[Team 30] Project F4: Plant Phenotyping

Adithya Raghu Ganesh
Department of Computer Science
NC State University
araghug@ncsu.edu

Prasanth Yadla
Department of Computer Science
NC State University
pyadla2@ncsu.edu

Sharath Narayana
Department of Computer Science
NC State University
snaraya9@ncsu.edu

***Abstract*—Plant phenotyping of a plant is describing the visual or observable characteristics such as height, biomass, leaf shape and so on. In this project we are using phenotyping to find the water requirement of the plant [1]–[3].The number of leaf tips, width of the plant and collars of the plant can be phenotypic indicators of water stress.**

Our aim is to apply Image Processing and Neural networks to given images to detect leaves and collars as key points. This would help in identifying the phenotype features of the plant in an automated fashion. Specifically, we used the Tensorflow Object Detection API, an open source framework built on top of Tensorflow to localize and identify leaf tips and collars. We have achieved over 55% accuracy in detecting the leaf tips using Inception Network out of the total manual count and also detected collars to a reasonable accuracy.

I. Model Training and Selection

A. Data Pre-processing

This project was implemented using python along with the associated libraries such as tensorflow[4], protobuf[5], Numpy, OpenCV ., etc. By using the .mat files generated from the manually annotated images using MATLAB, we wrote an automated script that parses the co-ordinates and converts them into python friendly .dat format file. Doing this, decouples any usage of MATLAB and it's respective formatted files. We had then written another script to read the contents of the .dat file and output a CSV-formatted file containing required information to generate the TFRecord Data File. The CSV formatted file contains the following columns :-

1) Image Filename (location)
2) width of image
3) height of image
4) xmin (lower x co-ordinate of bounding box)
5) ymin (lower y-co-ordinate of bounding box)
6) xmax (upper x-co-ordinate of bounding box)
7) ymax (upper y-co-ordinate of bounding box)
8) class (leaf or a collar)

Each row in this CSV formatted file, consists of data related to exactly one object (be it collar or a leaf-tip). There were a total of **476 images** and a total number of rows or **detection objects of 5913.** This exact format is required to generate the portable and serializable TFRecord format file, used by Tensorflow. We then used OpenCV and tensorflow to convert this data to the TFRecords object. We also split the total data into train and test before generating their respective TFRecords.

B. Framework and Environment

The Framework we used was Tensorflow Object Detection API[6]. The Framework consists of pre-loaded libraries [7] and deep learning models which we can directly plug in and train it according to our needs.

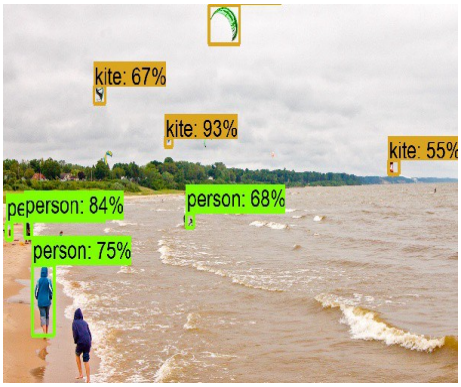

Fig. 1: Example Object Detection using Tensorflow Object Detection API

The computer we used for this is one of the nodes of the CSC ARC Cluster. The Machine Configuration is as follows :-

1) Platform and OS:- CentOS Linux, v7.7.1908
2) RAM :- 92GB
3) GPU :- NVidia Quadro P4000
4) CPU :- 16 Intel Core i5 processors

Initially, we used the framework on a local CPU machine, but as it was taking long, we migrated to this environment. The process was to clone the framework, install the necessary python packages in a virtual environment using pip, change the pbtxt file to suit the number of classes, place the training data at the required location, change the main model config file to the desired one and then run the command to train it using the latter config model.

The training is a laborious process, with each step taking around 1-1.5 seconds. So, totally, for the loss of the object detection model to decrease to a reasonable level, we had to train for over 12,000-15,000 steps, sometimes, to over 22,000 steps. This could take at least 2-3 hours.

For using different CNN-based models, we had to download the initial model checkpoint for that model and load the respective configuration file (which usually comes with the

framework). Then, we run the training command pointing to this config file as one of the command line arguments. The trained model checkpoints are saved and the inference graph is loaded into a separate object detection script to detect it on the test image samples.

