    aspect_ratios,
                                    this_scale,
                                    next_scale):
Computes an array of the spatial positions and sizes of the anchor boxes for one
predictor layer
of size `feature_map_size == [feature_map_height, feature_map_width]`.
```

```
def generate_encoding_template(self, batch_size):
Produces an encoding template for the ground truth label tensor for a given batch.
```

Section [Class]: AnchorBoxes

```
A Keras layer to create an output tensor containing anchor box coordinates
and variances based on the input tensor and the passed arguments.
```

Section [Class]: SSD Loss

```
def smooth_L1_loss(self, y_true, y_pred):
Compute smooth L1 loss.
```

```
def log_loss(self, y_true, y_pred):
Compute the softmax log loss.
```

```
def compute_loss(self, y_true, y_pred):
Compute the loss of the SSD model prediction against the ground truth.
```

Section: MobileNet SSD

```
def build_model(image_size,
                n_classes,
                l2_reg=0.0,
                min_scale=0.2,
                max_scale=0.9,
                aspect_ratios=[1, 2, 3, 0.5, 0.33]):
Build a Keras model with SSD architecture.

```

Section: Data Wrangler

```
def read_csv(xml_path, filename):
    This function reads CSV File and returns a list
```

```
def xml_to_csv(xml_directory):
    converts all xml files into a single csv in same directory
```

```
def image_augmentation(filename, data_csv=None, img_dir=None, img_sz=224,
translate=0, rotate=0, scale=1, shear=0,
                       hor_flip=False,
                       ver_flip=False,
                       applyfilter=False):
    This function checks for multiple classes/bounding box in an image, resizes and
augments images and returns augmented images and bounding box co-ordinates
```

```
def image_batch_generator(img_dir, csv_data, steps_per_epoch, batch_size,
label_encoder, img_sz=224, translate=0,
                          rotate=0,
                          scale=0, shear=0, hor_flip=False, ver_flip=False,
applyfilter=False):

  Generator which returns batches of numpy array consisting of 2 numpy arrays for
training
```

```
def data_generator(img_dir, xml_dir, label_encoder, batch_size=None,
steps_per_epoch=None, img_sz=224,
                   translate=0, rotate=0, scale=0, shear=0, hor_flip=False,
ver_flip=False, applyfilter=False):
    Generate batches of tensor image data with real-time data augmentation from image
and xml directory. The data will be looped over (in batches).
```

<u>Section: Decoder</u>

```
def _greedy_nms(predictions, iou_threshold=0.45, coords='corners'):
    The same greedy non-maximum suppression algorithm as above, but slightly modified
for use as an internal function for per-class NMS in `decode_detections()`.
```

```
def decode_detections(y_pred,
                      confidence_thresh=0.01,
                      iou_threshold=0.45,
                      top_k=200):
    Convert model prediction output back to a format that contains only the positive
box predictions
    (i.e. the same format that `SSDInputEncoder` takes as input).
    After the decoding, two stages of prediction filtering are performed for each
class individually:
    First confidence thresholding, then greedy non-maximum suppression. The filtering
results for all
    classes are concatenated and the `top_k` overall highest confidence results
constitute the final
    predictions for a given batch item.
```