INVESTIGATION OF GREEN STRAWBERRY DETECTION USING R-CNN WITH VARIOUS ARCHITECTURES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Daniel Rivers

March 2022

COMMITTEE MEMBERSHIP

TITLE:	Investigation of Green Strawberry Detection Using R-CNN with Various Architectures
AUTHOR:	Daniel Rivers
DATE SUBMITTED:	March 2022
COMMITTEE CHAIR:	Jane Zhang, Ph.D. Professor of Electrical Engineering
COMMITTEE MEMBER:	Wayne Pilkington, Ph.D. Professor of Electrical Engineering
COMMITTEE MEMBER:	Helen Yu, Ph.D. Professor of Electrical Engineering

ABSTRACT

Investigation of Green Strawberry Detection Using

R-CNN with Various Architectures

Daniel Rivers

Traditional image processing solutions have been applied in the past to detect and count strawberries. These methods typically involve feature extraction followed by object detection using one or more features. Some object detection problems can be ambiguous as to what features are relevant and the solutions to many problems are only fully realized when the modern approach has been applied and tested, such as deep learning.

In this work, we investigate the use of R-CNN for green strawberry detection. The object detection involves finding regions of interest (ROIs) in field images using the selective segmentation algorithm and inputting these regions into a pre-trained deep neural network (DNN) model. The convolutional neural networks VGG, MobileNet and ResNet were implemented to detect subtle differences between green strawberries and various background elements. Downscaling factors, intersection over union (IOU) thresholds and non-maxima suppression (NMS) values can be tweaked to increase recall and reduce false positives while data augmentation and negative hardmining can be used to increase the amount of input data.

The state of the art model is sufficient in locating the green strawberries with an overall model accuracy of 74%. The R-CNN model can then be used for crop yield prediction to forecast the actual red strawberry count one week in advance with a 90% accuracy.

Keywords: Image Processing, Deep Learning, Selective Segmentation, Convolutional Neural Network, Data Augmentation, Negative Hard Mining, Crop Yield Prediction

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1
INTRODUCTION

Fruit detection and crop yield prediction are problems that have existed for decades. Currently, the total number of red strawberries are approximated by manually counting a portion of the crop and extrapolating it across the entire field, but artificial intelligence can be used to automate the process. Traditional solutions involve using an image processing approach of choosing key features to extract. Then green strawberries can be detected using these key features. The implementation outlined in this paper uses R-CNN (region based convolutional neural network) as a state-of-the-art deep neural network for end-to-end green strawberry detection. The densely layered network will be able to detect the most subtle nuances between the green berries and the leaves, given enough data.

When the deep architecture is fine-tuned, R-CNN should be able to detect the number of green strawberries better than traditional approaches. With this green strawberry count, we can subsequently use crop yield prediction to forecast how many red strawberries will be grown in one weeks time. An understanding of the strawberry's growth cycle is needed in order to predict its growth.

1.1 Statement of Problem

California produces approximately 88% of the strawberries grown in the US [1]. Peak strawberry season occurs between April and August, but the coastal climate allows strawberries to be harvested year round [1]. The year round strawberry season makes the crop heavily labor intensive. The multi-billion dollar industry has experienced labor shortages for years with the worker shortage being exasperated during the pandemic [2].

With such a long harvesting season, it’s difficult to know how many people to hire on any given day. Portions of the field are manually counted for red strawberries to better understand how many people are needed. Berries are hand picked to ensure high quality fruit and typically harvested every three days during peak season [1]. If strawberries are not picked they will rot, resulting in fruit flies and spiders that can spoil large portions of the field. Due to labor shortages, it’s difficult to hire enough workers that are willing to work long hours harvesting the crop [3]. Hiring the right number of people based on the number of ripened fruit will result in a more efficient field with better fruit yield.

1.2 Data Collection

A dataset of strawberry images and clippings was curated using photos taken at Cal Poly SLO’s verticillium field from 2018 and 2019. The images were taken between February and June of their respective years using various smartphone cameras. Drone images were tested previously but lacked the resolution to successfully detect the berries. Each field image was taken approximately 5 feet above the strawberry plant cluster with a blue reference object in frame to understand the local distance. Pictures were taken twice a week in the morning to mimic the rate strawberries are picked during peak harvesting season. Figure 1 below shows the verticillium field and how various beds were planted that year.

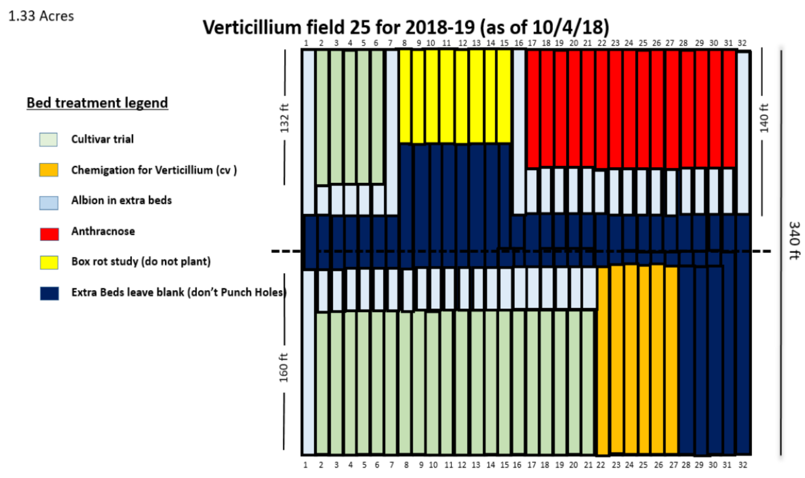


Fig. 1. Verticillium Field 25 for 2018-19

A section of 28 strawberry clusters were marked off for the data collection. These plants weren't harvested to ensure all the fruit would be present when taking the photos. Each strawberry cluster contains four plants. A field image is shown in Figure 2 below. A total of 447 field images were taken in 2019, each containing the same blue reference object which is exactly one square foot.

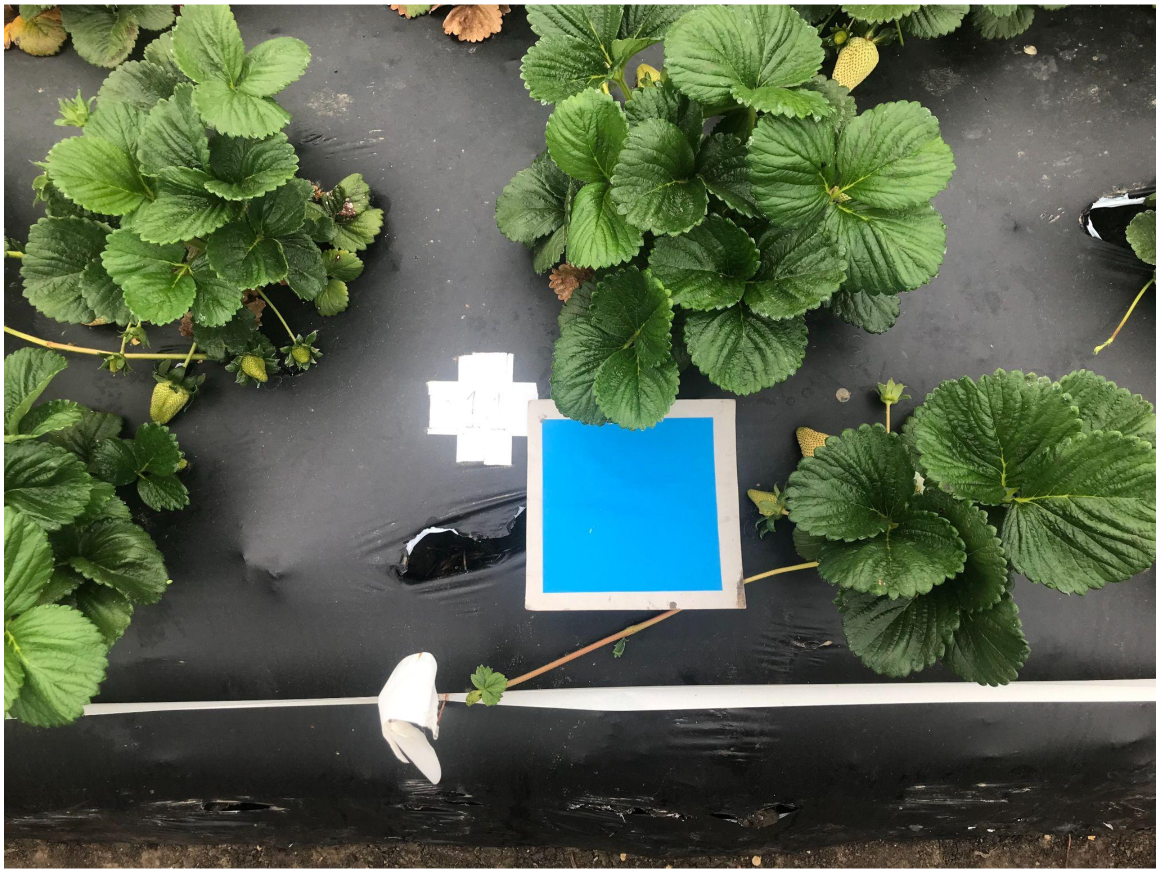


Fig. 2. Field Image c11-4-27-19

The above image is a picture of cluster 11 early in the morning. The luminous lighting reduces shadows which will help in locating each green strawberry. The bottom left plant in cluster 11 didn't last past February, giving this cluster only three plants to detect berries from. Fig 3 below shows a field image from cluster 12 taken closer to noon. The sun casts long shows making it more challenging to detect the green strawberries.

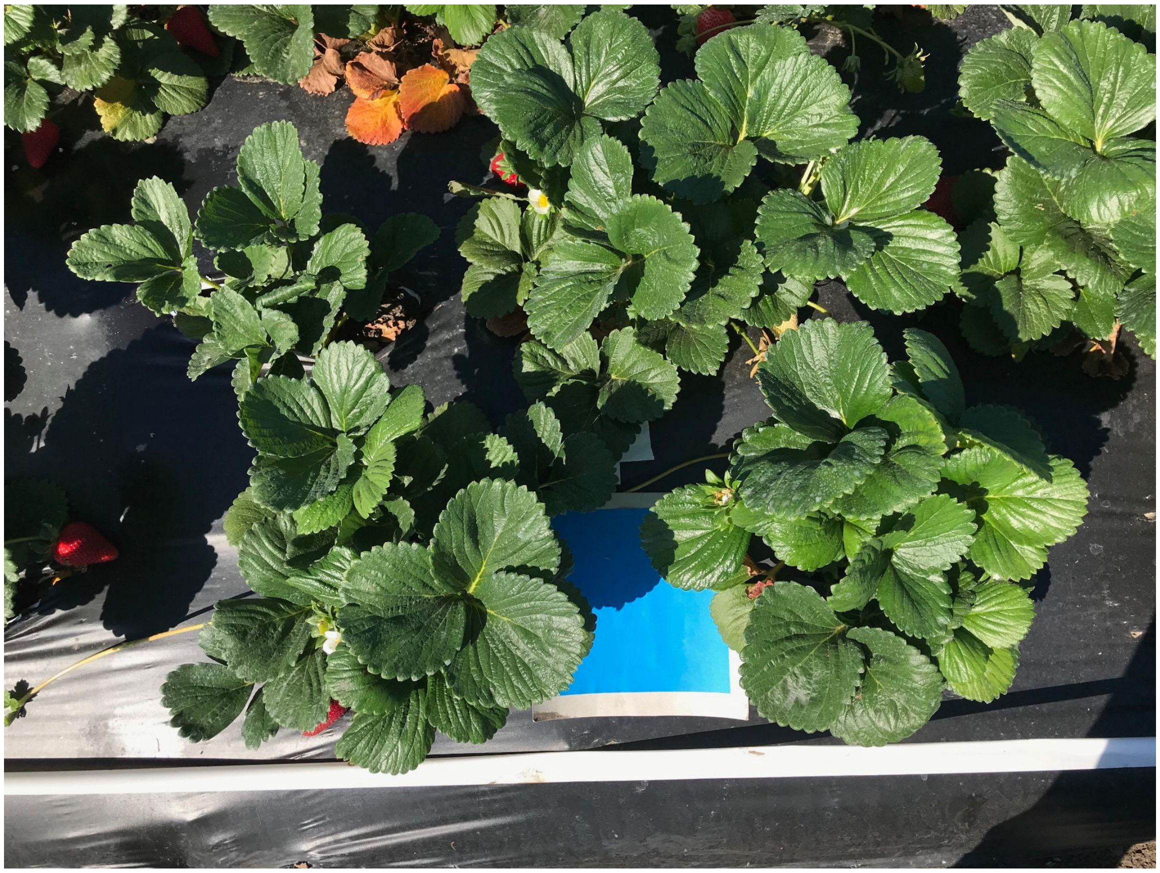


Fig. 3. Field Image c12-5-11-19

For this project, we are choosing to detect green strawberries instead of red ones for two main reasons. The first reason is green strawberry data is more useful for the farmer in regards to hiring workers. Green strawberries typically take one week to mature into a fully ripened red berry, so ideally the total number of green strawberries will strongly correlate with the number of red strawberries that will be harvested in one week. This will give the farmer time to hire the appropriate number of workers. Telling the farmer the total number of red strawberries that same day wouldn't be nearly as helpful. The second reason is that green strawberries are more difficult to detect and are a challenging input for the deep learning methods explored in this thesis topic. Previous work has already been done using neural networks for red strawberry detection with satisfactory results. Cal Poly student Yavisht Fitter implemented a network based off of VGG to detect red strawberries with an 88% accuracy [4]. The data he used from 2018 will be combined with the images taken in 2019 to form a larger dataset to detect green strawberries.

1.3 What is a Green Strawberry?

The lifecycle of a strawberry starts as a flower and ends as a large red berry. The middle stages span from a bud to a larger green berry. The term "green strawberry" is quite vague as all of the middle stages could appear to fit that description. Figure 4 below shows the lifecycle of one strawberry as it matures over one month. The ideal green strawberry is shown in image 5. The berry has reached its maximum size and has begun to turn more pale. Over the next week the berry will gradually turn red and ripen. This is the green strawberry that we want to detect.

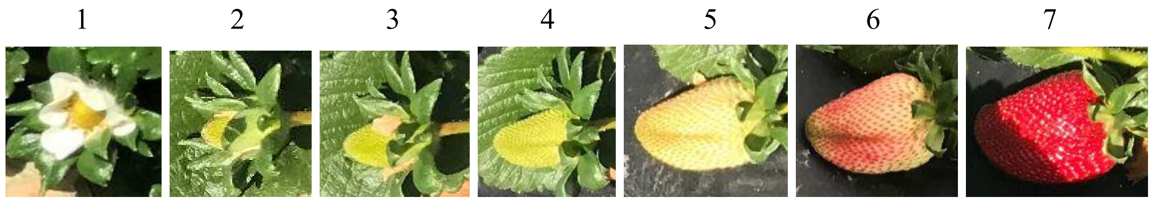


Fig. 4. A Strawberries Life Cycle from 4-6-19 to 5-1-19

Now that we have an image of what we are trying to detect, we need to define edge cases for what is and what is not a green strawberry. This is necessary for the first step of annotating the field images as we want our annotations to be consistent throughout. Inconsistent labels can confuse the classifier with poor data and reduce its accuracy. Figure 5 below shows a few examples of edge cases.

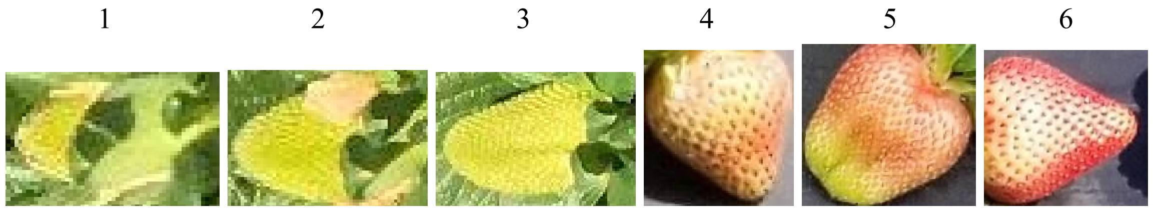


Fig. 5. Green Strawberry Edge Cases

As the green strawberry matures, it progressively becomes more red. Images 4 through 6 show berries in various shades of red. Originally, the field images were annotated with zero tolerance for red as it makes the annotation process much simpler. This can be problematic however as the image classier may see little difference between the berry in image 4 and the green strawberry defined earlier. This leads to more false positive guesses reducing the overall accuracy. Additionally, the red color first appears towards the stem and travels down the fruit, meaning some berries can appear both pink and green in color as shown in image 5. There are many images of green and red strawberries in our dataset but far fewer pink strawberries. Because of this, the classifier would make decisions on what is a green or red strawberry based on which category it most appears to belong to. This means a pink berry that is more green than red would be classified as green. Due to the nature of the dataset and how the classifier performed, the field data was annotated with a more lenient definition of a green strawberry. A green strawberry can contain red as long as it more clearly belongs to the green strawberry category rather than the red category.

Images 1 through 3 in Figure 5 above are the same as images 2 through 4 in Figure 4 from before but have been cropped precisely around the berry as a localization algorithm would. When the region of interest is precisely around the berry and the ROI is downscaled to the input image blob size for the classifier, information on the scale is lost. The difference in the growth cycle for images 2 through 4 were mostly related to scale so they all look very similar now. There are minimal differences between green strawberry snippets that are two weeks apart in growth, which is problematic as the original reason for detecting green berries was to estimate the number of red ones later on. If our definition of a green strawberry spanned across two weeks of its growth cycle, then less than half of the detected berries would become red ones in one weeks time. This would undermine the original reason for detecting green strawberries.

Our definition of a green strawberry takes the middle ground between a bud and a fully grown berry. If the proposed green strawberry is closer in size to a fully grown berry than a bud, then it is classified as such. This definition restricts the time a green strawberry lasts to about one week. This range is still very lenient but the leniency allows for more green strawberry snippets to be entered as positive training data. There is a trade off between the amount of training data and the specificity of what is a green strawberry. The definition has changed after evaluating the classifier accuracy to include more strawberry snippets as positive examples. If more training data was available, a more specific definition of what a green strawberry could be used.

A common question is why detect green strawberries rather than a different stage in the strawberry life cycle, like flowers? If we want to give the farmers as much time beforehand, wouldn't it be better to choose a strawberry stage closer to the beginning of its life cycle? Other strawberry stages like flowers prove to be more ineffective than green berries for one main reason. Farmers will prune strawberry flowers during early periods of their growth, so there is no certainty the flower will exist in the near future thus not all flowers turn into ripened strawberries. Green strawberries are not pruned, so they prove to be a more useful metric for crop yield prediction.

1.4 Project Objectives

The goal of this project is to build a model that will detect green strawberries given an input field image. The model will return the total number of green strawberries in each image and place bounding boxes on positive guesses. This will allow the user to confirm whether or not the guess was a true positive or false positive. A successful project will correctly locate and classify green strawberries in the image with an accuracy that is sufficient for approximating the number of red strawberries in the field one week later.

Chapter 2
LITERATURE REVIEW

2.1 Strawberry Detection Under Various Harvestation Stages (Yavisht Fitter, 2019)

This paper discusses and implements three different classifiers. Fitter's paper uses a histogram of oriented gradients (HOG), local binary patterns (LBP), and a convolutional neural network (CNN) to detect ripened strawberries. A dataset of 600 images was used as the basis of the training and test images. This paper is the starting point for inspiration for this thesis topic and the future works mentions the use of an R-CNN implementation.

The first step in detecting ripened strawberries is to determine the most useful features to extract. The RGB values are obviously useful as the ripe strawberries will be red in color. Some branches and leaves also contain traces of red pixelation, so this trait alone isn't sufficient. The RGB images can be converted into HSV to obtain additional features.

The image resolution affects the accuracy of the strawberry detection and the speed that the images are processed. Image sizes of 4000 by 5000 have a high accuracy but take significantly longer to process. Low res images like 1000 by 2000 aren't as accurate but are exponentially quicker to process.

Unripened strawberries can also be classified by using local binary patterns (LBP). The image is first converted to grayscale then the LBP model can be generated using Matlab. This model can be combined with a support vector machine (SVM) to create the classifier. A histogram of oriented gradients (HOG) can also be used to classify ripe and unripe strawberries. The number of orientations can be altered to change the performance of the HOG model.

A CNN was used to differentiate ripe and unripe strawberries. The neural network requires a large amount of data to perform well and the model was fed nearly 2000 images of ripe and unripe strawberries. The model for the CNN was taken from the VGG Net architecture as it provides a general structure that's applicable to many datasets. The VGG architecture was implemented in Keras and achieved the highest accuracy rates of the three implementations.

In Fitter's paper, the HOG implementation didn't work well, so the various parameters weren't examined thoroughly. The results were inconclusive, but with more work the HOG implementation could work well despite this paper stating otherwise.

On the other hand, the LBP solution was implemented smoothly and detected the red strawberries at a 74% accuracy rate. This high accuracy was achieved when using the large image size (5000x4000).

The CNN was the clear winner of the three implementations with an 88% accuracy for red strawberry detection. This high accuracy was also efficient using the 600x800 sized images with the sliding windows method. The high efficiency and accuracy made the CNN the best implementation. In this thesis paper, we will evaluate how this CNN compares to other DNNs and we will explore how the accuracy rate changes with various input image sizes.

2.2 A Deep Learning Method for Recognizing Elevated Mature Strawberries (X. Li et al., 2018)

This paper utilizes the HOG+SVM and CaffeNet CNN to count matured strawberries. SVM was the baseline implementation which correctly classified strawberries at 84% accuracy. The CNN correctly classified the strawberries at 95% after 650 iterations and achieved a 99.5% accuracy at around 20,000 iterations. This paper will be used as a starting point for fine tuning a DNN implementation to our strawberry application.

The strawberry line information can be gathered using a Hough transform. This information is used as the basis for the HOG gradient direction feature. SVM can then be applied to detect mature strawberries from unripened ones. The methodology above is heavily dependent on the RGB information from these images. This dependence negatively affects the classification rate for the implementation when factors like shade and weather are added to the equation.

The color and shape of ripened strawberries were the two most important features for the SVM classifier. A flowchart of how the HOG feature vector is formed is shown in Figure 6 to the right. The H tone component in the HIS (hue, saturation, intensity) color model was found to be the best trait. These traits were analyzed after feeding multiple cropped images of both ripe and unripened strawberries.