C. *The Model*

We selected, tested and trained a wide variety of CNN-based machine learning models for our use-case.They include SSD RESNET, FASTER RCNN INCEPTION, MASK RCNN INCEPTION, SSD Inception COCO V1, SSD Inception COCO V2, SSD MOBILENET COCO V1 [8] and SSD MOBILENET COCO V2 with tensorboard visualization while training.We found that the latter four models worked well, with SSD Inception COCO V2 giving the best performance. The final detection on test images was done using modified object detection script.

Inception Net

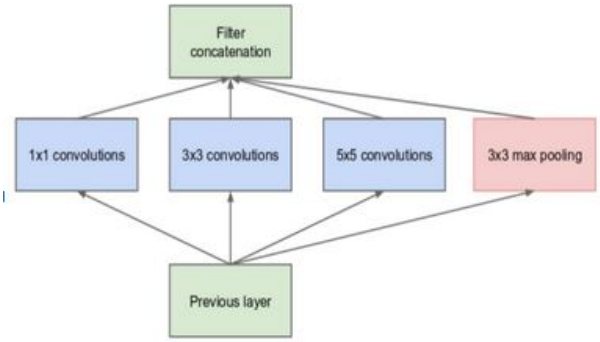

Fig. 2: Inception Network V2 Architecture

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Fig. 3: Inception Network V2 Layers Info

The Inception network V2 first introduced by Szegedy et. al, at Google Inc. [9] works when salient features are of varying size. With different size we won't know the right kernel size for the convolution network. It uses multiple filters of different size on the same level and finally concatenates the convolutions as shown in the architecture diagram. There are over **22 Layers 5 and million parameters trainable** . The number of layers are comparatively less in this network. Max pooling is also performed in between layers which helps the right salient feature size to be selected while training. After multiple trail and errors we found this model gave a good accuracy for our problem statement.

D. *Baseline*

MobileNet V1 is a CNN architecture model for Image Classification and Mobile Vision. It uses low computation without any need for heavy hardware support. MobileNet works on mobile and embedded systems. It is depth wise separable convolution and single filter based Neural Network. There are **53 layers and 3.47 million parameters**

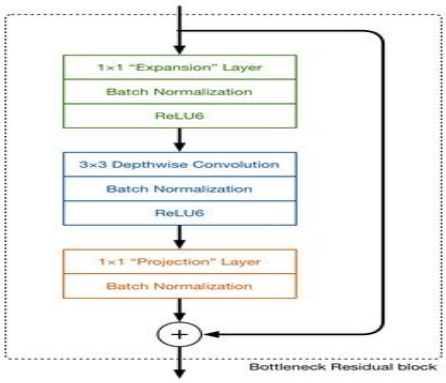

Fig. 4: MobileNet V1 Architecture

The layer information is shown here :-

II. Experimental Section

A. *Metrics*

The main metric we used to compare our manually annotated images vs those detected by the object, is the ratio of the number of leafs/collars detection to those of the manually annotated image.

We also find the average number of objects detected my a given model and plot it over the number of steps. We also use this to serve as a reference to compare different architectures and models.

Additionally, we plot the loss of the model over the number of steps using tensorboard, which can serve a good visualization as to how the model is performing.

B. *Model Selection*

Some Basic configurations available in config file in JSON format are as follows.

1) height and width scale
2) matched threshold
3) iou similarity
4) min_depth, max_depth
5) num_layer before predictor
6) use_dropout

Table 1. MobileNet Body Architecture

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

Whole Network Architecture for MobileNet

Fig. 5: MobileNet V1 Layers

7) kernel size
8) activation function
9) regulizer (L1, L2 .,and weight)
10) batch norm
11) classification loss
12) localization loss
13) optimizer type including learning rate, decay steps, decay factor, momentum_optimizer_value, epsilon

Hyper parameter selection or tuning is the process of selecting the right hyperparameter values for the algorithm. The right values gives the optimal accuracy for the algorithm.

As shown below we tuned multiple hyperparameters [10]. The learning rate was reduced as there was over-fitting while we had higher values. The image size was reduced as larger image size gave us memory and buffer issues while executing our code. Further we tried multiple activation functions such as Relu, TanH and Sigmoid. Few other hyperparameters included threshold,batch size and number of steps.

- Learning rate:
  - Used 0.001, 0.004 and 0.00004
- Detection Threshold:
  - From 0.5 to 0.4 and 0.3
- Image size:
  - 1200*1080 to 600*600 and 400*400
- Activation function:
  - Relu, Sigmoid, TanH
- Batch Size:
  - From 24 to 64, 128
- Number of steps:
  - From 20000 to 15000

Using the above mentioned changes, we tried to achieve the best loss over steps.

C. *Performance and Comparison to Baseline*

Firstly, we compare the loss variation from the baseline(Mobilenet v1) to our final model (inception v2), along with few other models we have used (like Mobilenet v2). We then compare the average objects detected and show several test images used for inference.

*1) **Loss Variation**:* A loss function is used to optimize the parameter values in a neural network model. Loss functions map a set of parameter values for the network onto a scalar value that indicates how well those parameter accomplish the task the network is intended to do. The loss functions for each of the network is shown below. The dark red line indicates the mean value of the loss over training steps.

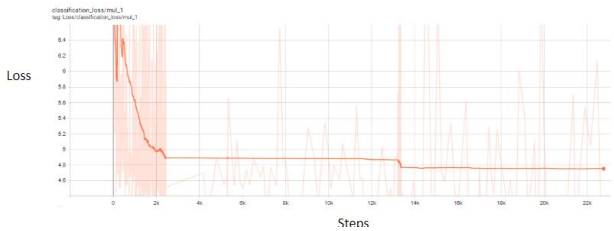

Fig. 6: Mobile Net V1 Loss

In the above image we can see the loss values of the mobile net v1. The loss value varied a lot initially and steadied over time. The value is also reduced over the number of steps. The loss mainly varied from 3 to 6.

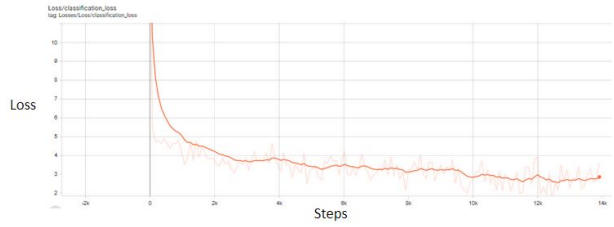

Fig. 7: Mobile Net v2 Loss

In the above image we can see the loss values of the mobile net v2. The loss mainly varied from 2 to 4 and followed the same pattern as the V1.

In the above image we can see the loss values of the Inception net v2. The loss mainly varied from 2 to 2.5 which is a low value compared to the above models.

*2) **Average Images detected**:* This sections shows the average prediction graphs where the comparison between actual count and the predicted count is shown below. The first image shows the mobile net prediction which varied between 2 to 3. The next image shows the inception network predictions value

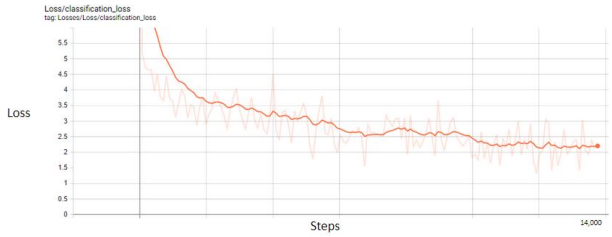

Fig. 8: Inception Net v2 Loss

which varied between the 3 to 5. This shows that inception network had a better prediction accuracy compared to that of the mobile network predictions.

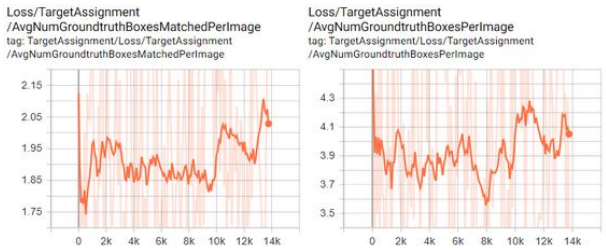

Fig. 9: Avg object found Mobile Net

The above figure shows the mobile network prediction graph. The left hand side shows average prediction count and the right hand side the actual count.