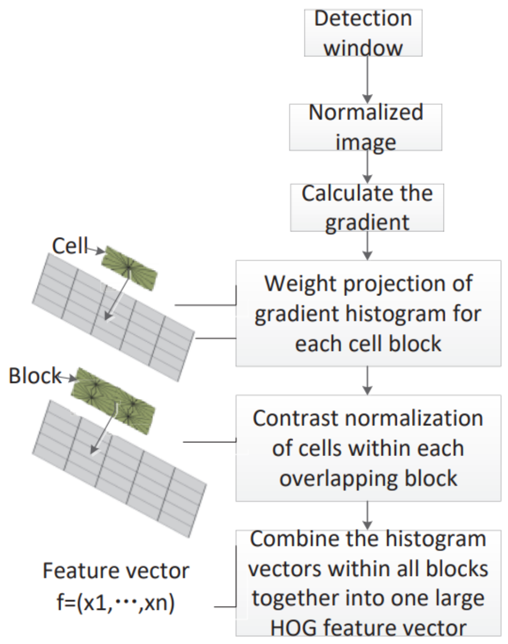

Detection window
Normalized image
Calculate the gradient
Cell
Weight projection of gradient histogram for each cell block
Block
Contrast normalization of cells within each overlapping block
Feature vector f=(x1,⋯,xn)
Combine the histogram vectors within all blocks together into one large HOG feature vector

Fig. 6. HOG Feature Extraction Process

In addition to SVM, a CNN called CaffeNet was used to classify the strawberries. CaffeNet is a general CNN structure that was optimized by AlexNet. The network contains eight weighted layers with five being convolutional and the other three being fully connected layers. The CNN has two class labels that are used to maximize the multi-category logistic regression. Random gradient descent was used to approach the minimum of the cost function.

The HOG+SVM implementation identified the ripened strawberries with an 84% accuracy. Discolored branches or reddish leaves prove to be misclassified as strawberries occasionally, which led to a number of false positives. The HOG implementation is heavily dependent on the red hue and contrast of the strawberries with the background. The SVM approach worked in a general case, but a CNN with a sufficient amount of data will prove to be much better.

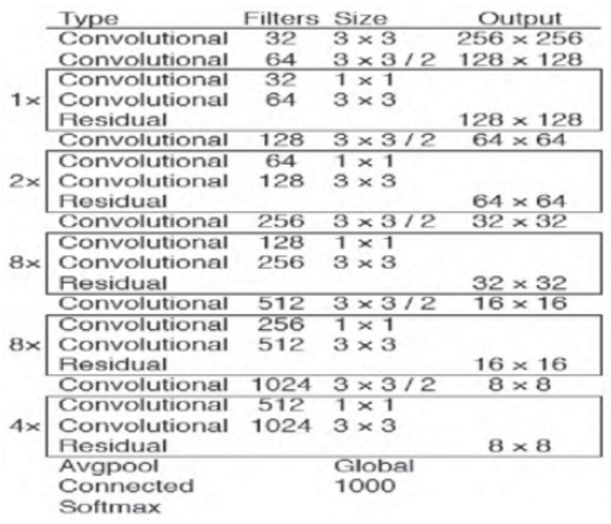
CaffeNet had a 95% accuracy after 650 iterations and a 99.5% accuracy after 20,000 iterations. The recognition time for 650 iterations was 44ms, which is plenty quick for most applications. This means the DNN can correctly classify the strawberries with 11% more accuracy with only a 44ms delay. Because of the quick response time and unmatched accuracy, CaffeNet was the better implementation of the two and worked well for red strawberry detection.

2.3 Target Detection of Banana String and Fruit Stalk Based on YOLOv3 Deep Learning Network (R. Zhang, X. Li, L. Zhu, M. Zhong and Y. Gao, 2021)

Harvesting bananas is a labor intensive process and can be dangerous. There are many inefficiencies in sending people to locate and pick bananas. To make this process safer and reduce the health risks that these workers experience, the YOLO (You Only Look Once) deep architecture was implemented to detect the fruit. This paper analyzes the YOLOv3 deep neural network and implements it using the Keras and Tensorflow framework. This paper is the reference used for analyzing the YOLOv3 architecture.

Detecting the banana stalk and banana string is key in locating and determining the yield for a given tree. A DNN was chosen over traditional methods as the stalk and string features are not trivial to extract and a neural network would likely find subtle relationships that are needed to detect these characteristics with a high accuracy rate.

For their application, the features needed to be extracted as quickly as possible with a high accuracy rate. YOLOv3 was chosen



	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Fig. 7. Structure of YOLOv3

as the DNN executes quickly and has the highest accuracy rate when compared to YOLOv1 and YOLOv2. YOLOv3 is Redmon's improved version of YOLOv2 with a Darknet53 structure instead of a Darknet19. The 53 convolutional layers (as shown in Figure 7) and the addition of Batch normalization, allows YOLOv3 to achieve high accuracy rates with fewer epochs. YOLOv3 continues to use k-means clustering like its previous iterations. The database only consists of 800 images but each has a high resolution of 2048x4096. The training to test ratio was 9:1.

Loss was largely unaffected after the 5th epoch. Data up to the 100th epoch was collected and the banana handle detection accuracy plateaued at 88% and string accuracy at 98%. The banana handle detection was the most difficult part and image scientists were happy with an 88% accuracy rate. YOLOv3 is much faster to train when compared to other DNNs like ResNet-50, yet has a high accuracy rate despite the smaller dataset of only 800 images. The performance of YOLOv3 for banana detection seems promising if applied to our strawberry detection application.

2.4 Faster R-CNN Implementation Method for Multi-Fruit Detection Using Tensorflow Platform (H. Basri, I. Syarif and S. Sukaridhoto, 2018)

Millions of tons of dragon fruit and mangos are harvested in Indonesia each year. Mangos take around 110 - 120 days to mature and due to shipping times and inefficiencies in farming, many fruits are thrown out. The faster R-CNN implementation can be applied to detect ripened fruits more quickly and accurately to improve harvest yields. In this work, the faster R-CNN architecture was analyzed to see if it's a good model for our application.

The R-CNN architecture was implemented on a MobileNet via the TensorFlow python library. The model first executes generic object detection to find bounding boxes (BBs) around all of the regions of interest (ROI). Once the BBs are found, each is classified as either a mango or pitaya. The structure of Faster R-CNN is shown in Figure 8 below. The dataset consists of 700 mangos and 700 pitayas. The image input size was variable but typical dimensions were around 300x300 pixels. Overall, the dataset is quite small, which could be problematic as the R-CNN model works best with large amounts of data.

The target accuracy of the Faster R-CNN model was 99% but the actual was much lower. The accuracy fluctuates between 70.6% and 64.4% depending on the number of mult-adds and the input image size as shown in table 1. Downscaling the images subsequently lowers the accuracy but increases performance. With no downsampling, the MobileNet modeled after the Faster R-CNN architecture received an accuracy of 70.6%. The lower result is most likely due to the smaller dataset on the multiple classes.

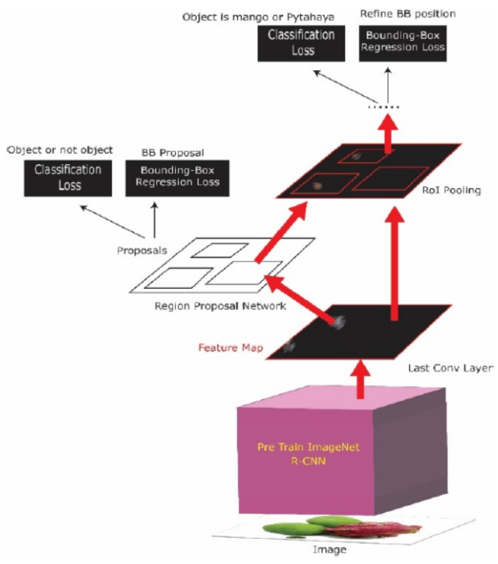
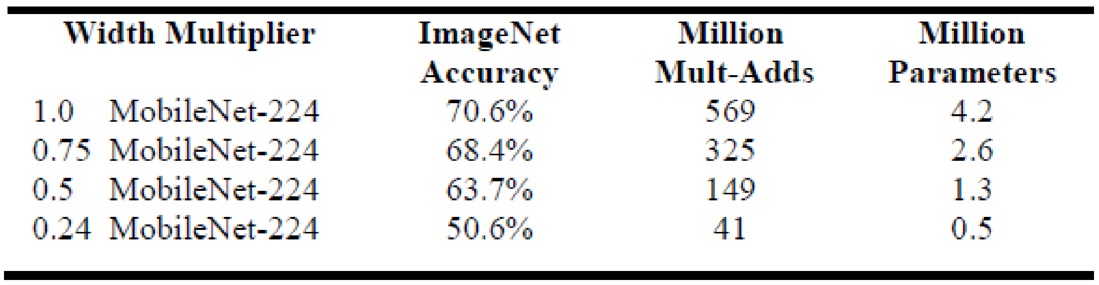


Fig. 8. Structure of Faster R-CNN

Table 1. MobileNet Width Multiplier



Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.24 MobileNet-224	50.6%	41	0.5

2.5 Analysis of Deep Learning Architectures for Object Detection - A Critical Review
(M. Pandiya, S. Dassani and P. Mangalraj, 2020)

This paper compares some of the most popular DNN architectures and evaluates the performance in regards to accuracy and the learning rate. These DNNs were tested using the CIFAR10 dataset along with the Simpsons dataset. CIFAR10 consists of 10 common images like airplanes and birds while the Simpsons dataset has images of 20 of the most popular characters. Although the tested application is different, the optimal learning rates and execution times of the architectures will be useful.

This paper starts by explaining the general structure of a DNN with the convolutional, pooling, activation and fully connected layers. The activation function in most DNNs today is ReLu and the last layer of the neural network is for binary classification of softmax. Four popular DNN architectures were evaluated including AlexNet, VGG, InceptionNet and ResNet.

AlexNet introduced max pooling and ReLu for the first time while GoogleLeNet introduced the inception block. ResNet created the residual block but this architecture wasn't evaluated fully in this paper.

Table 2. Evolution of CNN

Year	Network	Characteristic	Architectureal Complexity
1998	LeNet	First popular CNN architecture	5 layers with around 0.06M parameters
2012	**AlexNet**	Uses the ReLU activation function,droupout and overlap Pooling	8 layers with 60M parameters
2014	**VGG**	Homogeneous Topology and Small kernel size	138M parameters and 19 layers
2015	**Google LeNet(Inception v1)**	Introduced the concept of different kernel size in single block	22 layers , 4M parameters
2016	ResNet	Introduced Residual learning and Identity mapping based skip connection	1.7M and 110 layers when trained on CIFAR-10
2016	Inception-ResNet	Combines the split transform merge and the residual block	55.8M parameters and 572 layers
2017	DenseNet	Cross layer information flow in a multi-path network	For CIFAR-10,15.3M parameters with 250 layers

Three architectures, AlexNet, VGG, and InceptionNet were fully tested. The results in this paper are slightly counterintuitive by stating the AlexNet architecture is the best model for differentiating the various objects. ResNet was not tested due to hardware limitations and training times.

This paper concluded that fine tuning the CNN parameters is more valuable than the raw model. AlexNet performed the best, according to the authors, as the parameters were more finely tuned for the application at hand. Figure 9 to the right shows the accuracy

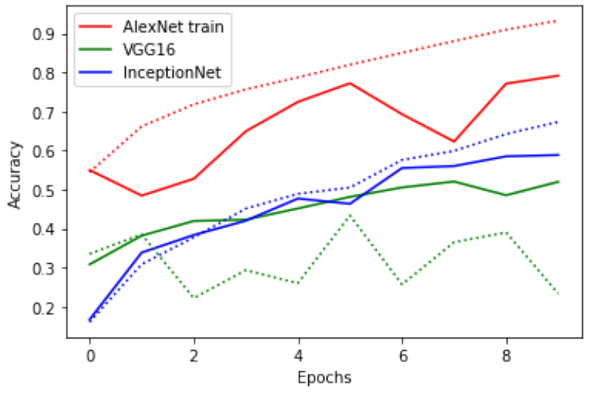


Fig. 9. DNN Test Results

of AlexNet outperforming VGG16 and InceptionNet. The main takeaway from this paper is to spend time tweaking network parameters before making conclusions on what model is best for the application.

2.6 Object Detection With Deep Learning: A Review (Z. Zhao, P. Zheng, S. Xu and X. Wu, 2019)

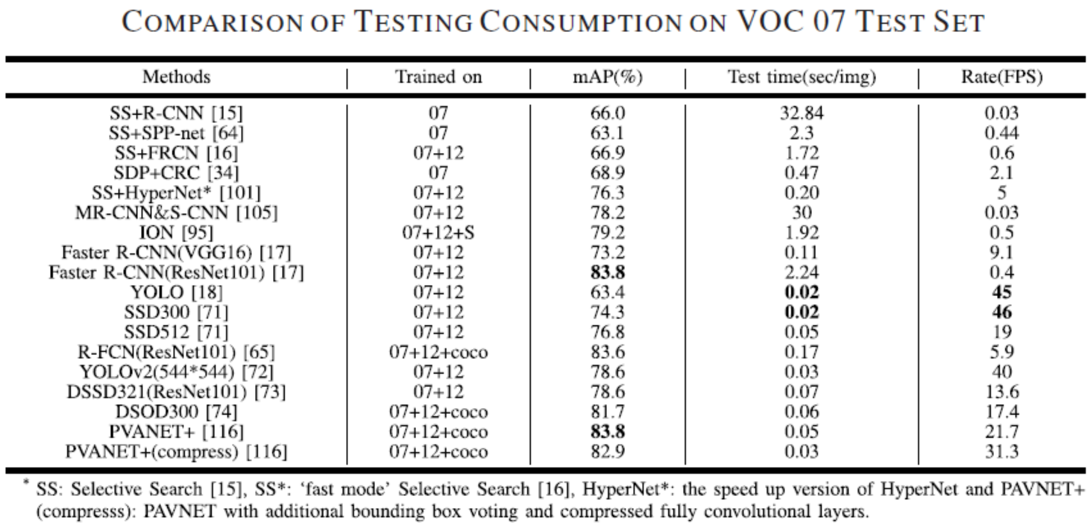
This paper compares some of the most popular DNNs and determines the best architecture for their specialized application. Object detection can be summed up into three main groups of generic, salient and facial recognition. The generic object detection section is the main focus as it's most pertinent to our project of fruit classification.

The first section of this paper is dedicated to the history of image processing and deep learning. The traditional methods of feature extraction and classification are described in a general flowchart with more focus on the state of the art deep learning workflow. The neural networks are evaluated based on the dataset and application.

Generic object detection is focused on placing bounding boxes on ROIs and classifying each object based on previous training. Some of the most popular DNNs for object detection that this paper describes include R-CNN, Fast R-CNN, Faster R-CNN, YOLO, YOLOv2 and YOLOv3.

ResNet has the highest overall accuracy rate while YOLO has the fastest test time. These results are intuitive and concur with results from previous papers. Table 3 below shows the results of some of the most popular DNNs. These results will be useful for choosing a model to be the basis for the strawberry detection.

Table 3. DNN Analysis



COMPARISON OF TESTING CONSUMPTION ON VOC 07 TEST SET

Methods	Trained on	mAP(%)	Test time(sec/img)	Rate(FPS)
SS+R-CNN [15]	07	66.0	32.84	0.03
SS+SPP-net [64]	07	63.1	2.3	0.44
SS+FRCN [16]	07+12	66.9	1.72	0.6
SDP+CRC [34]	07	68.9	0.47	2.1
SS+HyperNet* [101]	07+12	76.3	0.20	5
MR-CNN&S-CNN [105]	07+12	78.2	30	0.03
ION [95]	07+12+S	79.2	1.92	0.5
Faster R-CNN(VGG16) [17]	07+12	73.2	0.11	9.1
Faster R-CNN(ResNet101) [17]	07+12	**83.8**	2.24	0.4
YOLO [18]	07+12	63.4	**0.02**	**45**
SSD300 [71]	07+12	74.3	**0.02**	**46**
SSD512 [71]	07+12	76.8	0.05	19
R-FCN(ResNet101) [65]	07+12+coco	83.6	0.17	5.9
YOLOv2(544*544) [72]	07+12	78.6	0.03	40
DSSD321(ResNet101) [73]	07+12	78.6	0.07	13.6
DSOD300 [74]	07+12+coco	81.7	0.06	17.4
PVANET+ [116]	07+12+coco	**83.8**	0.05	21.7
PVANET+(compress) [116]	07+12+coco	82.9	0.03	31.3

* SS: Selective Search [15], SS*: 'fast mode' Selective Search [16], HyperNet*: the speed up version of HyperNet and PAVNET+ (compresss): PAVNET with additional bounding box voting and compressed fully convolutional layers.

2.7 Fine-tuned MobileNet Classifier for Classification of Strawberry and Cherry Fruit Types (Venkatesh, N. Y, S. U. Hegde and S. S, 2021)

This paper goes into depth on a MobileNet based deep learning architecture for detecting strawberries and cherries. The overall accuracy of the model is about 98.6% with 0.38% loss.

Strong rains and plant disease are prevalent in strawberry and cherry plants in India. Measuring these two factors, we can extrapolate the harvest yield if we can accurately count fruits early in the season. Spectroscopic imaging has been used in the past to detect the fruit, but it has a high cost for detecting diseases. Machine learning using modern cameras is more cost effective and a DNN with enough training data should be able to detect the strawberries and cherries at a high accuracy rate.

For the dataset, they gathered 4250 strawberry and 3878 cherry training images alongside 990 strawberry and 1012 cherry test images. With 10,000 images in total, a MobileNet architecture was made using the TensorFlow library. The model contains 88 layers and replaces some of the MobileNet layers with new functional layers. Some of the layers replaced include the new depthwise layer, pointwise layer, ReLu and Batch normalization layer, and the global average pooling layer with the fully connected layer removed.

Preprocessing techniques were used to eliminate the background noise to further increase the accuracy of detecting the fruit. The images were converted to HSV and the mean value was computed for the LAB model. Attributes of the fruit can be accentuated using equations for the homogeneity, correlation, energy, and contrast.

The modified MobileNet architecture is similar to the original but takes out the fully connected layer. The parameters for the stride and kernel size have also been modified to accommodate the 256x256 input images. Figure 10 shows the design changes for the MobileNet model.

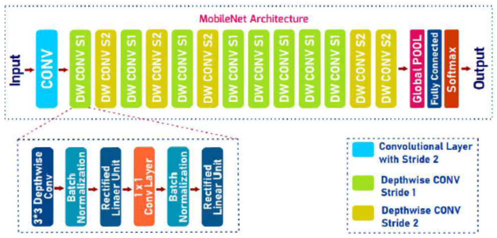


Fig. 3. MobileNet CNN Model

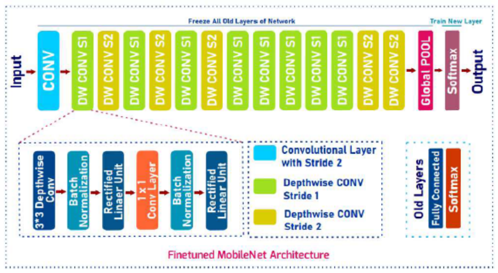


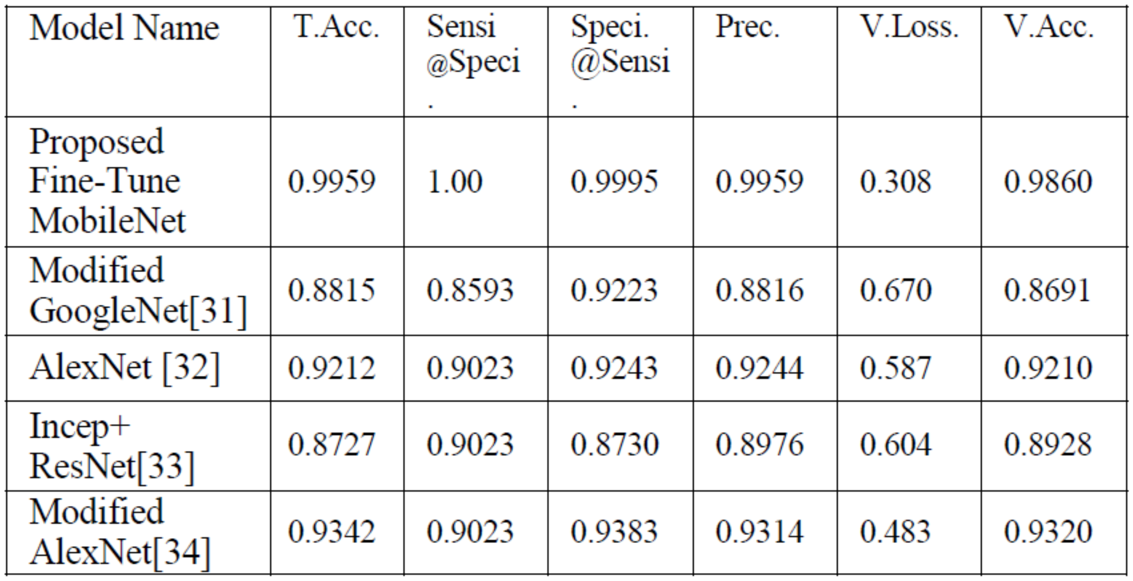
Fig. 4. Proposed MobileNet Fine-Tuned MobileNet

Fig. 10. Proposed MobileNet Model

The model was built using Python3.8, Anaconda3, OpenCV-Python, and the Keras library. The results were tested on an Intel i7 processor at 2.34 GHz with 2GB of RAM and an NVIDIA graphics card. The fine tuned mobile net took 36 minutes to train while GoogleNet took 223 minutes and VGGNet16 took 250 minutes.

Not only was the fine tuned MobileNet the fastest to train but it also had the highest accuracy rate at 98.6% with only 0.38% loss. All of the tested model results are shown in table 4 below.

Table 4. Comparison of Fine-Tuned MobileNet with other DNNs



Model Name	T.Acc.	Sensi @Speci.	Speci. @Sensi.	Prec.	V.Loss.	V.Acc.
Proposed Fine-Tune MobileNet	0.9959	1.00	0.9995	0.9959	0.308	0.9860
Modified GoogleNet[31]	0.8815	0.8593	0.9223	0.8816	0.670	0.8691
AlexNet [32]	0.9212	0.9023	0.9243	0.9244	0.587	0.9210
Incep+ ResNet[33]	0.8727	0.9023	0.8730	0.8976	0.604	0.8928
Modified AlexNet[34]	0.9342	0.9023	0.9383	0.9314	0.483	0.9320

The results in this paper show that a fine tuned MobileNet using the TensorFlow library is a good implementation for fruit detection. This IEEE publication will be referenced when building the architecture and additionally the ripened strawberries from its database will be merged with the data collected for Cal Poly's field in 2018.

2.8 Grape detection, segmentation, and tracking using deep neural networks and three-dimensional association (Thiago T. Santos, Leonardo L. de Souza, Andreza A. dos Santos, Sandra Avila, 2020)

This paper compares the YOLO model against the Mask R-CNN. Mask R-CNN is Faster R-CNN with the added layer mask component. This is used to accurately detect the grape clusters and to further extrapolate crop yield from the mask sizes. This paper is used as a reference for comparing the YOLO and Mask R-CNN models.

This paper presents four separate contributions to the field of deep learning. First, they propose a new method of creating image annotations using interactive image segmentation. This is used to generate image masks to separate the grape clusters from the background. These image masks can be compared with the results of the Mask R-CNN model.

Second, they created a new database of 110 images to use for both training and test. These images contain five different grape species with multiple grape clusters in each image. There are 1307 clusters in this dataset. This was combined with the Embrapa Wine Grape Instance Segmentation Dataset (WGISD), which contains 300 images with 4432 clusters in total.

After combining the datasets and properly annotating the images, they used them as the input of two different DNN models. This paper compared and contrasted YOLO versus Mask R-CNN in terms of object detection and image segmentation. The image annotation work gave more criteria to evaluate the Mask R-CNN architecture, but regardless the R-CNN algorithm proved to be the best for cluster detection.

Lastly, they developed a fruit counting methodology to determine the yield from the cluster masks. This work is more specific to the grape detection problem. For the strawberry detection algorithm, we will focus on the analysis of the YOLO and Mask R-CNN models and determine which will be best suited for the application.

Training was done using 80% of the database images with 20% used for test. The input image blobs were scaled at 1024x1024x3 with 110 images used in total. The results of Mask R-CNN, YOLOv2 and YOLOv3 are shown in table 5 below.

Table 5. Object Detection Results for Mask R-CNN vs YOLO

	Mask R-CNN				YOLOv2				YOLOv3			
IoU	AP	P_{box}	R_{box}	F_1	AP	P_{box}	R_{box}	F_1	AP	P_{box}	R_{box}	F_1
0.300	0.805	0.907	0.873	0.890	0.675	0.893	0.728	0.802	0.566	0.901	0.597	0.718
0.400	0.777	0.891	0.858	0.874	0.585	0.818	0.667	0.735	0.494	0.829	0.550	0.661
0.500	0.719	0.856	0.824	0.840	0.478	0.726	0.591	0.652	0.394	0.726	0.481	0.579
0.600	0.611	0.788	0.759	0.773	0.288	0.559	0.455	0.502	0.261	0.587	0.389	0.468
0.700	0.487	0.697	0.671	0.684	0.139	0.390	0.318	0.350	0.125	0.405	0.269	0.323
0.800	0.276	0.521	0.502	0.511	0.027	0.172	0.140	0.154	0.036	0.205	0.136	0.164

YOLOv2 and YOLOv3 differ based on the number of convolutional layers in the Darknet CNN. Additionally, YOLOv2 uses softmax while YOLOv3 uses multi-label classification. These changes increased the accuracy of YOLOv3 but neither of these YOLO models are quite as good as the MAsk R-CNN implementation. The Mask R-CNN model was the clear winner for grape cluster detection.

This paper also mentioned that the Mask R-CNN implementation has proven to work well for many other fruit types, so this model may work well for our strawberry application.

2.9 Segmentation as Selective Search for Object Recognition (K. E. A. van de Sande, J. R. R. Uijlings, T. Gevers and A. W. M. Smeulders, 2011)

The selective segmentation algorithm aims to return a list of bounding boxes where the object of interest could be located. The goal of selective search is to return fewer bounding boxes than other localization algorithms like sliding windows, while maximizing the likelihood an object is in one of the boxes.

In 2011, state of the art methods for object localization involved an exhaustive search over the image. This search strategy could find the object at a high accuracy rate but would return a needlessly large number of bounding boxes. The large number of BBs would decrease performance as each bounding box needs to be inputted into the classifier to determine if it is the object or not. The selective segmentation algorithm aims to use multiple invariant color spaces to combine similar regions, thus reducing the number BBs.

The proposed algorithm also works with different scales as shown in Figure 11 below. At larger scales, the algorithm returns exponentially fewer boxes for a more focused search. For smaller object detection, the algorithm is sensitive enough to detect subtle differences in hue and saturation. This algorithm will work well in our application for locating the green strawberries amongst the mostly green plants.

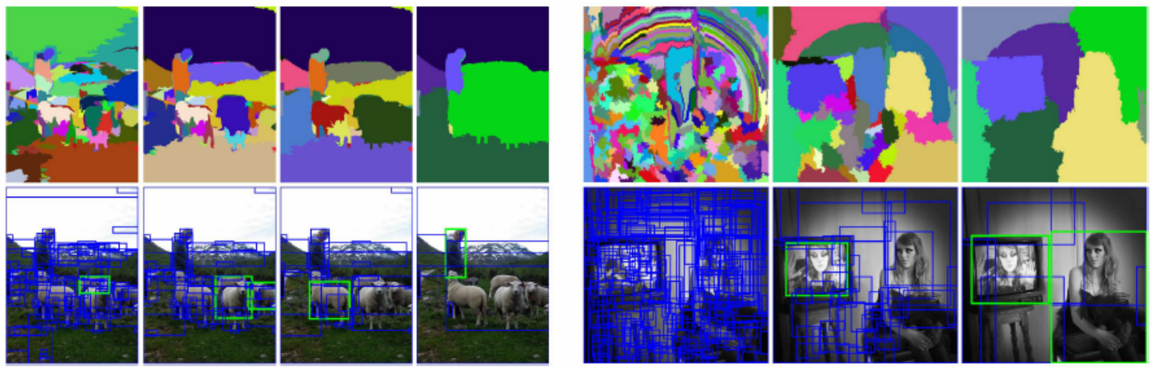
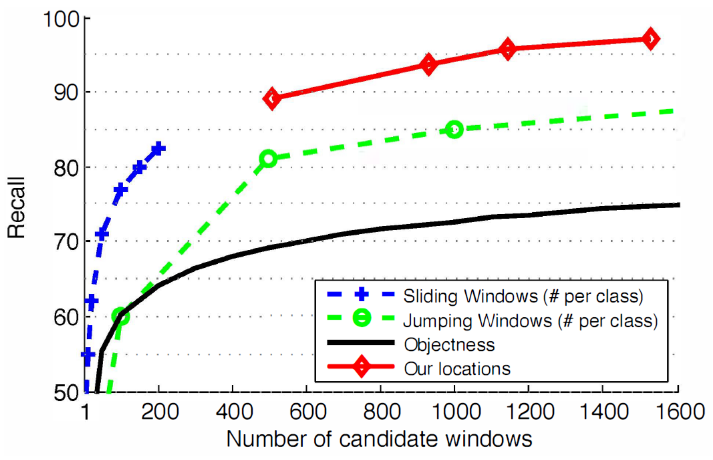


Fig. 11. Hierarchical Grouping Algorithm Examples

Selective segmentation was compared against sliding windows, jumping windows and other state of the art localization algorithms like objectness. The main metric used to evaluate the algorithms was the recall rate which is defined as the number of true positives over the number of true positives plus false negatives. Figure 12 below shows the recall rate for each tested algorithm.

The selective segmentation algorithm has a recall rate of 96.7% while the next leading algorithm, jumping windows, had a recall rate of 94%. The number of proposed bounding boxes is also significantly lower for the selective segmentation algorithm. The small number of bounding boxes will be useful later on when the classifier is added to help lower the number of false positives and expedite the testing time.



	Max. recall (%)	# windows
Sliding Windows [13]	83.0	200 per class
Jumping Windows [27]	94.0	10,000 per class
'Objectness' [1]	82.4	10,000
Our hypotheses	96.7	1,536

Fig. 12. Max Recall for Each Tested Algorithm

Chapter 3
TECHNICAL BACKGROUND

Object detection is a combination of both localization and classification. In order to successfully detect green strawberries, a localization algorithm must be applied on the field images to find the regions of interest (ROIs), then these regions are inputted into the classifier to make a binary decision on whether or not the input blob is a green strawberry. This chapter goes into depth on the localization and classification methods proposed in the prior research.

3.1 Convolutional Neural Network Overview

Convolutional neural networks (CNNs) were first introduced in the 1980s. Yann LeCun is credited with the development of the first neural network LeNet. The CNN was designed for handwritten digits but the applications of a neural network have expanded from then [21]. Deep neural networks (DNNs) are neural networks with a large number of convolutional layers, which are made possible with newer hardware.

DNNs are more relevant today than in the 80s due to advances in GPUs and accelerators. Modern day computers have enough processing power and RAM to support DNNs with many convolutional layers and parameters. These large neural networks contain millions of trainable parameters and are built around the core function of multiply and accumulate [21].

3.1.1 Neural Network Architecture

A neural network contains multiple convolutional layers in order to take an input blob image and dissect it down into a single likelihood value. The layers of a neural network can be broken down into three main categories of convolutions, pooling, and the fully-connected layer. One of the first networks designed was LeNet and will be used as an example for learning the different components of a CNN. Figure 13 below shows an example of LeNet-5. The smaller network contains 60k parameters.

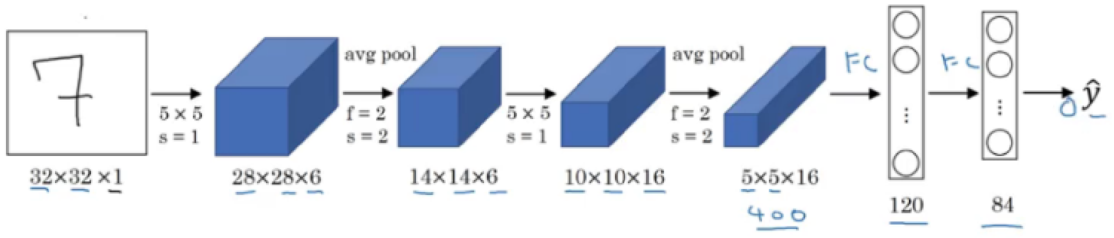


Fig. 13. LeNet-5 Architecture

The neural network above contains three convolutional layers, with two average pooling and two fully connected layers. The convolutional layers act as filters with 6 feature maps each. The average pooling layer reduces the number of parameters and further reduces the layer size. Finally the fully connected layers reduce the parameter down to 10 options using softmax. Each of the ten values correspond to a written digit that LeNet-5 was designed to detect [21].

The depth of a neural network refers to the number of layers, where architectures with a high layer count are considered deep neural networks (DNNs). These DNNs are often accompanied by a localization algorithm to propose image regions for the classifier. Three of the main deep architecture packages are R-CNN, YOLO and SSD.

3.1.2 R-CNN Overview

The region based convolutional neural network (R-CNN) was designed by Ross Girshick in order to reduce the number of proposed regions for the network [22]. Previous localization algorithms would return tens of thousands of bounding boxes, where each region could possibly contain the target object. The large number of proposed boxes made previous architectures largely inefficient as each region took time being inputted into the classifier. R-CNN was developed to reduce the number of regions [26].

R-CNN revolves around its localization algorithm, selective search. The algorithm segments the images into many subsections to generate a large number of candidate regions. Then a greedy algorithm merges similar regions together to form larger bounding boxes. These larger regions are then used to choose the final bounding boxes returned by the algorithm. The returned regions are much fewer in count when compared to other algorithms like sliding or jumping windows [12]. Section 3.3 describes the algorithm in more detail.

3.1.3 YOLO Overview

YOLO stands for 'You Only Look Once' and is able to detect objects in real-time with its fast executing algorithm. The localization algorithm utilizes residual blocks and bounding box regression to execute quickly.

Residual blocks are used to divide the image into smaller square regions. Each region is then given a binary classification on whether or not an object exists within the box. These boxes are then conglomerated using bounding box regression. Bounding boxes are categorized using their width, height, class and center point. Figure 14 below shows an example of a proposed region where Pc is the probability from the classifier and C is the class name [23].

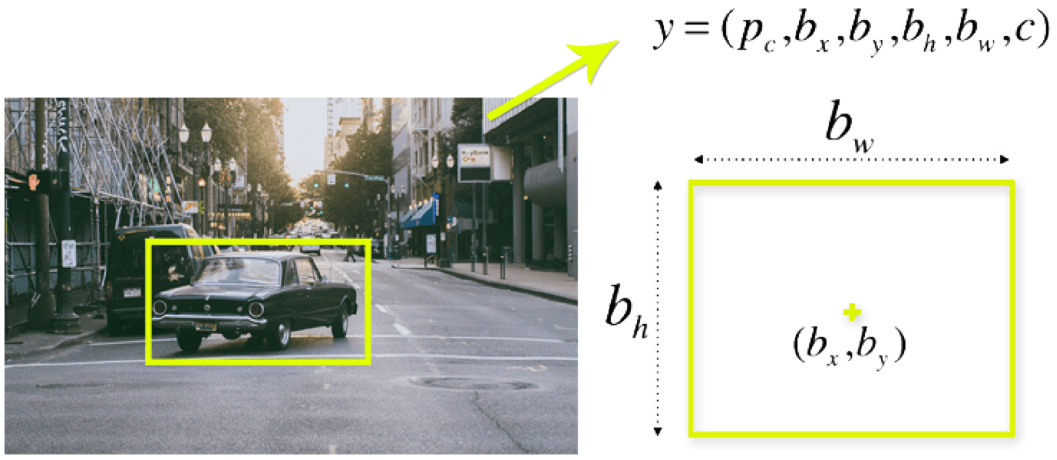


Fig. 14. Proposed Region from YOLO

YOLO is optimized to minimize execution time while still maintaining high accuracies for object detection. Because of this, YOLO is chosen for many real-time computer vision tasks.

3.1.4 SSD Overview

Single Shot Detection (SSD) was introduced by C. Szegedy et al. in 2016 as a means for even faster object detection [24]. As the name suggests, the single shot algorithm locates and classifies regions in a single forward pass [25]. The algorithm makes use of bounding box regression, like YOLO, by implementing MultiBox. MultiBox was created by Szegedy to propose bounding boxes regardless of any class it may belong to. SDD built off of this algorithm and added additional work to the prior probabilities to increase the overall mAP [25]. Figure 15 below shows the architecture for MultiBox.

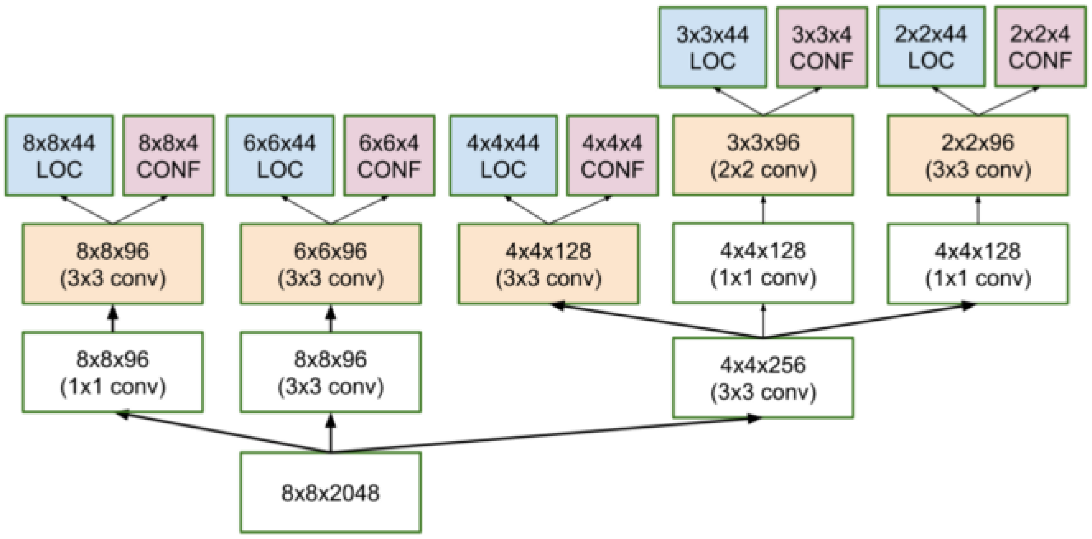


Fig. 15. MultiBox Architecture

The SSD architecture uses the VGG16 neural network to classify objects. When tested on the Pascal VOC2007 dataset, SSD-500 received the highest mAP of 76.8%. SSD is great for quickly classifying objects in an image but struggles for small object detection. The green strawberry dataset contains many small berries which may make SSD perform more poorly. A more thorough localization algorithm may be required to detect the smaller strawberries.

3.2 DNN Research Analysis

Three of the most popular deep neural networks are R-CNN, YOLO and SSD. R-CNN was analyzed more thoroughly than YOLO and SSD due to the selective segmentation algorithm. Selective search was able to detect the smaller, more subtle green strawberries in the image where YOLO and SSD performed more poorly. You Only Look Once (YOLO) and Single Shot Detection (SSD) are two algorithms that find the most prominent objects more quickly than selective search (SS), but SS had a much higher recall rate that was able to find green strawberries obscured by leaves or shadows. Because of the selective search algorithm, R-CNN was analyzed in depth while YOLO and SSD were not. Various neural networks like VGG16, MobileNet and ResNet50 were injected into the R-CNN architecture to see which would perform the strawberry classification the best.

Some of the main characteristics to look at when choosing a neural network is the mean accuracy percentage (mAP), the test time and the dataset it was tested on [9]. Our object detection algorithm was designed with accuracy as the most important metric. Accuracy was prioritized for two main reasons. The first is that selective search already takes a significant amount of time to perform for each inputted field image. The exact metrics will be analyzed in the classification results section in 4.3, but the additional time added from classifying each bounding box is much less significant. The second is the final crop yield prediction will only be as useful as the ability to accurately count the green berries in each image. The crop yield prediction will be exponentially more difficult if our classification accuracy is low. Table 6 below shows DNN metrics taken from prior research papers on multi-class object detection [9].

Table 6. Final DNN Analysis

DNN	Dataset	mAP(%)	Test time(sec/img)
SS+R-CNN	07	66.0	32.84
SS+HyperNet	07+12	76.3	0.20
R-FCN(ResNet101)	07+12+coco	83.6	0.17
YOLO	07+12	63.4	**0.02**
YOLOv2	07+12	78.6	0.03
SSD300	07+12	74.3	**0.02**
SSD512	07+12	76.8	0.05
Faster R-CNN(VGG16)	07+12	73.2	0.11
Faster R-CNN(ResNet101)	07+12	**83.8**	2.24

The DNN analysis above was tested on the 07+12 dataset, which involves the PASCAL visual object class challenge for years 2007 and 2012 [9]. Each PASCAL dataset contains 20 unique objects with approximately 10,000 images that were used for the competition which is similar but smaller than imagenet. The coco dataset includes 164,000 images with more than 100 common objects ranging from cars to dogs. The large datasets used in table 5 favor the networks with more convolutional layers, but a larger CNN may have negligible accuracy increase for smaller datasets. An example of this is the accuracy of VGG16 and VGG19 when trained on imagenet. The accuracy of the two models is nearly identical with the top-5 accuracy of VGG19 being 0.1% less accurate than VGG16 [13]. The difference is negligible and adding the additional trainable parameters could potentially lead to overfitting problems.

If we go by the highest mAP value, then Faster R-CNN using ResNet 101 would be the best choice as shown in table 6 above. Even though this DNN has the best results on paper, it may not be the best choice for our implementation as Faster R-CNN may have trouble detecting green strawberry features and ResNet101 has 101 convolutional layers with 44 million trainable parameters which may be excessive for our smaller dataset [13]. The large neural network provides high accuracy but is heavily data dependent. Our dataset is relatively small at 447 images so the larger neural network may be largely unnecessary and it would be better to choose a smaller DNN for boost performance. Additionally, a needlessly large neural network could lead to overfitting problems and undermine the effectiveness of the classifier.

It should be noted that of the CNNs mentioned in table 5, MobileNet is not shown. This is because the results of the fine-tuned MobileNet as described in [10] seem too good to be true with a mAP of 99.9. Combining these results with the data from “Object Detection With Deep Learning: A Review” [9] would inaccurately show MobileNet to perform significantly better than other CNNs. In order to gain a more accurate depiction of MobileNet, we will evaluate the classifier's accuracy for both MobileNet and MobileNetV2 and compare them against other CNNs like VGG16. This will also be a great comparison as the dataset and various parameters will be consistent when evaluating each model.

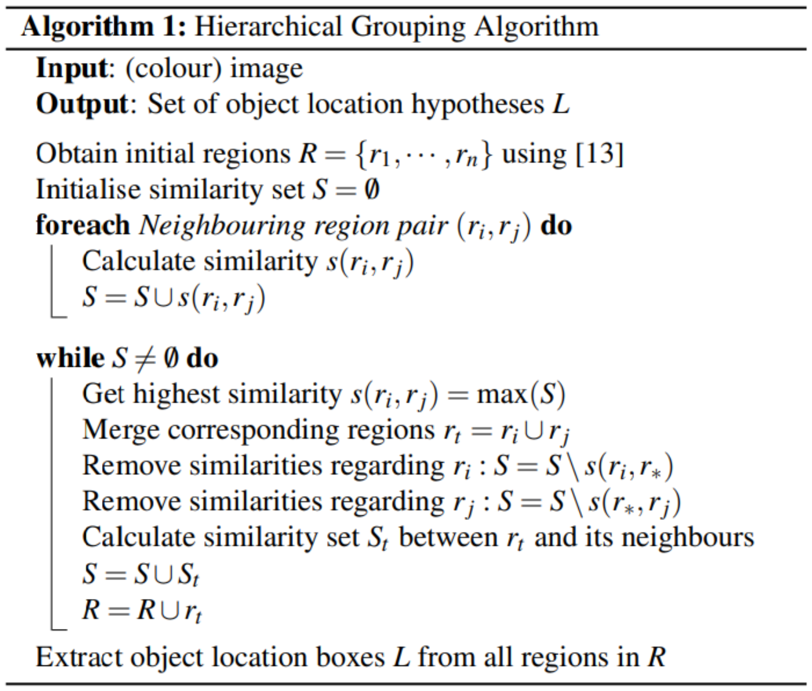
3.3 The Localization Algorithm

R-CNN uses the selective segmentation (SS) algorithm to return a list of bounding boxes with high likelihood values. The results of A. van de Sande's paper show how SS performs better than sliding windows, jumping windows and other state of the art localization algorithms [12]. These results state that SS returns fewer ROIs with a higher recall rate. Figure 12 from the literature review shows how SS outperforms sliding windows and jumping windows in terms of recall rate [12]. Sliding windows does return fewer bounding boxes per class and this localization algorithm was chosen by Fitter for his red strawberry detection, so we will test both SS and sliding windows to see which will be best for the green berry detection [4].

3.3.1 Selective Search (SS)

Selective search aims to maximize recall by eliminating bounding boxes with a low likelihood of an object being in it. The strategy is inspired by an exhaustive search but uses various techniques to reduce the time spent searching. To achieve this, the image is split into a rectangular grid with a fixed aspect ratio. Then several sampling methods are used to merge neighboring regions that are similar in various color regions. The algorithm returns ROIs invariant to scale or linear transformations. Many diverse color spaces are used to help the algorithm find every possible bounding box [25].