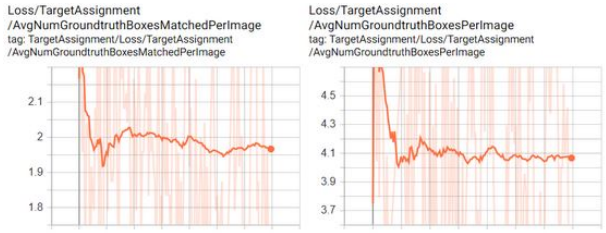

Fig. 10: Avg object found Inception Net

The above figure shows the inception network prediction graph. We can see that this one is more close to the actual compared to the previous model networks.

III. Accuracy and Results

The efficiency of the model can be identified only using the accuracy measure. We used loss and detection ratio as the two metrics to calculate the accuracy of the models trained. The first table below shows the loss and the prediction ratio of the model. As Mobilenet V1 didn't classify or detect any leaf tips we kept that as our baseline. Mobilenet V2 gave us an prediction ratio of 25 and loss around 3.125 which is comparatively less that V1. Inception net gave us better results compared to MobileNet with a loss of 2.25.

Mode	Loss	Ratio
MobileNet V1 (baseline)	4.754	0
MobileNet V2	3.125	45
InceptionNet V2	2.25	55

The table below shows the accuracy of detection of the tips in the image. Baseline gave us an accuracy around 10 to 20 percent and inception net around 85.

Model	Accuracy
Baseline	10-20
Inception Net V2	80-85

The result or output of this project would be the plant with the bounding box or the tips which indicate the leaf and collar. We used a threshold of 0.7 score to confirm that its a leaf of collar. Each of the image given below is an image from the testing set where the model predicted the tips for the leaf and collar.

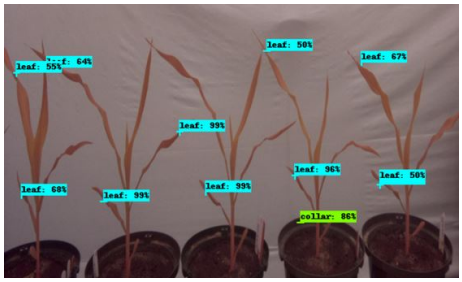

Fig. 11: Test Image 1

In the test image 1 we can that about 10 leaf tips have been detected in the image including 1 collar. We can see that there are no wrong detection in the image.

Fig. 12: Test Image 2

In the test image 2 we can see around 5 leaf tips and 2 collar tips have been detected, but we can see that one collar have been wrongly detected which is a cause of over-fitting of the model.

In the test image 3 we can again see multiple leaf and one collar being detected in the image. Even though few of the leaves have not been detected, there are no wrong classification.

IV. Conclusion

After training multiple models and tuning a large number of hyperparameters we selected inception network based model and that trained to identify leaf tips and collars on images. Also there where few other models which was used but didn't give any positive results. Few of those models where SSD

Fig. 13: Test Image 3

Resnet V2, Faster RCNN due to memory management issues in the server and lack of good hardware support. In the future, we plan to obtain better hardware to run more complex models for better detection.

Data Augmentation can also be tried to get more training images to train the model well. Images could have been cleaned well as color of the images varied a lot which could have also hampered the tip detection. We can change maintain a uniformity in the coloring pattern of the image to get better results in the future.

REFERENCES

[1] Michael P. Pound, Jonathan A. Atkinson, Alexandra J. Townsend, Michael H. Wilson, and Griffiths. Deep machine learning provides state-of-the-art performance in image-based plant phenotyping. *GigaScience*, 6(10), 08 2017.

[2] Aniruddha Tapas. Transfer learning for image classification and plant phenotyping. *International Journal of Advanced Research in Computer Engineering and Technology (IJARCET)*, 5(11):2664–2669, 2016.

[3] Md Mehedi Hasan, Joshua P Chopin, H. Laga, and Stanley J. Miklavcic. Detection and analysis of wheat spikes using convolutional neural networks. In *Plant Methods*, 2018.

[4] https://github.com/tensorflow/.

[5] https://github.com/protocolbuffers/protobuf.

[6] https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202.

[7] https://towardsdatascience.com/custom-object-detection-using-tensorflow-from-scratch-e61da2e10087.

[8] https://medium.com/analytics-vidhya/image-classification-using-mobilenet-in-the-browser-b69f2f57abf.

[9] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[10] https://pythonprogramming.net/introduction-use-tensorflow-object-detection-api-tutorial/.