The algorithm starts by quickly proposing thousands of bounding boxes as a basis for selective search. Then neighboring regions are analyzed for similarity using many various color spaces including but not limited to RGB, Lab and HSV. The highest similarity is stored and the most similar regions are merged. The new region is added to the list and the two old boxes are removed. Figure 16 below showcases the algorithm's general method [25].



```
Algorithm 1: Hierarchical Grouping Algorithm
  Input: (colour) image
  Output: Set of object location hypotheses $L$

  Obtain initial regions $R = \{r_1, \cdots, r_n\}$ using [13]
  Initialise similarity set $S = \emptyset$
  foreach Neighbouring region pair $(r_i, r_j)$ do
      Calculate similarity $s(r_i, r_j)$
      $S = S \cup s(r_i, r_j)$

  while $S \neq \emptyset$ do
      Get highest similarity $s(r_i, r_j) = \max(S)$
      Merge corresponding regions $r_t = r_i \cup r_j$
      Remove similarities regarding $r_i : S = S \setminus s(r_i, r_*)$
      Remove similarities regarding $r_j : S = S \setminus s(r_*, r_j)$
      Calculate similarity set $S_t$ between $r_t$ and its neighbours
      $S = S \cup S_t$
      $R = R \cup r_t$

  Extract object location boxes $L$ from all regions in $R$
```

Fig. 16. Selective Search Grouping Algorithm

Many algorithms utilized before SS applied an exhaustive search across the image which can return 100,000 or more ROIs with weak recall. The large number of bounding boxes increases the execution time and can subsequently increase the number of false positives. SS aims to return 1,000-10,000 strong regions with high recall [12].

Van de Sande's paper, "Segmentation as Selective Search for Object Recognition" is the basis for OpenCV's implementation of the algorithm and its strategies. Selective segmentation can be fine tuned using five various strategies. Strategies can be applied to look for similarities in color, fill, multiple, size or texture [14]. These strategies will be tested amongst each other to see which will detect the subtleties of the green strawberries the best. SS also has two modes for a fast or quality search. The fast search returns more bounding boxes with a lower recall rate but does so much more quickly than the quality search which prioritizes high recall. These modes will also be analyzed.

3.3.2 Sliding Windows (SW)

The sliding windows algorithm takes a window of a set width and height and returns each region as it crosses over the image. The algorithm returns regions across the entire image of the same size. SW can be applied multiple times to an image with a number of different window sizes in order to find objects of varying scales. The algorithm executes quickly but proposes a large number of bounding boxes with an overall poor recall rate [12].

In order to evaluate both of these localization algorithms, the 447 field images must be annotated with bounding boxes around each green strawberry. Then from the annotated images, we can analyze the recall rate of each algorithm and decide which is best.

3.4 Image Annotations

There are a number of programs available for annotating images. The one used for this project is called MakeSense. It's a free, online application where the user can draw bounding boxes on each imported image using its intuitive GUI. Figure 17 below shows an image with 7 annotated green strawberries.

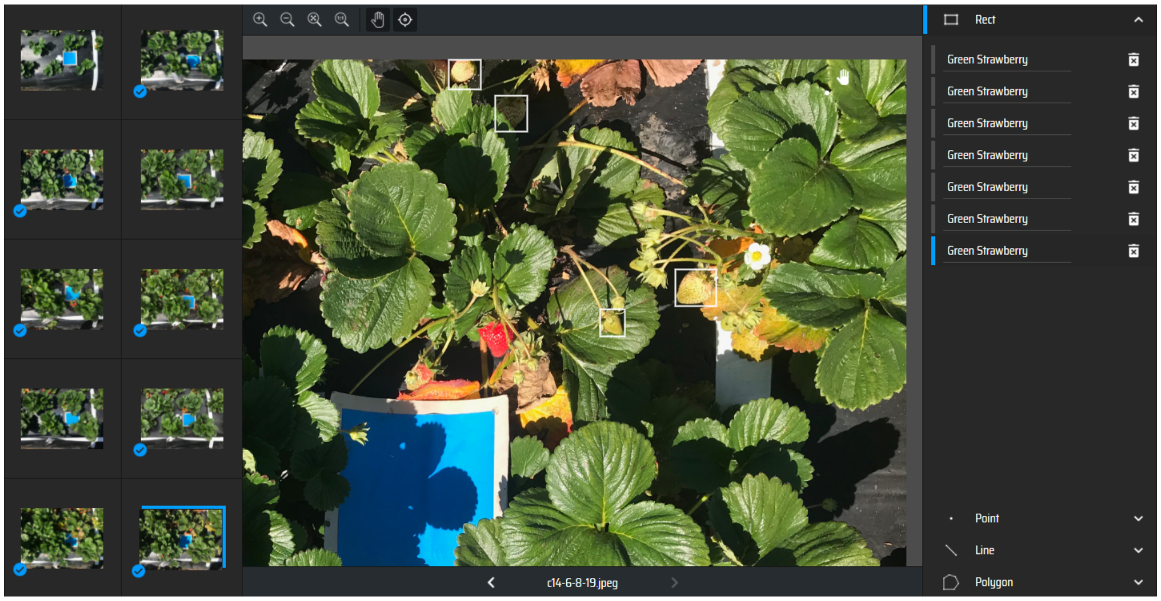


Fig. 17. Annotating Field Images using MakeSense

After annotating all 447 field images, the bounding boxes, labels and picture names could be exported to a CSV file. The generated CSV file is then reformatted to be used by the pandas python library [15]. The pandas library contains useful helper functions in using the bounding box information. Some methods include listing the total number of green berries per image and iterating through each bounding box.

The full dataset includes 1286 green strawberry annotations from 2019 in addition to the 1286 green snippets from 2018. The number of snippets being the same for both years is purely a coincidence. Unfortunately, 2572 green snippets is quite a small number of images to be used to train a CNN. These positive examples can be increased using data augmentation.

3.5 Expanding the Dataset

Data augmentation and negative hard mining were used to increase the amount of images used to train the classifier. Data augmentation was necessary for the R-CNN implementation because of the large DNN size. “Due to a large number of trainable parameters, in order to obtain multilevel robust features, data augmentation is very important for deep learning-based models (Faster R-CNN with “07,” “07 + 12,” and “07 + 12 + coco”) [9].” The datasets VOC 2007, VOC 2012, and coco already have more base images than our custom dataset, so adding augmentations is critical.

3.5.1 Data Augmentation

CNNs require a high volume of data to be effective which can be problematic with databases containing a small number of images. Data augmentation is used to increase the number of training and test images by applying translations, rotations and other image modifications. Data augmentation can turn one positive image into 45 or more. If done well, the augmented data will make the classifier more robust by providing much more data to train the parameters.

Data augmentation can be achieved using the CV2 library. The image processing module contains many helpful methods for affine transforms to both scale and translate images [17]. Notably, the cv2.warpAffine method allows for both rotations and translations to occur given a translation matrix T. Additionally, various image brightnesses were used to increase the data. This was done by increasing the gamma. With these methods, a ratio of one input snippet to 48 augmented images could be achieved using 36 rotations (10 degrees), 8 translations and 4 separate brightnesses. This ratio of 1:48 was the highest amount of data augmentation used and ultimately chosen to analyze each DNN.

3.5.2 Negative Hard Mining

Negative hard mining involves taking false positive clippings and inputting them into the training data. These negative examples help train the classifier on what is not a green strawberry to further reduce false positives later on. Negative hard mining is also useful for increasing the raw number of images used in our dataset. Overfitting can occur if too many false positive examples are used as training data. With the use of data augmentation and negative hard mining, we now have a larger dataset, with many positive and negative examples, that is ready for training.

3.6 Image Classification

In order to classify an image as a green strawberry or not, we need to successfully train a DNN model. In section 3.1 we analyzed various DNNs like VGG, MobileNet, and Resnet at a high level looking at the mAP%, execution time and the dataset it was tested on. In the following sections, we will give an overview of each DNN and explain some of their applications.

3.6.1 VGG

VGG is short for Visual Geometry Group, which is the name of the group of researchers at the University of Oxford that created the DNN. The architecture was first implemented for ImageNet back in 2014 and has been utilized ever since as a top neural network for image processing [18]. VGG16 is the standard size for the model with 16 convolutional layers. The model contains 138 million parameters for the standard 244x244 input blob size. The architecture uses max pooling along with ReLU as the activation function. Figure 18 below gives an overview of the model.

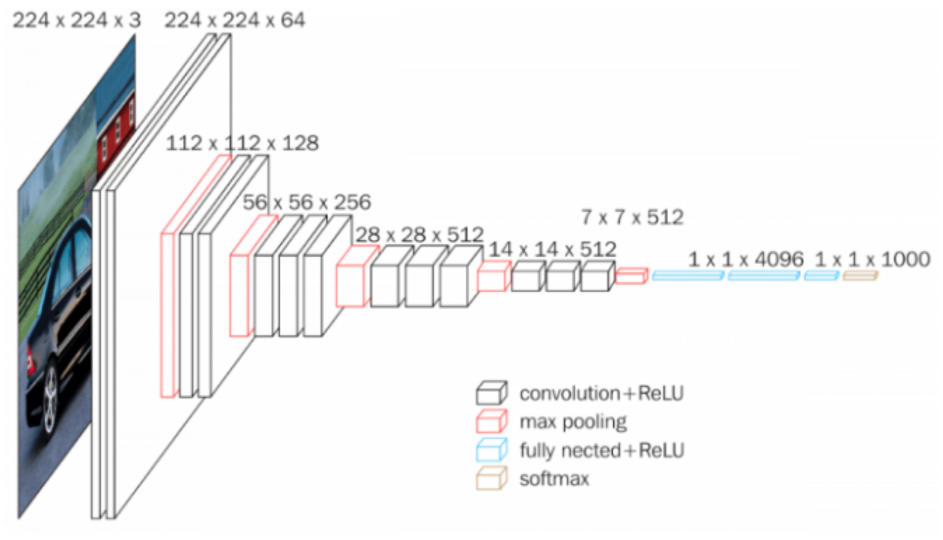


Fig. 18. VGG16 Architecture

VGG16 is a top DNN choice for many image processing applications. The large parameter count will help detect subtle features in the image. The model can also be adapted to take a 128x128 input blob. This would be helpful as the size of many green strawberry snippets range anywhere from 50 to 150 pixels wide. The lower size would allow for more images to be trained using the same amount of RAM and would subsequently reduce the trainable parameter count of the model.

3.6.2 MobileNet

MobileNet is a lightweight neural network developed by Google. As the name suggests, the model was designed for mobile applications. The architecture contains 4.2 million parameters across 28 convolutional layers. The model was designed for efficiency to allow computers with lower processing power to run the DNN. MobileNet is certainly the most efficient neural network for its smaller size.

Like VGG16, MobileNet uses average pooling and ReLU as its activation function. Despite MobileNet having nearly twice the number of convolutional layers as VGG16, the trainable parameter count is much lower, in part due to the depth-wise separable convolution [19]. This involves two layers, the depth-wise convolution used for filtering the input channels and the point-wise convolution used to combine the channels into a new feature. A standard convolutional layer contains 9 times more multiplications than a depth-wise separable layer [19]. Figure 19 below shows how MobileNet compares to other models like VGG16.

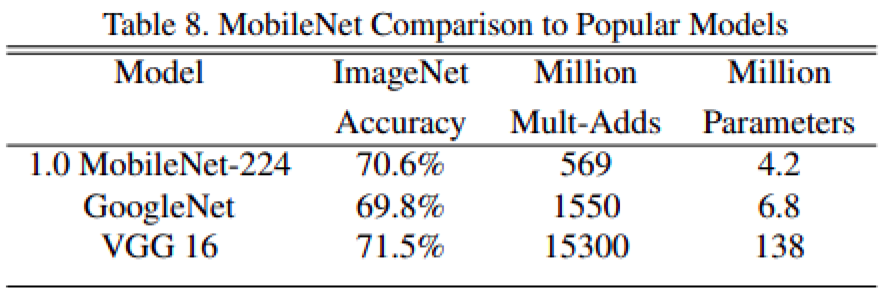


Table 8. MobileNet Comparison to Popular Models

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

Fig. 19. MobileNet vs Other Models

After training each model in Figure 18 on ImageNet, MobileNet obtained similar accuracies to other top models like VGG16 while using far fewer mult-adds and parameters. The ability for the model to obtain high accuracies while using far fewer parameters is impressive.

In section 3.1, we stated how mAP% is the top metric for our application in choosing which DNN to use. VGG16 did achieve a slightly higher accuracy than MobileNet but did so at a large expense to efficiency and overall size. For our project, we will make sure to implement both to see if the larger parameter count in VGG16 is necessary. If not, it will be much more efficient to implement MobileNet and additionally would give us more flexibility if we needed to house our solution in a portable form.

3.6.3 ResNet

ResNet is short for residual network and was first introduced in 2015. The DNN was proposed by Kaiming He and won the ImageNet competition that same year. ResNet was so impactful as it made it possible for a DNN to have hundreds of convolutional layers while maintaining high accuracy rates [20]. One of the main concepts ResNet introduces is the identity shortcut, which allows the DNN to stack additional layers without increasing the training error. A residual block contains two layers combined with this identity matrix. Figure 20 below shows a block diagram of the residual block.

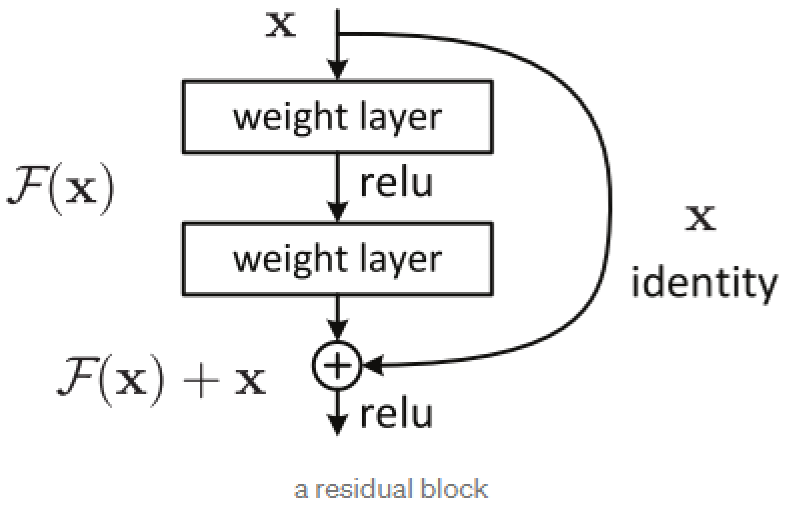

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
a residual block

Fig. 20. Residual Block

ResNet is the deepest neural network of the three we've looked at with typical implementations involving 50, 101, or 152 convolutional layers. ResNet50 contains 23 million parameters, ResNet101 contains more than 44 million and ResNet152 contains approximately 60 million parameters. Despite the high layer count, the number of trainable parameters is much smaller than some other DNNs like VGG16 at 138 million. It will be interesting to analyze the tradeoff between layer depth and parameter count. Our results from implementing the DNNs is shown in chapter 4 below.

Chapter 4
IMPLEMENTATION & RESULTS

This section will elaborate on the design choices for the object classifier. We will start by explaining how the dataset was compiled, then explain how the localization algorithm and classifier were implemented. After explaining the implementation, the results will be analyzed in regards to validation accuracy. Then from the detected green strawberries, we will analyze the correlation between our image results and the actual strawberry count to try to predict the crop yield.

4.1 Data Compilation

Images were annotated using the MakeSense program. The application has an intuitive GUI and is completely free as an online application. Figure 21 below shows an example of annotating an image from cluster 12.

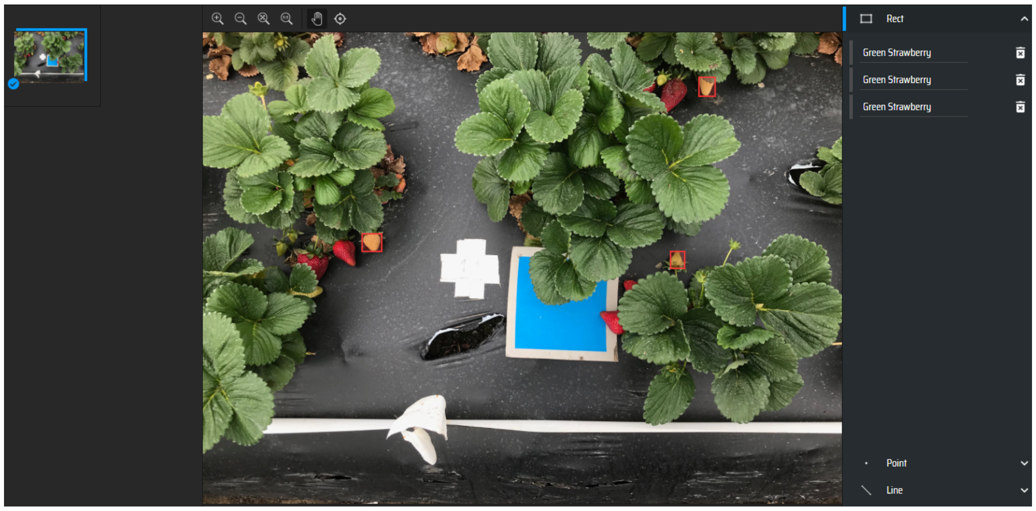
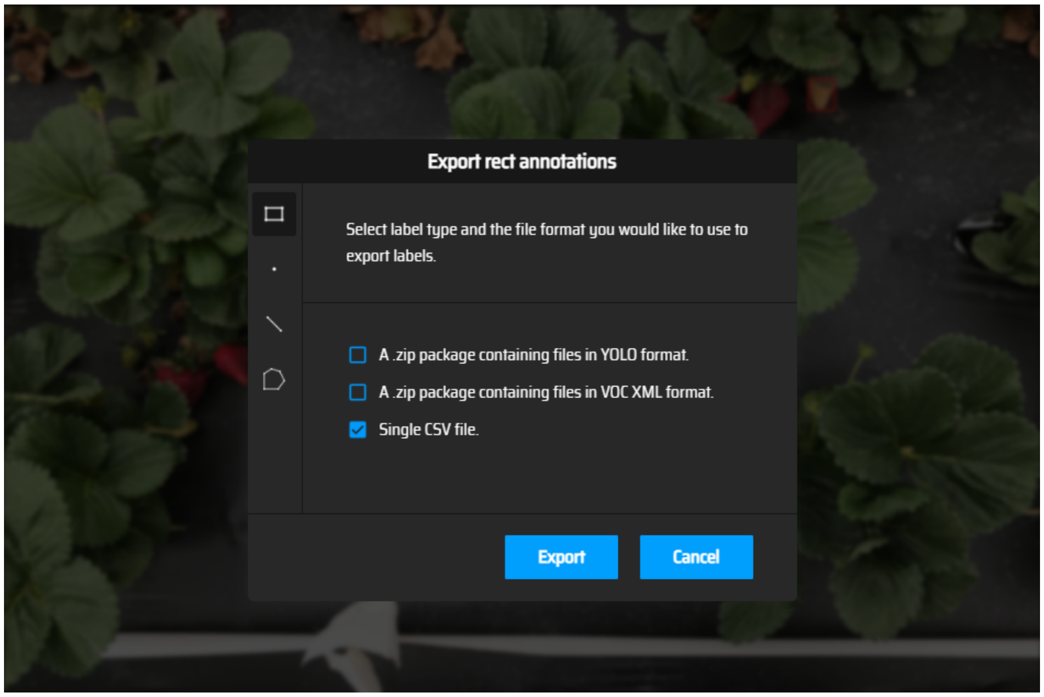


Fig. 21. Annotating an Image using MakeSense Program

The left column in Figure 21 shows each image ready for annotating, while the right column shows each bounding box label in the image. As we are doing single class classification of green strawberries, we only need to differentiate green strawberries from background elements. Once all of the images are annotated, we can easily export the annotations into one CSV file. Figure 22 below shows the export options MakeSense provides.


Export rect annotations

Select label type and the file format you would like to use to export labels.

- [ ] A .zip package containing files in YOLO format.
- [ ] A .zip package containing files in VOC XML format.
- [x] Single CSV file.

Fig. 22. MakeSense Exporting Options

Creating the strawberry dataset involved merging labeled images from more than 5 separate students. Many images were labeled with the same date or the same cluster number, which prevented some images from merging into the same directory. A python script was written to rename all images to have unique names and to follow a uniform naming convention for each student.

4.1.1 CSV Parsing and Reformatting

The MakeSense program was great for obtaining the coordinates for the bounding boxes, but the format was slightly different than the one needed by the pandas library. The CSV file that was generated by the MakeSense program contains the label name, bounding box, file name, width and height of the image. A few lines of the CSV file generated by the MakeSense program is shown in Figure 23 below.

```
Green Strawberry,3021,13,104,116,c11-4-13-19.JPG,4032,3024
Green Strawberry,3200,181,92,133,c11-4-17-19.JPG,4032,3024
Green Strawberry,3027,111,92,92,c11-4-17-19.JPG,4032,3024
Green Strawberry,711,0,139,134,c11-4-17-19.JPG,4032,3024
```

Fig. 23. CSV File Generated from MakeSense

The pandas method "read_csv" requires a unique CSV file for each training image, where the first number is the number of bounding boxes and each line after contains the coordinate for the bounding box. An example is shown in Figure 24 below.

```
3
66 641 154 133
1611 1019 104 91
1566 932 108 83
```

Fig. 24. CSV File for pandas.read_csv(path)

A short python script was written to take the large CSV file generated from the MakeSense program and convert it into small CSV files per each input image. With the CSV files properly formatted, they're ready to use for two functions.

The first is to extract green strawberry snippets from the field images in 2019. These clippings were combined with the data from 2018 to make up the 1286 positive examples used in the dataset. The second function of the annotated data, was to properly evaluate the end-to-end object detection accuracy, by comparing bounding boxes of green berry guesses with the actual coordinates from the CSV files. This comparison is achieved by taking the IOU of both bounding boxes where an IOU of 1 means both boxes are perfectly aligned and 0 means there's no overlap. Various IOU thresholds were tested and evaluated further in the localization results in section 4.5.

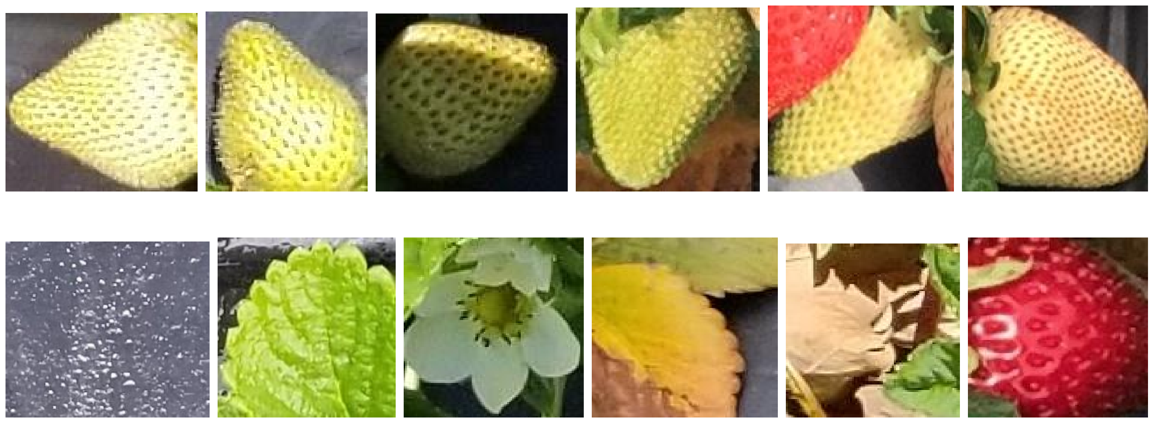


Fig. 25. Positive and Negative Snippets

The full dataset includes 1286 unique green strawberry snippets with various background elements like leaves, flowers and patches of dirt. Examples of various snippets are shown in Figure 25 above, where the top row contains positive images and the bottom contains negative examples. The number of background images is variable and can be increased or decreased depending on the number of positive samples used. The number of positive samples were increased using data augmentation. Typically 40,000 to 60,000 background images of size 128x128 were used for 80% training, 10% validation, and 10% test. Just over 105,000 images were used to train the VGG, MobileNet and ResNet models with a ratio of approximately 1 positive sample for each negative one.

4.1.2 Data Augmentation Implementation

Data augmentation was handled using the cv2 library. Affine transforms, rotations and varying brightness scales could be used to increase the number of positive green strawberries. Without data augmentation our dataset contains only 1286 green strawberry snippets, but by adding the augmented data, we can now have more than 60,000 unique positive green strawberry snippets as an extreme example.

Augmentation ratios of 48:1, 32:1, and 15:1 were tested alongside no augmentation. With a ratio of 48:1, more than 60,000 positive images are produced. The large number of positive images is useful for the classifier as CNNs are heavily data dependent, but the high ratio of augmentation can decrease the quality of the input data. The ratio of positive to negative images in the training data also affects the sensitivity of the classifier. As an example, no data augmentation would drive down the prediction threshold lower for a green strawberry. When using data augmentation, a general rule was to keep the number of positive and negative images to a 1:1 ratio.

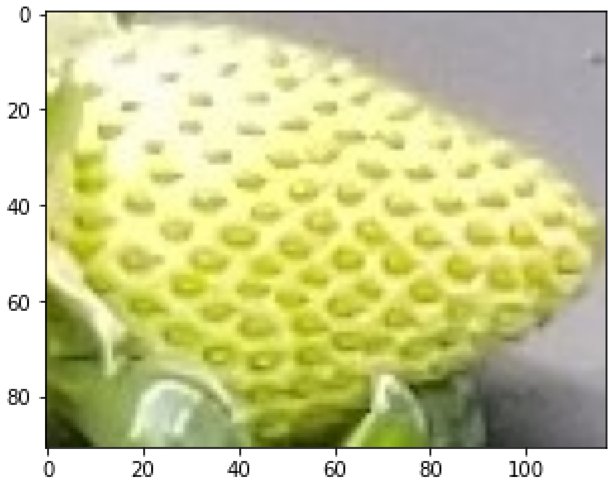


Fig. 26. Original Green Strawberry Clipping

Image translations and rotations were made using the cv2.warpAffine method. By default, empty space created by the warpAffine method is filled using black pixels but this can be changed using varying border modes. The black pixels contain no information and having them could harm the accuracy of the classifier as it learns. Figure 26 above shows a normal strawberry clipping.

The following figures show the different border modes and how they affect the image. Figures 27-30 show differing border modes for fig 26 rotated 45 degrees.

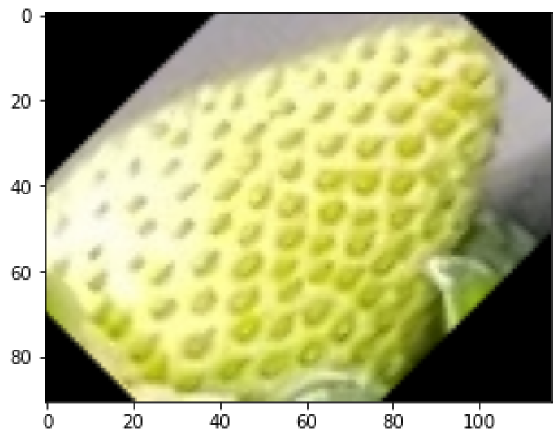


Fig. 27. Rotation with No Border Mode

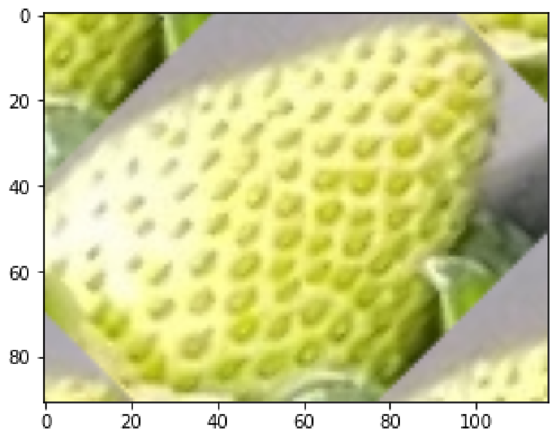


Fig. 28. Rotation with BORDER_WRAP

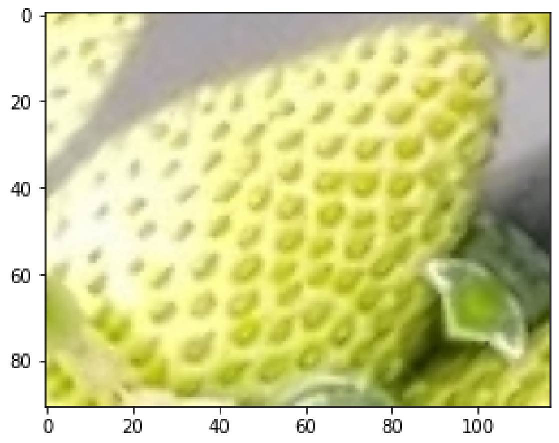


Fig. 29. Rotation with BORDER_REFLECT

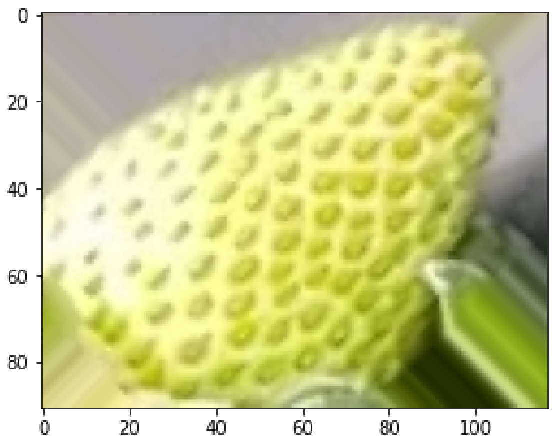


Fig. 30. Rotation with BORDER_REPLICATE

No border mode in Figure 27 is clearly poor for the classifier. The border wrap feels unnatural and the border replicate creates long streaks of color in the image. The border reflect felt the most natural of the cv2 border modes and was used for the image augmentation. Figures

31-34 show the border modes when applied to translated images. Border reflect feels the most natural in this case as well.

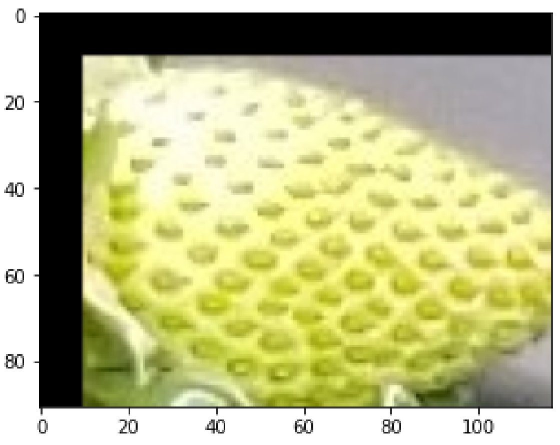


Fig. 31. Translation with No Border Mode

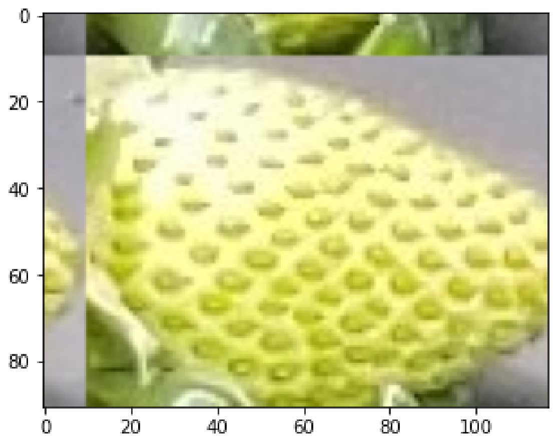


Fig. 32. Translation with BORDER_WRAP

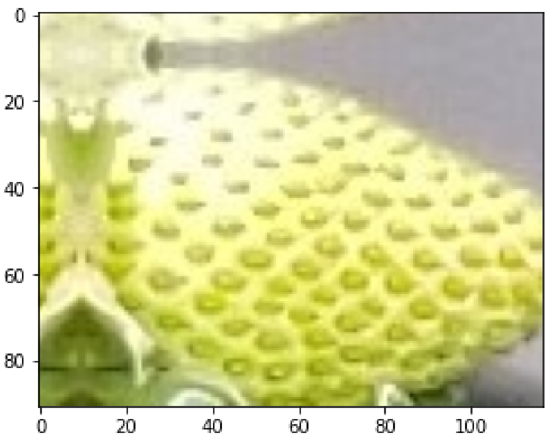


Fig. 33. Translation with BORDER_REFLECT

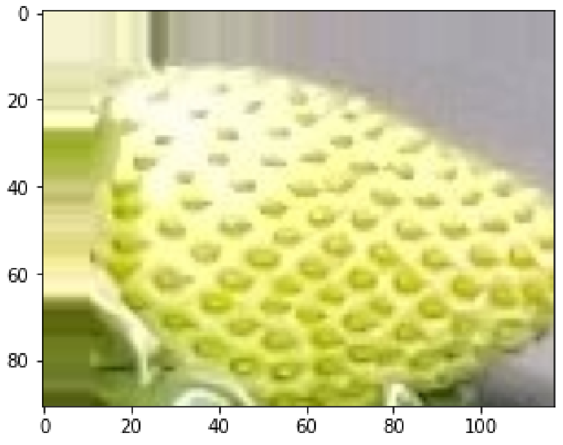


Fig. 34. Translation with BORDER_REPLICATE

Data augmentation was only applied to positive snippets as there were plenty of background elements to add to the database. Thousands of images of flowers, leaves, dirt and black tarp were added from the data collected in 2018. Negative hard mining was also applied to mitigate false positives and increase negative examples overall. Adding these negative examples further improved the accuracy of the classifier.

4.2 Classification Implementation

In this section, we will go over the general implementation of the classifier and explain some of the most important parameters like batch size, number of epochs, and number of validation steps. Many of the overall design choices, like input blob size, were carried over for each DNN when testing VGG16, MobileNet and ResNet50. The order in which each method is explained is consistent with how the IPython code was implemented on the Jupyter notebook.

4.2.1 Input Blob Size

The default input blob size for the keras models VGG16, MobileNet and ResNet50 is 244x244 over the RGB spectrum. The average size of our strawberry snippets is just over 100x100 so the 244x244 input size is unnecessarily large. A typical smaller input size is 128x128. Both 224 and 128 were tested and the 128 size proved to be useful as the image size takes up approximately one third of the memory as the 224. This made the model easier to train and allowed for more training images to be added using the same amount of RAM.

4.2.2 Model Parameters

The dataset was split into 10% test, 10% validation, and 80% training. The model was optimized for validation accuracy over 100 epochs, 50 steps per epoch, 10 validation steps, and a batch size of 32. After each model was trained, the accuracy and loss were analyzed so see which architecture performed the best.

4.3 Classification Results

Each model was trained with a high augmentation ratio of 48:1. Other ratios like 32:1 and 15:1 were tested for VGG16, but the accuracy fell as fewer positive training images were used. Each model was also adapted for the 128x128 blob size to include a head of average pooling, relu activation function, and softmax. Table VII below shows the accuracy and loss for VGG16, MobileNet and ResNet50 using the dataset with just over 105,000 total images. All of the DNNs were tested using an Intel i7 processor, an AMD Radeon RX 5700 XT GPU and 16GBs of RAM.

Table 7. Classifier Results

Model	Val Accuracy (%)	Test Accuracy (%)	Val Loss (%)	Loss (%)	Test Time (s)
VGG16	**99.69**	93.57	01.64	06.75	221
MobileNetV2	91.56	83.13	20.97	24.37	**53**
ResNet50	99.687	95.28	**00.64**	**03.21**	160
ResNet101	99.375	**95.42**	01.23	03.27	208
ResNet152	99.37	94.01	1.69	3.86	220

Both VGG16 and ResNet50 performed extremely well with test accuracies of 93.5% and 95.3% respectively. MobileNet's accuracy was worse at 83.1% but this was expected as the network had far fewer trainable parameters than the others. The sections below analyze the classification results further for each DNN model.

4.3.1 VGG16 Classification Results

Our VGG16 model consists of nearly 15 million parameters with 13 convolutional layers, 5 max pooling layers and 3 fully connected layers. The parameter count is lower than the standard VGG16 model as the input blob size has been significantly reduced. With 100 epochs, 50 steps per epoch, and 10 validation steps, the model takes around an hour to train. Figure 35 below shows the training and validation loss and accuracy.

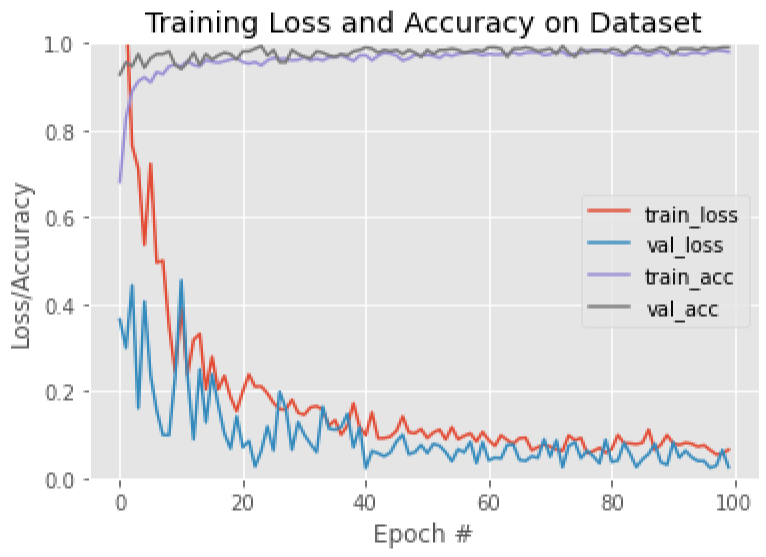


Fig. 35. VGG16 Training Loss and Accuracy

The VGG16 model had the slowest test time but great validation and test accuracies. VGG16 will be one of the top models to analyze for the end-to-end green strawberry detection once the localization algorithm is added.

4.3.2 MobileNetV2 Classification Results

Our MobileNetV2 implementation consists of 2.4 million parameters with 53 convolutional layers. With 100 epochs, 50 steps per epoch, and 10 validation steps, the model takes under an hour to train. Figure 36 below shows the training and validation loss and accuracy.

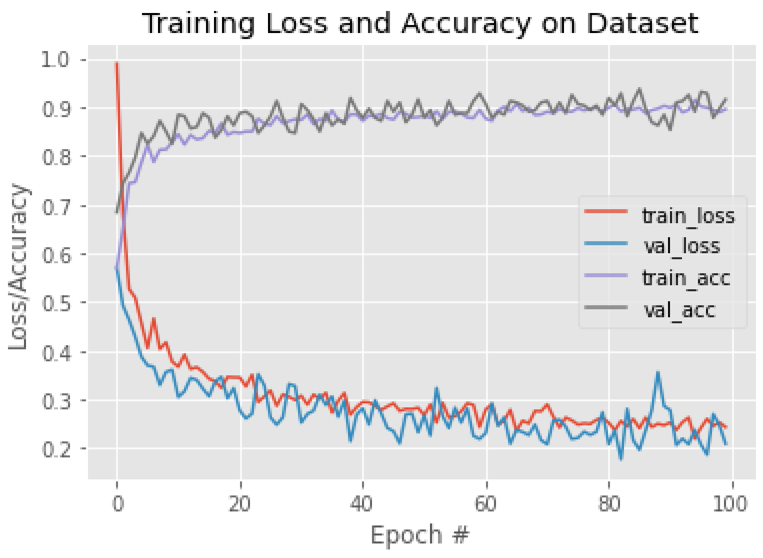


Fig. 36. MobileNetV2 Training Loss and Accuracy

4.3.3 ResNet50 Classification Results

ResNet50 contains 24 million parameters over 50 convolutional layers. The model was trained with the same parameters as the previous models and also took an hour to train. Figure 37 shows the training loss and accuracy of ResNet50.

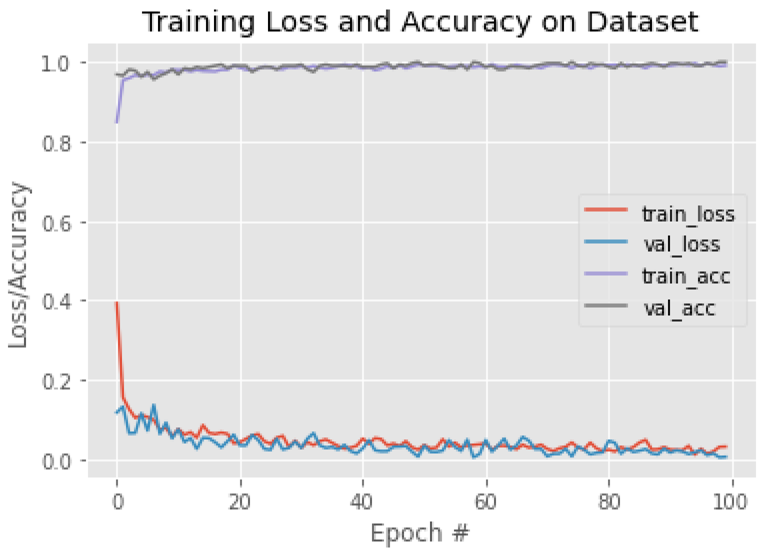


Fig. 37. ResNet50 Training Loss and Accuracy

ResNet101 had the highest test accuracy, while ResNet50 had the lowest loss. All of the ResNet models had high accuracy and low loss, making them ideal DNNs to use for the end-to-end green strawberry detection. ResNet50 had a much lower test time compared to ResNet101 and ResNet152 at 160 seconds compared to 208 and 220. Because of the lower test time, ResNet50 was analyzed more thoroughly for the end-to-end object detection and compared against VGG16 to see which DNN architecture performed best. Additionally, the fewer layers and trainable parameters meant Resnet50 was less prone to overfitting problems than its 101 and 152 counterparts. In the next section, we will discuss the overall object detection results and analyze the accuracy when the R-CNN selective search algorithm is added.

4.4 Localization Implementation

For this project, selective segmentation was tested against the sliding window algorithm, but sliding windows returned nearly 10 times the bounding boxes as SS and the object detection took 3-5 times longer to execute. The large number of ROIs returned by sliding windows also increased the false positive rate which negatively affected the overall accuracy. SS proved to be far more effective than sliding windows and was analyzed in depth.

Selective segmentation can be fine tuned using five various strategies. Strategies can be applied to look for similarities in color, fill, multiples, size or texture [14]. The fill strategy was used as it had the highest recall, where recall is the ratio of true positives over the total proposed bounding boxes. The strategy helps determine which regions are similar when merging bounding boxes.

Selective search can be accessed using the cv2 library ximgproc [14]. The selective segmentation algorithm takes 10-12 minutes to run for a full scale 3024x4032 field image. The initial test set contains 88 images, meaning the localization algorithm alone will take nearly 15 hours to run. Downscaling was used to reduce the field image resolution which exponentially decreased the time spent on the selective segmentation algorithm. A downscaling factor of 2 takes the processing time down to a minute and a half, while a value of 4 takes it down to 8 seconds each. The localization algorithm returns fewer bounding boxes for the downscaled images but also reduces the recall rate. The highest downscaling factor that still had a decent recall rate was a factor of 6. This downscaling factor had a good balance of low processing time while maintaining a high accuracy.

The localization implementation involves applying selective segmentation to each field image in the test set to obtain a list of bounding boxes. Boxes are removed if they are smaller or larger in area than possible for a green berry and elongated boxes are also removed if their width is 5 times larger than their height or vice versa. Each box is rescaled to the 128x128 input blob size and inputted into the classifier to obtain a likelihood score of whether or not it is a green strawberry. Figure 38 below shows a diagram of the overall R-CNN implementation.

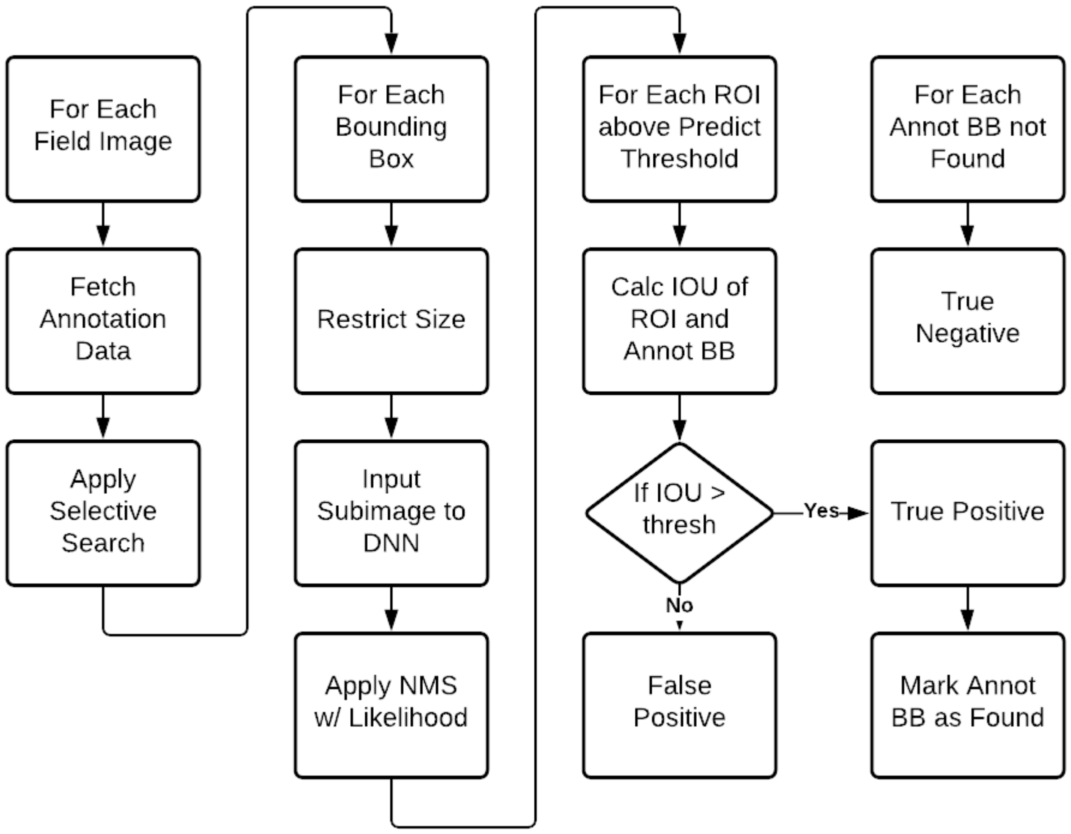


Fig. 38. R-CNN Object Detection Diagram

The selective segmentation algorithm often returned multiple overlapping bounding boxes. This would cause the classifier to falsely detect multiple green strawberries where there would only be one. To reduce overlapping boxes, non maxima suppression (NMS) was applied. The list of bounding boxes and likelihood values are passed into the non-maxima suppression method to reduce boxes that have an IOU value over 0.3. The final list contains regions of interest that are above the prediction threshold and are appropriately sized.

True positives were determined if a region had a likelihood value greater than the prediction threshold and if the box had an IOU value greater than the IOU threshold for any of the CSV annotation regions. False positives and false negatives were tracked to get an accurate picture of how the object detection was performing on the test set. False positives were also stored on disk to further analyze what regions the classifier thought were true berries but weren’t. These false positive images were also used for negative hard mining, where the images can be added into the training set later to reduce the false positives. Many trials were run with varying DNN models, downscaling factors, and thresholds. The object detection results are analyzed below.

4.5 Object Detection Results

VGG16 was the first DNN model to be implemented. Various IOU, NMS and prediction thresholds were tested to increase total recall for the selective search algorithm and to increase the total accuracy of the green strawberry detection model. The accuracy of the model was determined by dividing the true positives by the total guesses plus the false negatives. Figure 39 below shows how the accuracy was determined.

$$Object\ Detection\ Accuracy = \frac{True\ Positives}{True\ Positives + False\ Positives + False\ Negatives} = \frac{True\ Positives}{Total\ Guesses + False\ Negatives}$$

Fig. 39. Object Detection Accuracy Equation

Once the localization algorithm was optimized for recall by solidifying values for IOU and NMS, the classifier was added to obtain the final numbers for true positives, false positives and false negatives. Table VIII below shows a selection of some of the many trials run for each model.

Table 8. Object Detection Results

Model	Accuracy (%)	Time (mins)	predict val	d-scale factor	IOU val	NMS val	test set (clusters)
VGG16 (no aug)	45.9	33	0.05	6	0.3	0.3	11,12,13,14
VGG16 (15:1 aug)	42.5	40	0.6	6	0.3	0.3	11,12,13,14
VGG16 (32:1 aug)	45.2	46	0.4	6	0.3	0.1	11,12,13,14
VGG16 (48:1 aug)	71.0	35	0.875	6	0.3	0.3	11,12,13,14
MobileNetV2 (48:1 aug)	08.5	**10**	0.875	6	0.3	0.3	11,12
ResNet50 (48:1 aug)	63.0	15	0.85	6	0.3	0.3	11,12
ResNet50 (48:1 aug)	**74.0**	32	0.725	6	0.3	0.3	11,12,13,14

From all of the trials tested, ResNet50 had the best accuracy with a 74% accuracy using full data augmentation with the larger test set using clusters 11-14. VGG16 had the second best accuracy of 71% when using the 48:1 augmentation ratio. Overall, ResNet50 performed the best as the accuracy and test time were better than VGG16. Examples of true positives, false positives and false negatives from ResNet50 are shown in Figure 40 below. The first two images are two positives followed by two false positives and two false negatives. The false positives come from one berry containing too much red and another with an inaccurate bounding box returned from selective search. The two false negatives on the right also come from the selective search algorithm not returning bounding boxes for the smaller green berries.

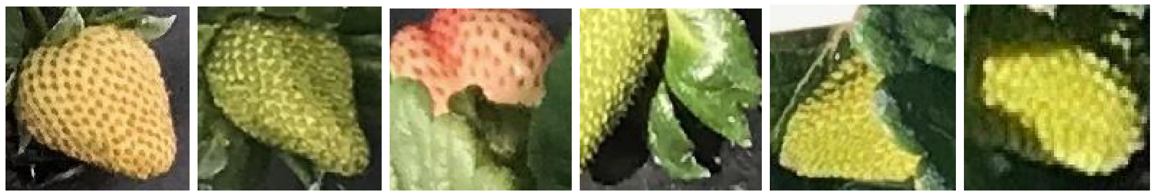


Fig. 40. True and False Positives / Negatives

The MobileNetV2 model accuracy was lower than all the other models tested. The overall detection accuracy was poor and returned many false positives. MobileNetV2 has 2.4 million trainable parameters which is far fewer than the 15 million of VGG16. The large number of false positives detections come from various leaves and background elements. The lightweight MobileNet architecture worked well for the multifruit detection in the previous research paper but ultimately the CNN had trouble differentiating leaves from the green strawberries [7]. Perhaps a fine tuned MobileNet like the one proposed in the strawberry and cherry detection paper would help, but that paper only analyzed ideal clippings of just the fruit [10]. MobileNetV2 runs much more quickly than VGG16 at around 2.5 times faster for test evaluation, but the efficiency is overshadowed by the high false positive rates.

Overall, ResNet50 had higher accuracy rates than VGG16 when averaged over many tests. The accuracy averaged in the low 70s and found most green strawberries. Figure 41 below shows a field image with blue boxes as true positives and red boxes as false negatives. The R-CNN classifier found 9 of the 10 green berries. The green berry in the middle left was missed.

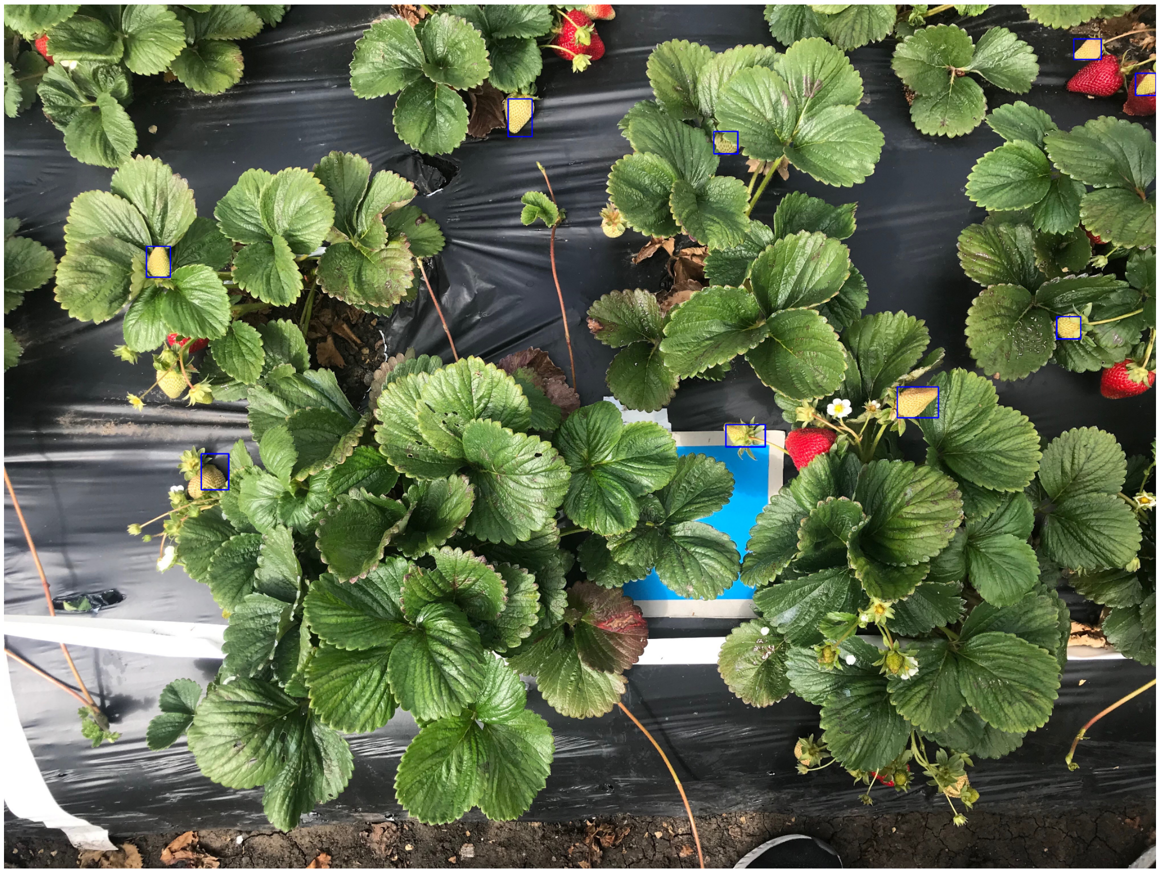


Fig. 41. Test Field Image with Proposed Boxes

4.5.1 Object Detection Conclusion

The overall R-CNN detection accuracy was much lower than the classification accuracy. As an example, the test accuracy for ResNet50 was 95.3% while the overall detection accuracy was 74%. This was due to the selective search recall rate.

With a downscaling factor of 6 and an IOU value of 0.3, the selective search algorithm had a recall of 90% and returned approximately 2000 boxes per image. Using an IOU value of 0.5 made the recall dip to 81%. This is because the regions proposed by selective search are often misaligned and imperfect. Because of this, the IOU value was lowered to keep the recall high and used to properly evaluate each DNN model. With an accuracy of 74%, we will see how useful the R-CNN model is for crop yield prediction.

4.6 Crop Yield Prediction

The total number of green strawberries per field image can be guessed by summing the number of proposed boxes that are above the prediction threshold. This number can be added with all the field images from that day to get an approximation for the number of green strawberries that are growing for the given clusters. Using this value, we can hope to approximate the number of red strawberries that will be grown one week from that date.

To accurately create a crop yield prediction algorithm, we will have to utilize all the data we have available. Currently, we have the number of visible green strawberries from our annotation data, the number of visible red ones from counting them per each field image and the number of green strawberry guesses per image from our best R-CNN implementation using ResNet50. We also have two other metrics for the actual green and red strawberry counts for the 2019 harvesting season. The actual berry counts were noted at the same time the images were taken. This information will be important as many strawberries could be occluded by the larger leaves.

All five of these metrics were analyzed in a spreadsheet and plots were made to observe the correlation between each value. In the year 2019, data was collected for 28 clusters twice a week from February to June. Four different students collected data from clusters 1-7, 8-14, 15-21 and 22-28. Table IX below shows the data from clusters 8 through 14 for each date.

Table 9. Strawberry Data from 2019 and R-CNN Implementation

Date	Total Green Guesses	Total Green Annotations	Total Red Annotations	Field Actual Green	Field Actual Red
2-7-19	0	0	0	0	0
2-18-19	0	0	0	0	0
2-21-19	0	0	0	0	0
2-24-19	0	0	0	0	0
3-14-19	0	0	0	0	0
3-17-19	0	0	0	0	0
3-31-19	0	0	0	0	0
4-6-19	0	0	0	0	1
4-10-19	0	0	1	0	0
4-13-19	0	0	3	0	0
4-17-19	0	0	4	0	1
4-24-19	0	0	4	0	0
4-27-19	0	0	5	0	1
5-1-19	14	16	15	0	3
5-4-19	15	12	19	0	6
5-8-19	33	27	33	0	24
5-11-19	12	9	23	0	26
5-18-19	25	23	29	51	44
5-22-19	32	32	18	82	33
5-29-19	44	39	28	71	34
6-5-19	17	17	31	77	61
6-8-19	23	20	26	89	70

Data for the actual green strawberry count metric was first collected starting May, 22nd of 2019. This unfortunately will make the graphs including actual green berries less significant. This is less concerning though as the definition of what a green strawberry is has certainly changed since first collecting the data in 2019. A graph of the three green strawberry metrics for guesses, annotations and actual count is shown in Figure 42 below.

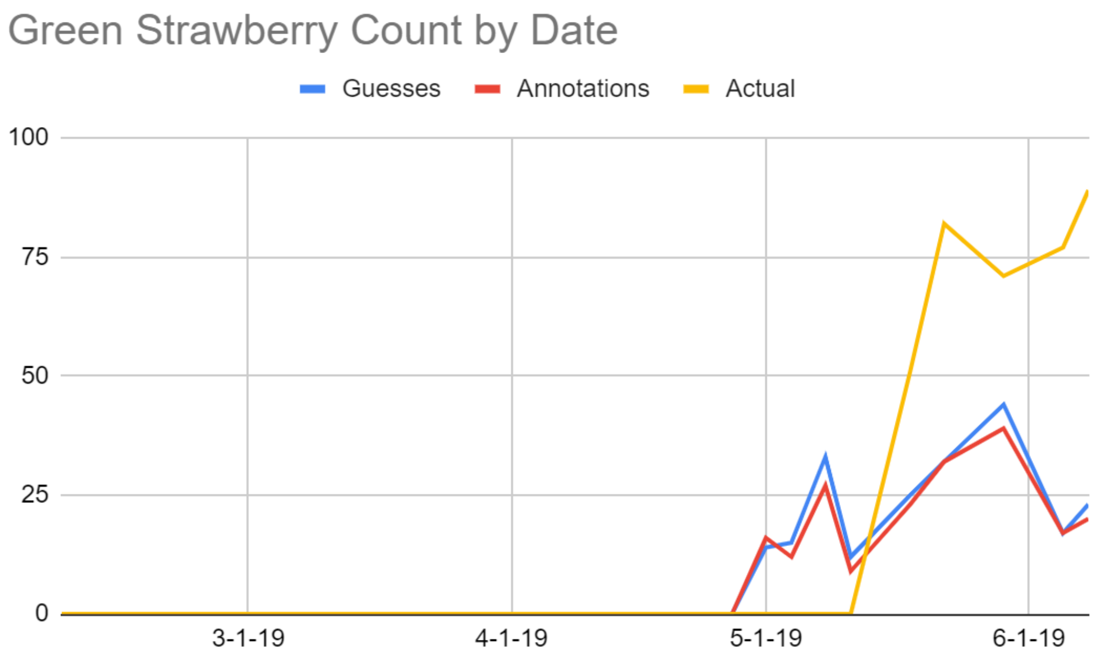


Fig. 42. Green Strawberry Count by Date

Despite the object detection accuracy being at 74%, the guesses and annotations are much more correlated as many of the false positives and false negatives cancel each other out. When the actual green strawberries are first counted in May, the values are nearly double that of the visible count. This is likely due to a combination of leaves occluding green strawberries and the definition of a green strawberry being more ‘loose’ in 2019. Many of the counted berries from that year should have been classified as buds instead.

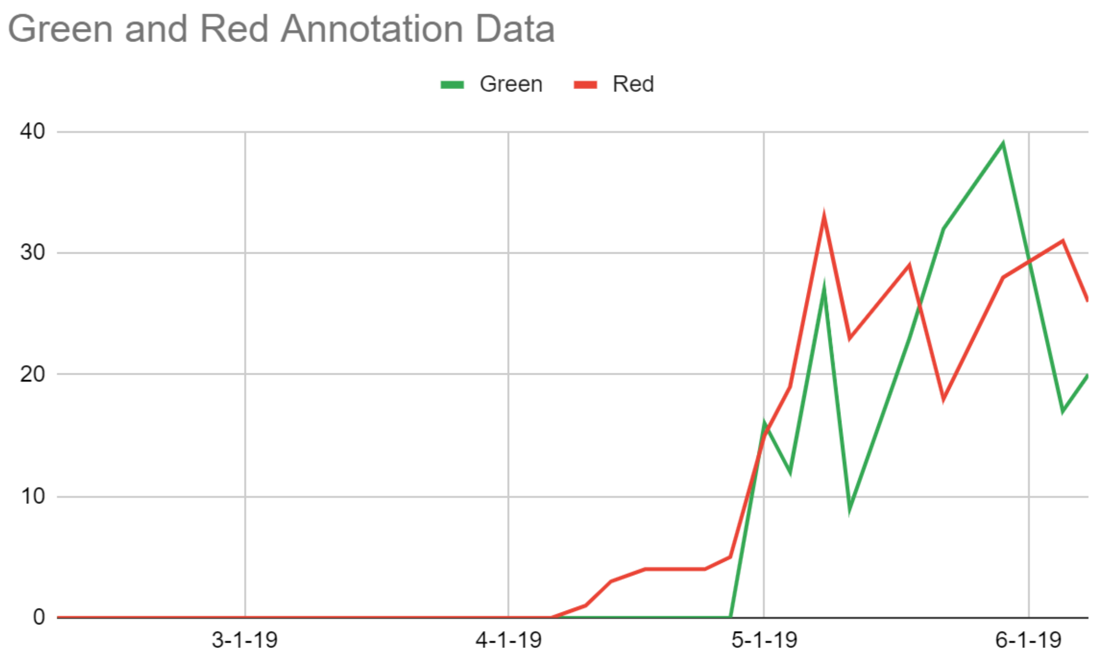


Fig. 43. Green and Red Annotation Data

Figure 43 above shows the green and red annotation data. For the strawberry growth cycle, one would assume the number of green berries would first peak before the red. However, the visible red strawberry count first starts on April 10th and the green starts on May 1st. This happens because of one main reason. The visible red strawberries starting in April are berries from other clusters that find themselves across the border of the field image. Figure 44 below shows an example of red strawberries from another cluster finding themselves in the image.

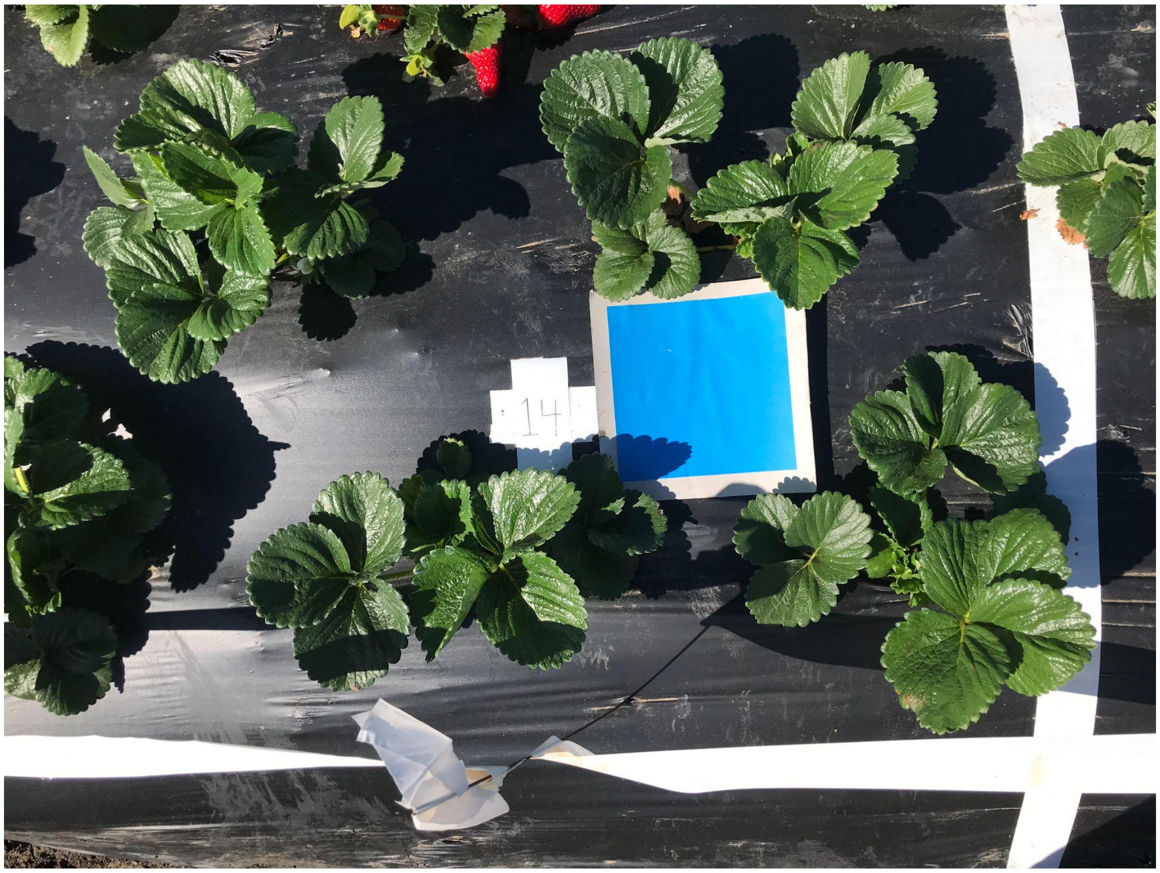


Fig. 44. Unwanted Red Strawberries in Cluster 14 Data

Despite this initial peak of red strawberries in Figure 34, the green and red annotation data are fairly well correlated with a correlation value of 0.87. A second peak of green berries occurs in late May in Figure 43. The green strawberry data for this peak is larger in count than the red berries. This is due to rot in the field reducing the number of green berries that fully develop into red berries. Without rot, the number of red berries should be higher in May and June.

The most important correlation is the green strawberry guesses with the red annotation and red actual data. Figure 45 below shows a plot of the guesses from our R-CNN model compared with the red annotation and red actual count.

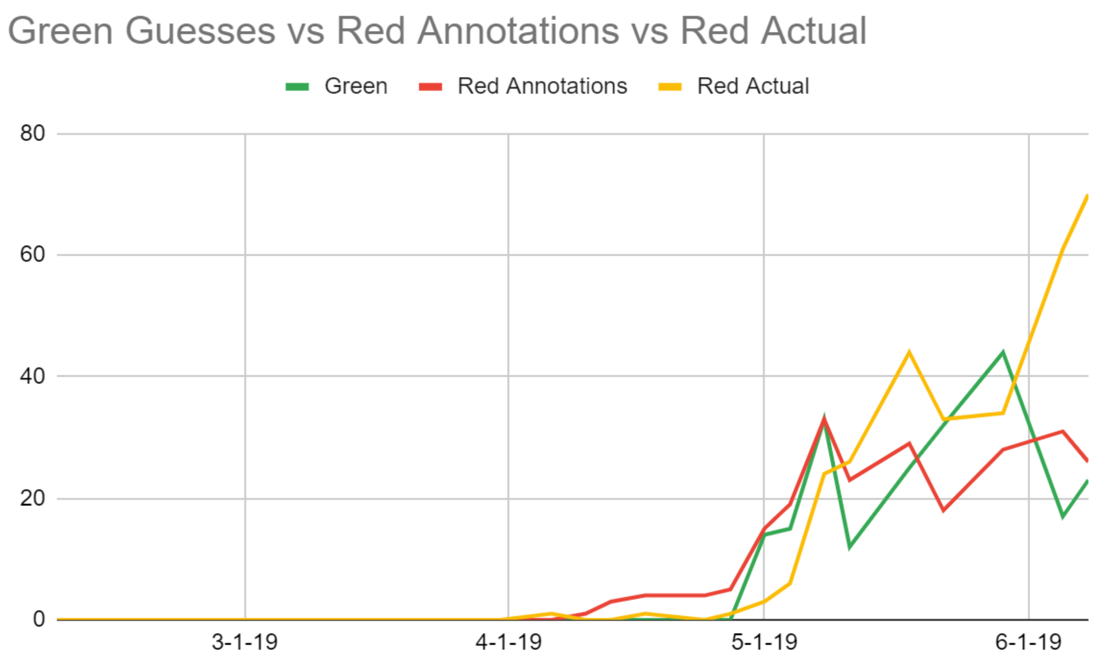


Fig. 45. Green Guesses vs Red Annotations vs Red Actual

The green guesses from our R-CNN model were compared with the red annotation and red actual data to find the correlation between the two. The correlation between the green guesses and the red annotation data was 0.89 and the correlation between the green guesses and the red actual data was 0.73. It makes sense that the results from the R-CNN model are more highly correlated with the visual data rather than the actual count.

To actually analyze the ability of the R-CNN model to predict red strawberries the following week the correlation between the green guesses and the actual red count was also analyzed when the green data is staggered one week in advance. If the green strawberry guesses correlate with the actual red count in one weeks time, then the crop yield prediction would be possible. Figure 46 below shows the green guess and the actual red count when staggered by one week.

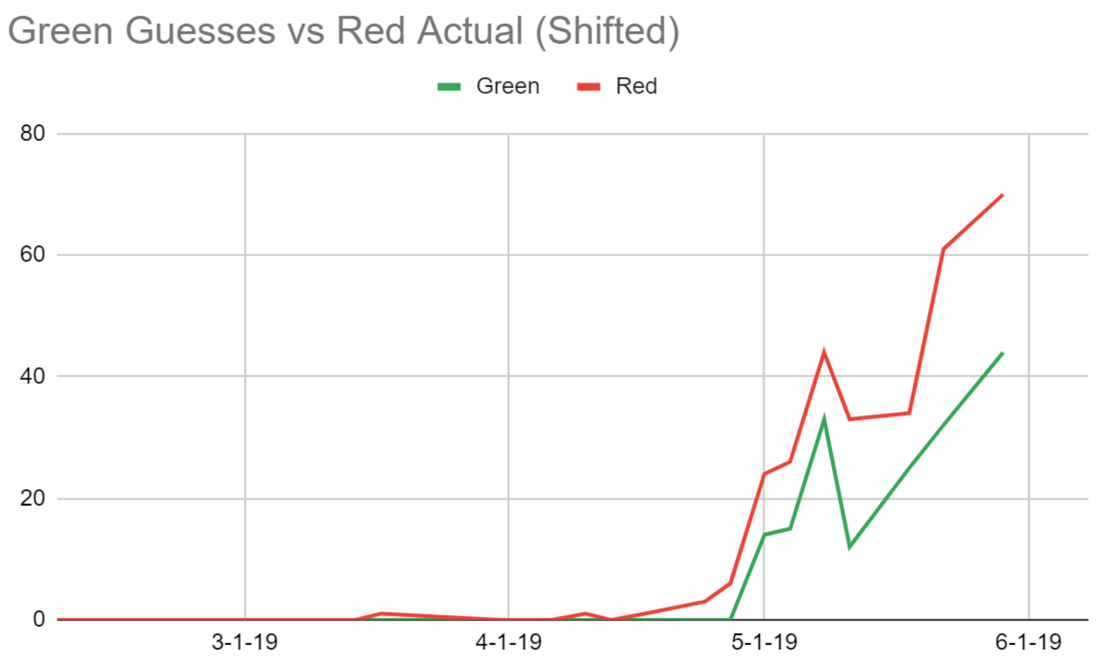


Fig. 46. Green Guesses vs Red Actual (Shifted One Week)

The correlation between the guesses and the actual red strawberries when staggered by one week is 0.977. Both line graphs are highly correlated while the actual red count is biased upwards by some margin. This makes sense as our green guesses come from the visual images and overlook berries obstructed by leaves.

The average difference between the actual red count and our green guesses is a factor of 1.77. This factor, when multiplied by the green strawberry guesses gives an approximation for the red actual count. When there is visible green strawberry data, multiplying by this coefficient of 1.77 gives a prediction for the red count. This prediction is correlated with the actual red count with a correlation coefficient of 0.902. Figure 47 below shows the shifted data when the green guesses is multiplied by this coefficient.

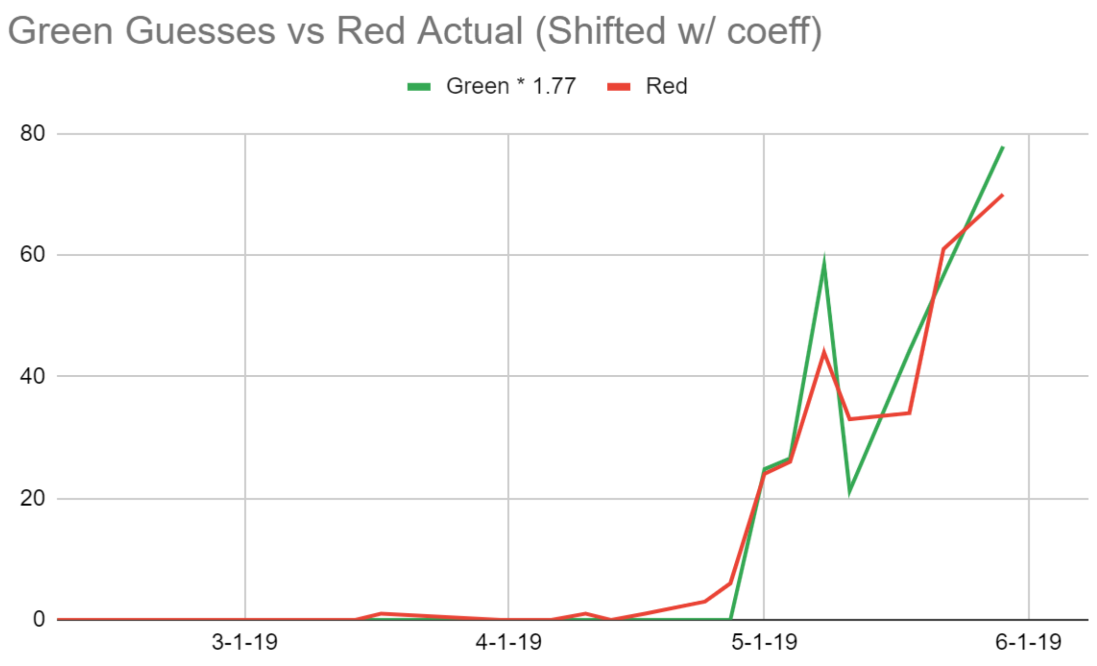


Fig. 47. Green Guesses vs Red Actual (Shifted One Week w/ Coeff)

This correlation is a promising indicator that a neural network can accurately predict the red strawberry count for the following week. Hopefully with more data, a more accurate coefficient can be found to predict the red strawberry count for future harvests.

Chapter 5
CONCLUSION & FUTURE WORKS

R-CNN was implemented as a modern approach for detecting green strawberries. Deep neural networks require large amounts of data to run effectively. Images taken from Cal Poly's strawberry field in 2018 and 2019 were used to create the dataset which was expanded using data augmentation and negative hard mining. These images were annotated using the MakeSense program and the generated CSV files were used for both training and testing.

The R-CNN architecture uses the selective search localization algorithm to find all the ROIs of an image while maintaining high recall. Various DNNs can be used to classify the image regions like VGG16, MobileNet or ResNet. MobileNet is a great choice for applications that have limited resources, but the model's test accuracy didn't compare to VGG and ResNet. MobileNet would be ideal for a RaspberryPi implementation but the parameters would need to be fine tuned like the model described by Venkatesh [10]. The accuracy of a DNN can be improved using non-maxima suppression, data augmentation and negative hardmining. ResNet50 had the highest classification accuracy at 95.3% and the best overall object detection results at 74%. The pairing of selective search and ResNet50 proved to be effective for detecting the subtle green berries in each field image.

The crop yield prediction results are very promising with the correlation between the R-CNN prediction and the red visual data, when staggered by one week, being at 0.977. A linear coefficient of 1.77 can be used to predict the red count with a correlation of 0.902. For the future, it would be interesting to test the model live during the strawberry's growth cycle, to see if the R-CNN model could guess the red berry count. If the model can truly predict the red count with a 90% accuracy, then more field testing should be done and steps should be taken towards creating a working prototype that a farmer could use. This prototype could involve using the ResNet50 architecture or trading it out with a fine-tuned Mobilenet architecture if resources are limited.

For the future, it would be useful to quantitative test other augmentation ratios to see if increasing the amount of positive data would further increase the classifier's accuracy. The ratio of 48:1 was the highest tested due to hardware limitations, but a more powerful computer with more RAM could implement ratios that are much higher. It would be insightful to see how much data could be augmented before the loss in quality data outweighs the sheer number of positive images. Quantitatively testing and analyzing various augmentation types and border modes would give further validation of the quality of the augmented data.

Additionally, testing other input blob sizes besides 128x128 and 224x224 would be useful. A blob size of 64x64 may be sufficient for extracting the subfeatures of a green berry. If so, the classifier would be much more efficient when processing each input blob and more training images could be added as less memory is used.

It would also be insightful to test other localization algorithms alongside other architectures besides R-CNN, like YOLO or SSD. It would also be interesting to compare R-CNN with its newer counterparts of Fast R-CNN and Faster R-CNN. Faster R-CNN uses a region proposal network to find ROIs so it would be useful to compare its recall with selective search. In order to increase the overall accuracy, the recall rate for the localization algorithm needs to be improved. Overall, keeping the base architecture of R-CNN the same and testing various neural networks was good for properly analyzing each DNN.

BIBLIOGRAPHY

[1] CalStrawberry, “California Strawberry Farming”, California Strawberry Commision, 2018.

[2] Washburn, Kaitlin, “In California farm country, growers struggle with labor shortage”, USA Today, April 6, 2020.

[3] S. Smith, C. Garcia, “Worker shortage Hurts California’s Agriculture Industry”, Planet Money, 2018.

[4] Fitter, Yavisht, and Jane Zhang. *Strawberry Detection Under Various Harvestation Stages*. Web.

[5] X. Li, J. Li and J. Tang, "A deep learning method for recognizing elevated mature strawberries," *2018 33rd Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, 2018, pp. 1072-1077, doi: 10.1109/YAC.2018.8406530.

[6] R. Zhang, X. Li, L. Zhu, M. Zhong and Y. Gao, "Target detection of banana string and fruit stalk based on YOLOv3 deep learning network," *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, 2021, pp. 346-349, doi: 10.1109/ICBAIE52039.2021.9389948.

[7] H. Basri, I. Syarif and S. Sukaridhoto, "Faster R-CNN Implementation Method for Multi-Fruit Detection Using Tensorflow Platform," *2018 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC)*, 2018, pp. 337-340, doi: 10.1109/KCIC.2018.8628566.

[8] M. Pandiya, S. Dassani and P. Mangalraj, "Analysis of Deep Learning Architectures for Object Detection - A Critical Review," *2020 IEEE-HYDCON*, 2020, pp. 1-6, doi: 10.1109/HYDCON48903.2020.9242776.

[9] Z. Zhao, P. Zheng, S. Xu and X. Wu, "Object Detection With Deep Learning: A Review," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 11, pp. 3212-3232, Nov. 2019, doi: 10.1109/TNNLS.2018.2876865.

[10] Venkatesh, N. Y, S. U. Hegde and S. S, "Fine-tuned MobileNet Classifier for Classification of Strawberry and Cherry Fruit Types," *2021 International Conference on Computer Communication and Informatics (ICCCI)*, 2021, pp. 1-8, doi: 10.1109/ICCCI50826.2021.9402444.

[11] Thiago T. Santos, Leonardo L. de Souza, Andreza A. dos Santos, Sandra Avila, Grape detection, segmentation, and tracking using deep neural networks and three-dimensional association, Computers and Electronics in Agriculture, Volume 170, 2020, 105247, ISSN 0168-1699, https://doi.org/10.1016/j.compag.2020.105247.

[12] K. E. A. van de Sande, J. R. R. Uijlings, T. Gevers and A. W. M. Smeulders, "Segmentation as Selective Search for Object Recognition," *2011 International Conference on Computer Vision*, 2011, pp. 1879-1886, doi: 10.1109/ICCV.2011.6126456.

[13] K. Team, “Keras Documentation: Keras applications,” *Keras*. [Online]. Available: https://keras.io/api/applications/. [Accessed: 14-Jan-2022].

[14] “CV::ximgproc::Segmentation::Selectivesearchsegmentationstrategy class reference,” *OpenCV*. [Online]. Available: https://docs.opencv.org/4.x/dc/d7a/classcv_1_1ximgproc_1_1segmentation_1_1SelectiveSearchSegmentationStrategy.html. [Accessed: 14-Jan-2022].

[15] “Pandas,” *pandas*. [Online]. Available: https://pandas.pydata.org/. [Accessed: 14-Jan-2022].

[16] S.-H. Tsang, “Review: MOBILENETV2 - light weight model (image classification),” *Medium*, 01-Aug-2019. [Online]. Available: https://towardsdatascience.com/review-mobilenetv2-light-weight-model-image-classification-8febb490e61c. [Accessed: 14-Jan-2022].

[17] “Image processing (imgproc module),” *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d7/da8/tutorial_table_of_content_imgproc.html. [Accessed: 14-Jan-2022].

[18] G. Learning, “What is VGG16? - introduction to VGG16,” *Medium*, 23-Sep-2021. [Online]. Available: https://medium.com/@mygreatlearning/what-is-vgg16-introduction-to-vgg16-f2d63849f615. [Accessed: 14-Jan-2022].

[19] S. PA, “An overview on MobileNet: An Efficient Mobile Vision CNN,” *Medium*, 11-Jun-2020. [Online]. Available: https://medium.com/@godeep48/an-overview-on-mobilenet-an-efficient-mobile-vision-cnn-f301141db94d. [Accessed: 14-Jan-2022].

[20] V. Feng, “An overview of ResNet and its variants,” *Medium*, 17-Jul-2017. [Online]. Available: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035. [Accessed: 14-Jan-2022].

[21] S. Saha, “A comprehensive guide to Convolutional Neural Networks - the eli5 way,” *Medium*, 17-Dec-2018. [Online]. Available:

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed: 27-Jan-2022].

[22] R. Gandhi, "R-CNN, fast R-CNN, Faster R-CNN, YOLO - object detection algorithms," *Medium*, 09-Jul-2018. [Online]. Available: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e. [Accessed: 27-Jan-2022].

[23] "Introduction to yolo algorithm for object detection," *Section*. [Online]. Available: https://www.section.io/engineering-education/introduction-to-yolo-algorithm-for-object-detection/. [Accessed: 27-Jan-2022].

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," *arXiv.org*, 29-Dec-2016. [Online]. Available: https://arxiv.org/abs/1512.02325. [Accessed: 27-Jan-2022].

[25] J.R.R. Uijlings, "Selective search for object recognition." [Online]. Available: https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013/UijlingsIJCV2013.pdf. [Accessed: 27-Jan-2022].

[26] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *arXiv.org*, 22-Oct-2014. [Online]. Available: https://arxiv.org/abs/1311.2524. [Accessed: 01-Feb-2022].

APPENDIX

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import os,cv2,keras
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf


# In[ ]:


path = "D:/Datasets/Strawberry 2019/Images"
annot = "D:/Datasets/Strawberry 2019/Annotations"


# In[ ]:


cv2.setUseOptimized(True);
ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()


# In[ ]:


train_images=[]
train_labels=[]
test_images=[]
test_labels=[]


# In[ ]:


def get_iou(bb1, bb2):
    assert bb1['x1'] < bb1['x2']
    assert bb1['y1'] < bb1['y2']
    assert bb2['x1'] < bb2['x2']
    assert bb2['y1'] < bb2['y2']

    x_left = max(bb1['x1'], bb2['x1'])
    y_top = max(bb1['y1'], bb2['y1'])
    x_right = min(bb1['x2'], bb2['x2'])
    y_bottom = min(bb1['y2'], bb2['y2'])

    if x_right < x_left or y_bottom < y_top:
        return 0.0

    intersection_area = (x_right - x_left) * (y_bottom - y_top)

    bb1_area = (bb1['x2'] - bb1['x1']) * (bb1['y2'] - bb1['y1'])
    bb2_area = (bb2['x2'] - bb2['x1']) * (bb2['y2'] - bb2['y1'])

    iou = intersection_area / float(bb1_area + bb2_area -
intersection_area)
    assert iou >= 0.0
    assert iou <= 1.0
    return iou


# In[ ]:


def nms(boxes, threshold=.3):

    boxes = sorted(boxes, key=lambda boxes: boxes[2],
            reverse=True)

    new_boxes=[]

    new_boxes.append(boxes[0])

    del boxes[0]

    for index, box in enumerate(boxes):
        for new_box in new_boxes:
            a =
{"x1":box[0],"x2":box[0]+box[2],"y1":box[1],"y2":box[1]+box[3]}
```

```
            b = 
{"x1":new_box[0],"x2":new_box[0]+new_box[2],"y1":new_box[1],"y2":new_bo
x[1]+new_box[3]}
            if get_iou(a, b) > threshold:
                del boxes[index]
                break
        else:
            new_boxes.append(box)
            del boxes[index]
    return new_boxes


# In[ ]:


def ResizeImage(im, factor):
    width = int(im.shape[1] / factor)
    height = int(im.shape[0] / factor)

    # dsize
    dsize = (width, height)

    # resize image
    return cv2.resize(im, dsize)


# In[ ]:


def limit_size(boxes):
    max_area = 150**2 // (downscale_factor**2)#(150 * (1/IOU_VAL))**2
// (downscale_factor**2) # max area defined by scale and IOU

    min_area = 50**2 // (downscale_factor**2)#(50 * (IOU_VAL/1))**2 //
(downscale_factor**2)

    boxes = sorted(boxes, key=lambda boxes: boxes[2],
            reverse=True)

    new_boxes=[]

    for index, box in enumerate(boxes):
        if min_area < box[2]*box[3] and box[2]*box[3] < max_area: #
```

```
area range
            if box[2]*5 > box[3] and box[3]*5 > box[2]: # no elongated
rectangles
                new_boxes.append(box)

    del boxes
    return new_boxes


# In[ ]:


# adds green strawberries based on original bounding boxes
# data augmentation is applied using function rather than
# natural augmentations in the selective search algorithm
def add_strawberry_boxes(im, boxes, isTest, e):
    for i,box in enumerate(boxes):
        x1 = box['x1'] * downscale_factor
        y1 = box['y1'] * downscale_factor
        x2 = box['x2'] * downscale_factor
        y2 = box['y2'] * downscale_factor
        timage = im[y1:y2,x1:x2]
        new_file = "im_"+str(e)+"_"+str(i)+".jpg"
        resized = cv2.resize(timage, (128,128), interpolation =
cv2.INTER_AREA)
        augmentations = get_augmentations(resized)
        if isTest:
            test_images.extend(augmentations)
            test_labels.extend([1] * len(augmentations))
            write_path = "D:/Datasets/Strawberry
2019/Snippets/test/positive/"
        else:
            train_images.extend(augmentations)
            train_labels.extend([1] * len(augmentations))
            write_path = "D:/Datasets/Strawberry
2019/Snippets/training/positive/"
        cv2.imwrite(write_path+new_file,resized)


# In[ ]:


def add_strawberry_snippets(path, label, test=None, num_skip=0,
max_num=999999):
    for i,f in enumerate(os.listdir(path)):
        if i < num_skip:
            continue
        if i > num_skip+max_num:
            continue
        im = cv2.imread(os.path.join(path,f))
        if type(im) is type(None) or im.shape[0] == 0:
            continue
        resized = cv2.resize(im, (128,128), interpolation =
cv2.INTER_AREA)
        if (test == True or (test == None and i%10 == 0)):
            if label == 1:
                augmentations = get_augmentations(resized)
                test_images.extend(augmentations)
                test_labels.extend([1] * len(augmentations))
            else:
                test_images.append(resized)
                test_labels.append(label)
        else:
            if label == 1:
                augmentations = get_augmentations(resized)
                train_images.extend(augmentations)
                train_labels.extend([1] * len(augmentations))
            else:
                train_images.append(resized)
                train_labels.append(label)


# In[ ]:


def get_image_translations(image, dist):
    translations = []
    height, width = image.shape[:2]
    T = np.float32([[1, 0, 0], [0, 1, dist]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, dist], [0, 1, dist]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, dist], [0, 1, 0]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, dist], [0, 1, -1*dist]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, 0], [0, 1, -1*dist]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, -1*dist], [0, 1, -1*dist]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, -1*dist], [0, 1, 0]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    T = np.float32([[1, 0, -1*dist], [0, 1, dist]])
    img_translation = cv2.warpAffine(image, T, (width, height),
borderMode=cv2.BORDER_REFLECT)
    translations.append(img_translation)
    return translations


# In[ ]:


def get_image_rotations(image, angle_inc=10):
    rotations = []
    h, w = image.shape[:2]
    (cX, cY) = (w // 2, h // 2)
    i = 0
    while i < 360:
        M = cv2.getRotationMatrix2D((cX, cY), i, 1.0)
        rotated = cv2.warpAffine(image, M, (w, h),
borderMode=cv2.BORDER_REFLECT)
        rotations.append(rotated)
        i += angle_inc
    return rotations
```

```
# In[ ]:


def adjust_gamma(image, gamma=1.0):
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255
        for i in np.arange(0, 256)]).astype("uint8")

    return cv2.LUT(image, table)


# In[ ]:


def get_image_brightness(image):
    images = []
    im = adjust_gamma(image, 0.5)
    images.append(im)
    im = adjust_gamma(image, 0.75)
    images.append(im)
    im = adjust_gamma(image, 1.25)
    images.append(im)
    im = adjust_gamma(image, 1.5)
    images.append(im)
    return images


# In[ ]:


def get_augmentations(image):
    augmentations = get_image_translations(image, 10)
    augmentations.extend(get_image_rotations(image, 10))
    augmentations.extend(get_image_brightness(image))
    return augmentations


# In[ ]:


downscale_factor = 6
IOU_VAL = 0.3
```

```
NMS_VAL = 0.3


# In[ ]:


import re

for e,i in enumerate(os.listdir(annot)):
    try:
        # filename formatting
        filestart = i.rsplit('.', 1)[0]

        filename = None
        for f in os.listdir(path):
            if re.match(filestart, f):
                filename = f
        if filename == None:
            print("No file starting with " + filestart)
            continue
        print(e,filename)

        start_image = cv2.imread(os.path.join(path,filename))
        image = ResizeImage(start_image, downscale_factor)

        df = pd.read_csv(os.path.join(annot,i))
        num_berries = int(df.count())
        gtvalues=[]
        for row in df.iterrows():
            x1 = int(row[1][0].split(" ")[0]) // downscale_factor
            y1 = int(row[1][0].split(" ")[1]) // downscale_factor
            x2 = int(row[1][0].split(" ")[2]) // downscale_factor
            y2 = int(row[1][0].split(" ")[3]) // downscale_factor
            gtvalues.append({"x1":x1,"x2":x2,"y1":y1,"y2":y2})

        # adds positive strawberry images
        add_strawberry_boxes(start_image, gtvalues,
(i.startswith("c13") or i.startswith("c14")), e)

        ss.setBaseImage(image)
        ss.switchToSelectiveSearchFast()
        all_boxes = ss.process()
        small_boxes = limit_size(all_boxes)
        ssresults = nms(small_boxes, NMS_VAL)

        del all_boxes
        del small_boxes

        imout = image.copy()
        counter = 0
        falsecounter = 0
        fflag = 1 # 0 if using selective segmentation to detect
positive examples
        bflag = 0
        for e,result in enumerate(ssresults):
            hit = False
            for gtval in gtvalues:
                x,y,w,h = result
                iou = get_iou(gtval,{"x1":x,"x2":x+w,"y1":y,"y2":y+h})
                if iou > 0.1:
                    hit = True
            if not hit:
                if falsecounter >= 20:
                    break
                timage = imout[y:y+h,x:x+w]
                resized = cv2.resize(timage, (128,128), interpolation =
cv2.INTER_AREA)
                new_file = "im_"+str(e)+"_"+str(falsecounter)+".jpg"
                if i.startswith("c11") or i.startswith("c12") or
                   i.startswith("c13") or i.startswith("c14"): # test
set
                    test_images.append(resized)
                    test_labels.append(0)
                    write_path = "D:/Datasets/Strawberry
2019/Snippets/test/negative/"
                else:
                    train_images.append(resized)
                    train_labels.append(0)
                    write_path = "D:/Datasets/Strawberry
2019/Snippets/training/negative/"
                cv2.imwrite(write_path+new_file,resized)
                falsecounter += 1

            if fflag == 1 and bflag == 1:
                break
        del ssresults
```

```
    except Exception as e:
        print(repr(e))
        print(e)
        print("error in "+filename)
        continue


# In[ ]:


add_strawberry_snippets("D:/Datasets/Strawberry 2018/Data/White", 1)
add_strawberry_snippets("D:/Datasets/Strawberry 2018/Data/Red", 0)
add_strawberry_snippets("D:/Datasets/Strawberry
2018/background_crops/background_crops", 0)
add_strawberry_snippets("D:/Datasets/Strawberry
2018/bud_crops/bud_crops", 0)
add_strawberry_snippets("D:/Datasets/Strawberry 2019/False_Positives",
0, 10000, 29000)


# In[ ]:


add_strawberry_snippets("D:/Datasets/Strawberry
2019/Snippets/test/positive", 1, True)
add_strawberry_snippets("D:/Datasets/Strawberry
2019/Snippets/test/negative", 0, True)
add_strawberry_snippets("D:/Datasets/Strawberry
2019/Snippets/training/positive", 1, False)
add_strawberry_snippets("D:/Datasets/Strawberry
2019/Snippets/training/negative", 0, False)


# In[ ]:


X_new = np.array(train_images)
y_new = np.array(train_labels)
X_test = np.array(test_images)
y_test_new = np.array(test_labels)


# In[ ]:
```

```
X_test.shape


# In[ ]:


from tensorflow.keras import Model
from tensorflow.keras import optimizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# from tensorflow.keras.applications.vgg16 import VGG16
# from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2
# from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.resnet import ResNet152


# In[ ]:


from tensorflow.keras.layers import Input
mobile_model = ResNet101(weights='imagenet', include_top=False,
input_tensor=Input(shape=(128, 128, 3)))
mobile_model.summary()


# In[ ]:


# creating new head for 128x128 blob
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout

headModel = mobile_model.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(128, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)
```

```
# In[ ]:


# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layers in (mobile_model.layers):
    print(layers)
    layers.trainable = False


# In[ ]:


X= mobile_model.layers[-2].output


# In[ ]:


predictions = Dense(2, activation="softmax")(X)


# In[ ]:


model_final = Model(inputs = mobile_model.input, outputs = headModel)


# In[ ]:


from tensorflow.keras import optimizers
opt = optimizers.Adam(learning_rate=0.0001)


# In[ ]:


model_final.compile(loss = keras.losses.categorical_crossentropy,
optimizer = opt, metrics=["accuracy"])


# In[ ]:
```

```
model_final.summary()


# In[ ]:


from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer


# In[ ]:


class MyLabelBinarizer(LabelBinarizer):
    def transform(self, y):
        Y = super().transform(y)
        if self.y_type_ == 'binary':
            return np.hstack((Y, 1-Y))
        else:
            return Y
    def inverse_transform(self, Y, threshold=None):
        if self.y_type_ == 'binary':
            return super().inverse_transform(Y[:, 0], threshold)
        else:
            return super().inverse_transform(Y, threshold)


# In[ ]:


lenc = MyLabelBinarizer()
Y = lenc.fit_transform(y_new)
lenc2 = MyLabelBinarizer()
y_test = lenc2.fit_transform(y_test_new)


# In[ ]:


X_train, X_val, y_train, y_val =
train_test_split(X_new,Y,test_size=0.10) # split remaining data into
```

```
train and val


# In[ ]:


print(X_train.shape,X_test.shape,X_val.shape,y_train.shape,y_test.shape
,y_val.shape)


# In[ ]:


import joblib

joblib.dump(X_train, "X_train.sav")

joblib.dump(X_test, "X_test.sav")

joblib.dump(X_val, "X_val.sav")

joblib.dump(y_train, "y_train.sav")

joblib.dump(y_test, "y_test.sav")

joblib.dump(y_val, "y_val.sav")


# In[ ]:


import joblib

X_train = joblib.load("X_train.sav")

X_test = joblib.load("X_test.sav")

y_train = joblib.load("y_train.sav")

y_test = joblib.load("y_test.sav")


# In[ ]:
```

```
X_val = joblib.load("X_val.sav")

y_val = joblib.load("y_val.sav")


# In[ ]:


print(X_train.shape,X_test.shape,X_val.shape,y_train.shape,y_test.shape
,y_val.shape)


# In[ ]:


tsdata = ImageDataGenerator(horizontal_flip=True, vertical_flip=True,
rotation_range=90)
testdata = tsdata.flow(x=X_test, y=y_test)
vldata = ImageDataGenerator(horizontal_flip=True, vertical_flip=True,
rotation_range=90)
valdata = tsdata.flow(x=X_val, y=y_val)
del X_val
del y_val
trdata = ImageDataGenerator(horizontal_flip=True, vertical_flip=True,
rotation_range=90)
traindata = trdata.flow(x=X_train, y=y_train)


# In[ ]:


from keras.callbacks import ModelCheckpoint, EarlyStopping


# In[ ]:


metric = 'accuracy'


# In[ ]:
```

```
checkpoint = ModelCheckpoint("ieeercnn_mobile_1.h5", monitor=metric,
verbose=1, save_best_only=True, save_weights_only=False, mode='auto',
period=1)
early = EarlyStopping(monitor=metric, min_delta=0, patience=100,
verbose=1, mode='auto')


# In[ ]:


my_batch_size = 32
my_steps_per_epoch = len(X_train)//my_batch_size
my_validation_steps = len(X_test)//my_batch_size
print(my_steps_per_epoch, my_validation_steps)


# In[ ]:


hist = model_final.fit(traindata, steps_per_epoch=my_steps_per_epoch,
epochs= 100, batch_size=my_batch_size, validation_data= valdata,
validation_steps=my_validation_steps, callbacks=[checkpoint,early])


# In[ ]:


import matplotlib.pyplot as plt
plt.plot(hist.history["accuracy"])
plt.plot(hist.history['val_accuracy'])
plt.title("model accuracy")
plt.ylabel("train accuracy")
plt.xlabel("val accuracy")
plt.legend(["train accuracy","val accuracy"])
plt.show()
plt.savefig('accuracy_chart.png')


# In[ ]:
```

```
# new plots
N = 100
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), hist.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), hist.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), hist.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), hist.history["val_accuracy"],
label="val_acc")
plt.title("Training Loss and Accuracy on Dataset")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="best")
plt.savefig('new_accuracy_chart.png')


# In[ ]:


print(len(X_train), len(X_test))


# In[ ]:


im = X_test[10]
plt.imshow(cv2.cvtColor(im, cv2.COLOR_BGR2RGB))


# In[ ]:


import time
t_start = time.time()

model_final.evaluate(X_test, y_test)

print("TEST TIME ", time.time() - t_start)


# In[ ]:
```

```
im = X_test[-37]
plt.imshow(cv2.cvtColor(im, cv2.COLOR_BGR2RGB))
img = np.expand_dims(im, axis=0)
out= model_final.predict(img)
if out[0][0] > 0.98:
    print("Green Strawberry")
else:
    print("Not Green Strawberry")


# In[ ]:


from keras.models import save_model

model_final.save('model_resnet152_full_aug')


# In[ ]:


# LOAD AND EVALUATE MODEL


# In[ ]:


import os,cv2,keras
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import torch

path = "D:/Datasets/Strawberry 2019/Images"
annot = "D:/Datasets/Strawberry 2019/Annotations"

downscale_factor = 6
IOU_VAL = 0.3
NMS_VAL = 0.3
correct_prediction = 0.725

def ResizeImage(im, factor):
    width = int(im.shape[1] / factor)
    height = int(im.shape[0] / factor)

    # dsize
    dsize = (width, height)

    # resize image
    return cv2.resize(im, dsize)

def get_iou(bb1, bb2):
    assert bb1['x1'] < bb1['x2']
    assert bb1['y1'] < bb1['y2']
    assert bb2['x1'] < bb2['x2']
    assert bb2['y1'] < bb2['y2']

    x_left = max(bb1['x1'], bb2['x1'])
    y_top = max(bb1['y1'], bb2['y1'])
    x_right = min(bb1['x2'], bb2['x2'])
    y_bottom = min(bb1['y2'], bb2['y2'])

    if x_right < x_left or y_bottom < y_top:
        return 0.0

    intersection_area = (x_right - x_left) * (y_bottom - y_top)

    bb1_area = (bb1['x2'] - bb1['x1']) * (bb1['y2'] - bb1['y1'])
    bb2_area = (bb2['x2'] - bb2['x1']) * (bb2['y2'] - bb2['y1'])

    iou = intersection_area / float(bb1_area + bb2_area -
intersection_area)
    assert iou >= 0.0
    assert iou <= 1.0
    return iou


# In[ ]:


# uses downscaling factor to get fullscale box coordinates
def get_full_coords(box):
    x_full = box[0] * downscale_factor
    y_full = box[1] * downscale_factor
    w_full = box[2] * downscale_factor
    h_full = box[3] * downscale_factor
    return [x_full, y_full, w_full, h_full]


# In[ ]:


def nms(boxes, threshold=.7):

    boxes = sorted(boxes, key=lambda boxes: boxes[2],
            reverse=True)

    new_boxes=[]

    new_boxes.append(boxes[0])

    del boxes[0]

    for index, box in enumerate(boxes):
        for new_box in new_boxes:
            a =
{"x1":box[0],"x2":box[0]+box[2],"y1":box[1],"y2":box[1]+box[3]}
            b =
{"x1":new_box[0],"x2":new_box[0]+new_box[2],"y1":new_box[1],"y2":new_bo
x[1]+new_box[3]}
            if get_iou(a, b) > threshold:
                del boxes[index]
                break
        else:
            new_boxes.append(box)
            del boxes[index]
    return new_boxes


# In[ ]:


def nms_pytorch(boxes, thresh_iou):
    """
    Apply non-maximum suppression to avoid detecting too many
    overlapping bounding boxes for a given object.
    Args:
        boxes: (tensor) The location preds for the image
```

```
            along with the class predscores, Shape: [num_boxes,5].
        thresh_iou: (float) The overlap thresh for suppressing
unnecessary boxes.
    Returns:
        A list of filtered boxes, Shape: [ , 5]
    """
    P = torch.Tensor(boxes)

    # we extract coordinates for every
    # prediction box present in P
    x1 = P[:, 0]
    y1 = P[:, 1]
    width = P[:, 2]
    height = P[:, 3]
    x2 = x1 + width
    y2 = y1 + height

    # we extract the confidence scores as well
    scores = P[:, 4]

    # calculate area of every block in P
    areas = (x2 - x1) * (y2 - y1)

    # sort the prediction boxes in P
    # according to their confidence scores
    order = scores.argsort()

    # initialise an empty list for
    # filtered prediction boxes
    keep = []

    while len(order) > 0:

        # extract the index of the
        # prediction with highest score
        # we call this prediction S
        idx = order[-1]

        if scores[idx].item() <= correct_prediction:
            break

        # push S in filtered predictions list
```

```
keep.append([int(x1[idx].item()),int(y1[idx].item()),int(width[idx].ite
m()),int(height[idx].item()),scores[idx].item()])

        # remove S from P
        order = order[:-1]

        # sanity check
        if len(order) == 0:
            break

        # select coordinates of BBoxes according to
        # the indices in order
        xx1 = torch.index_select(x1,dim = 0, index = order)
        xx2 = torch.index_select(x2,dim = 0, index = order)
        yy1 = torch.index_select(y1,dim = 0, index = order)
        yy2 = torch.index_select(y2,dim = 0, index = order)

        # find the coordinates of the intersection boxes
        xx1 = torch.max(xx1, x1[idx])
        yy1 = torch.max(yy1, y1[idx])
        xx2 = torch.min(xx2, x2[idx])
        yy2 = torch.min(yy2, y2[idx])

        # find height and width of the intersection boxes
        w = xx2 - xx1
        h = yy2 - yy1

        # take max with 0.0 to avoid negative w and h
        # due to non-overlapping boxes
        w = torch.clamp(w, min=0.0)
        h = torch.clamp(h, min=0.0)

        # find the intersection area
        inter = w*h

        # find the areas of BBoxes according the indices in order
        rem_areas = torch.index_select(areas, dim = 0, index = order)

        # find the union of every prediction T in P
        # with the prediction S
        # Note that areas[idx] represents area of S
        union = (rem_areas - inter) + areas[idx]
        # find the IoU of every prediction in P with S
        IoU = inter / union

        # keep the boxes with IoU less than thresh_iou
        mask = IoU < thresh_iou
        order = order[mask]

    return keep


# In[ ]:


def sliding_window(im, box_wid, inc):
    swresults = []
    height, width, c = im.shape
    x = 0
    y = 0
    while y + box_wid < height:
        while x + box_wid < width:
            swresults.append((x, y, box_wid, box_wid))
            x += inc
        x = 0
        y += inc

    return swresults


# In[ ]:


def limit_size(boxes):
    max_area = 150**2 // (downscale_factor**2)#(150 * (1/IOU_VAL))**2
// (downscale_factor**2) # max area defined by scale and IOU

    min_area = 50**2 // (downscale_factor**2)#(50 * (IOU_VAL/1))**2 //
(downscale_factor**2)

    boxes = sorted(boxes, key=lambda boxes: boxes[2],
            reverse=True)

    new_boxes=[]
    for index, box in enumerate(boxes):
        if min_area < box[2]*box[3] and box[2]*box[3] < max_area: #
area range
            if box[2]*5 > box[3] and box[3]*5 > box[2]: # no elongated
rectangles
                new_boxes.append(box)

    del boxes
    return new_boxes


# In[ ]:


from keras.models import load_model

model_final = load_model('model_resnet50_full_aug')


# In[ ]:


cv2.setUseOptimized(True);
ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()


# In[ ]:


# Selective Segmentation
# sees if SS finds all berries

import time

t_start = time.time()

true_positives = 0
false_positives = 0
false_negatives = 0

im_names = [i for i in os.listdir(path)]

for e,i in enumerate(im_names):
    if i.startswith("c11") or i.startswith("c12") or
i.startswith("c13") or i.startswith("c14"):
        t = time.time()
        start_image = cv2.imread(os.path.join(path,i))
        img = ResizeImage(start_image, downscale_factor)
        ss.setBaseImage(img)
        ss.switchToSelectiveSearchFast()
        s =
cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategyFill
()
        ss.addStrategy(s)
        t = time.time()

        all_boxes = ss.process()
        print("num of all boxes ", len(all_boxes))
        small_boxes = limit_size(all_boxes)
        print("num of small boxes ", len(small_boxes))
        ssresults = small_boxes#nms(small_boxes, 0.8)
        print("num of non_max_sup ", len(ssresults))
        del all_boxes
        del small_boxes

        # SLIDING WINDOW
#         ssresults = sliding_window(img, 120//downscale_factor, 10)
#         ssresults.extend(sliding_window(img, 100, 10))
#         ssresults.extend(sliding_window(img, 125, 15))

        print("process time ", time.time() - t)
        t = time.time()

        num_berries = 0
        berry_hits = []
        df = None
        annot_name = i.split(".")[0] + ".csv"
        file_name = os.path.join(annot,annot_name)
        if os.path.exists(file_name):
            df = pd.read_csv(file_name)
            num_berries = int(df.count()) # number of berries in image
            print("num_berries", num_berries, df)
            berry_hits = np.repeat(0, num_berries) # array to tell
which berries are correctly classified
        else:
            print("no annot file")
        small_bbs = 0
        for e,result in enumerate(ssresults):
            if type(df) is type(None):
                continue
            x,y,w,h = result
            j = 0
            for num, row in enumerate(df.iterrows()): # compare each
result with each real berry
                x1 = int(row[1][0].split(" ")[0]) // downscale_factor
                y1 = int(row[1][0].split(" ")[1]) // downscale_factor
                x2 = int(row[1][0].split(" ")[2]) // downscale_factor
                y2 = int(row[1][0].split(" ")[3]) // downscale_factor
                iou =
get_iou({"x1":x1,"x2":x2,"y1":y1,"y2":y2},{"x1":x,"x2":x+w,"y1":y,"y2":
y+h})
                if iou > IOU_VAL: # correct detection
                    if num_berries > 0 and berry_hits[j] == 1:
                        continue

                    berry_hits[j] = 1
                j += 1

        print("ssresults size e ", e)
        for j in range(0, num_berries):
            hit = berry_hits[j]
            if hit == 0:
                false_negatives += 1
            elif hit == 1:
                true_positives += 1

print("TOTAL TIME ", time.time() - t_start)
del ssresults
del berry_hits


# In[ ]:


print(downscale_factor)
print(true_positives, false_negatives, false_positives)
```

```
# In[ ]:


# detect green strawberries
import time

t_start = time.time()

true_positives = 0
false_positives = 0
false_negatives = 0

downscale_factor = 6
IOU_VAL = 0.3
NMS_VAL = 0.3
correct_prediction = 0.725

im_names = [i for i in os.listdir(path)]

for e,i in enumerate(im_names):
    if i.startswith("c11") or i.startswith("c12") or 
i.startswith("c13") or i.startswith("c14"):
        start_image = cv2.imread(os.path.join(path,i))
        downscale_im = ResizeImage(start_image, downscale_factor)
        ss.setBaseImage(downscale_im)
        ss.switchToSelectiveSearchFast()
        s = 
cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategyFill
()
        ss.addStrategy(s)
        all_boxes = ss.process()
        ssresults = limit_size(all_boxes)

        del all_boxes
        imout = start_image.copy()

        num_berries = 0
        berry_hits = []
        df = None
        annot_name = i.rsplit('.', 1)[0] + ".csv"
        file_name = os.path.join(annot,annot_name)
        if os.path.exists(file_name):
            df = pd.read_csv(file_name)
            num_berries = int(df.count()) # number of berries in image
            berry_hits = np.repeat(0, num_berries) # array to tell
which berries are correctly classified
            berry_bbs = np.repeat(None, num_berries)

        if num_berries == 0:
            continue

        print("num_berries ",num_berries)
        print("df ",df)

        P = []
        for e,result in enumerate(ssresults):
            x,y,w,h = result
            x_full,y_full,w_full,h_full = get_full_coords(result)
            timage =
start_image[y_full:y_full+h_full,x_full:x_full+w_full]
            resized = cv2.resize(timage, (128,128), interpolation =
cv2.INTER_AREA)
            im = np.expand_dims(resized, axis=0)
            out = model_final.predict(im)
            P.append((x,y,w,h,out[0][0]))

        del ssresults
        boxes = nms_pytorch(P, NMS_VAL)
        print("total boxes ", len(P), " nms ", len(boxes))
        del P

        gtvalues=[]
        if type(df) is not type(None):
            for row in df.iterrows():
                x1 = int(row[1][0].split(" ")[0]) // downscale_factor
                y1 = int(row[1][0].split(" ")[1]) // downscale_factor
                x2 = int(row[1][0].split(" ")[2]) // downscale_factor
                y2 = int(row[1][0].split(" ")[3]) // downscale_factor
                gtvalues.append({"x1":x1,"x2":x2,"y1":y1,"y2":y2})

        for e,result in enumerate(boxes):
            x,y,w,h,score = result
            x_full,y_full,w_full,h_full = get_full_coords(result)
            timage =
start_image[y_full:y_full+h_full,x_full:x_full+w_full]
            resized = cv2.resize(timage, (128,128), interpolation =
cv2.INTER_AREA)

            if score > correct_prediction: # predicting a green
strawberry
                hit = False
                save_image = True
                if num_berries > 0:
                    for j,gtval in enumerate(gtvalues):
                        iou =
get_iou(gtval,{"x1":x,"x2":x+w,"y1":y,"y2":y+h})
                        if iou > IOU_VAL: # correct detection
                            hit = True
                            if berry_hits[j] == 0 or score >
berry_bbs[j][4]:
                                berry_hits[j] = 1
                                berry_bbs[j] = (x,y,w,h,score)
                                break
                if not hit: # false positive
                    false_positives += 1
                    new_file =
"resized_"+str(130200+false_positives)+".jpg"
                    write_path = "D:/Datasets/Strawberry
2019/False_Positives/"
                    cv2.imwrite(write_path+new_file,resized)
                    cv2.rectangle(imout, (x_full, y_full),
(x_full+w_full, y_full+h_full), (0, 0, 255), 3) # draw red bb

        for j in range(0, num_berries):
            hit = berry_hits[j]
            if hit == 0:
                false_negatives += 1
            elif hit == 1:
                true_positives += 1
                x,y,w,h = get_full_coords(berry_bbs[j])
                score = berry_bbs[j][4]
                print("true positive score ", score)
                cv2.rectangle(imout, (x, y), (x+w, y+h), (255, 0, 0),
3) # draw blue bb

        plt.figure()
        plt.imshow(cv2.cvtColor(imout, cv2.COLOR_BGR2RGB))
        cv2.imwrite(i.rsplit('.', 1)[0] + '.png', imout)
```

```
print("TOTAL TIME ", time.time() - t_start)
del berry_hits
del berry_bbs
del boxes


# In[ ]:


print(true_positives, false_negatives, false_positives)
```