DETECTION OF RED-COCKADED WOODPECKER HABITATS USING YOLO ALGORITHMS

____

A Thesis

Presented to

The Faculty of the Department of Computer Science

Sam Houston State University

____

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

____

by

Emerson Alfred de Lemmus II

August 2022

DETECTION OF RED-COCKADED WOODPECKER HABITATS USING YOLO ALGORITHMS

by

Emerson Alfred de Lemmus II

APPROVED:

DEDICATION

I dedicate this work to my family, who supported me throughout my education, and to the Department of Computer Science, which provided me with a second family throughout my years at Sam Houston State University.

ABSTRACT

de Lemmus, Emerson Albert. *Detection of red-cockaded woodpecker habitats using YOLO algorithms*. Master of Science (Computing and Data Science), August, 2022, Sam Houston State University, Huntsville, Texas.

Habitat and population monitoring are crucial for the preservation of endangered species. However, gathering habitat data may be a hazardous and laborious task. As a result, wildlife ecologists increasingly turn to remote sensing and automation to collect large-scale ecological data on a given species. Particularly, the red-cockaded woodpecker (RCW) is a species endemic to the southeastern United States. Endangered since 1973, wildlife biologists have performed pedestrian surveys to assess the status of the species.

Through close interdisciplinary collaboration with ecologists, this work conducts a pilot study that automatically detects potential habitats of RCW. The dataset of 978 images was collected by a team of wildlife ecologists from Raven Environmental Inc. using unmanned aerial vehicles (UAVs). RCW habitat imagery is quite unique and is not available in the public domain, thus it is considered novel image data. The primary goal of this research is to assess the RCW habitat detection performance by You Only Look Once (YOLO) object detection algorithms. As for the demanding computing requirements of YOLO algorithms, only two small models, YOLOv4-tiny and YOLOv5n, are employed and assessed for this study. Specifically, the best hyperparameter values are identified per each model that maximize the precision performance for the training data. YOLOv4-tiny reached a training mAP (minimum Average Precision) of 0.96 (i.e., 96%) and a testing accuracy of 0.85 (i.e., 85%), while YOLOv5n achieved a training mAP of 0.78 (i.e., 78%) and a testing accuracy of 0.82

(82%). Overall, combining the inference results of both models achieved a 100% detection of de facto habitats.

This study realizes a real-time platform that integrates computer vision with domain knowledge and identifies potential habitats from large-scale image data. Therefore, the deployment of the study on wildlife ecosystems will significantly assist wildlife biologists in saving personnel hours through real-time detection of potential habitats and accelerating proactive field validating for the preservation of RCW.

KEY WORDS: Red-Cockaded woodpecker, Wildlife management, Remote sensing, Unmanned aerial vehicle, Object detection, YOLO algorithm

TABLE OF CONTENTS

Page

DEDICATION III

ABSTRACT IV

TABLE OF CONTENTS VI

LIST OF TABLES VIII

LIST OF FIGURES IX

CHAPTER I: INTRODUCTION 1

Problem Context 1

Red-Cockaded Woodpecker 2

Document Outline 4

CHAPTER II: RELATED RESEARCH 6

Computer Vision and Ecology 6

CHAPTER III: OBJECT RECOGNITION 10

Traditional Computer Vision 12

Modern Computer Vision 16

You Only Look Once 21

CHAPTER IV: PROPOSED APPROACH 27

Dataset Description 28

Hardware Tools 36

Software Tools 37

CHAPTER V: EXPERIMENTAL STUDY 39

Discussion 39

Training 40

Results 47

YOLOv5n 63

CHAPTER VI: CONCLUSION AND FUTURE WORK 70

Data 70

Demanding Computing Hardware 71

Model Ensembling 71

REFERENCES 72

APPENDIX 81

VITA 131

LIST OF TABLES

Table **Page**

1 Existing Works Combining Computer Vision and Ecology 9

2 Condensed model summary of YOLO 24

3 534 Images Taken at 76m Elevation 31

4 444 Images Taken at 46m Elevation 33

5 Computer Hardware Tools 36

6 Drones Deployed 36

7 Software tools for YOLOv4 37

8 Software tools used for YOLOv5 38

9 Annotation box statistics 43

10 Detailed performance of YOLOv4-tiny 60

11 YOLOv5n confusion matrix results 67

LIST OF FIGURES

Figure **Page**

1 Historic Red-cockaded Woodpecker Habitat Distribution 3

2 Relationship among Subfields in Artificial Intellence 6

3 Illustrating the Four Typical Computer Vision Tasks 8

4 Classification of Object Recognition Tasks 11

5 The Elementary Process of an Image Classifier 12

6 Traditional Computer Vision Approaches 14

7 Multi-layer Perceptron Standard Architecture 15

8 Flattening of a 2D Image to a 1D Input Vector 16

9 Modern Computer Vision Approaches 18

10 Basic Component of Convolutional Networks 19

11 Difference between single-stage detectors and two-stage detectors 20

12 A Simplified View of You Only Look Once Model Processing 21

13 Summary of the Detection Process Used by YOLO 22

14 History of the YOLO Model Famil 23

15 Montgomery County, Texas 29

16 Map Detailing all Locations where the Dataset was Gathered 30

17 Annotation Heatmap of 534 Images Taken at 76m Elevation 32

18 The Annotation Heatmap of 444 images Taken at 46m Elevation 34

19 Sample Images Demonstrating Altitude Differences 35

20 Sample Red-cockaded Woodpecker Habitat Signs 41

21 Illustration of a Small-size RCW Pine Resin Sign Compared to the Entire Image 42

22 Yolo Rescaling of an Image to Match Specified Input 44

23 Image Tiling in Practice 46

24 Scenario 1 using 534 images taken at 76m elevation 49

25 Scenario 2 using 534 images taken at 76m elevation 50

26 Scenario 3 using 534 images taken at 76m elevation 51

27 Scenario 4 using 444 images taken at 46m elevation 52

28 Proper annotation to capture pine resin cavity markers 53

29 Improper annotation featuring incorrectly labeled objects 54

30 Scenario 5 iteration merging batch 1 and batch 2 datasets 55

31 Optimal settings after trial and error 57

32 Optimal performance using YOLOv4-tiny with the merged dataset 58

33 Confusion matrix of YOLOv4-tiny 59

34 Sample inference with 76m altitude test data 61

35 Sample inferencing with 46m altitude test data 62

36 Performance of YOLOv5n with batch 1 data (top) and batch 2 data (bottom) 64

37 Distribution of class instances in YOLOv5 output result 65

38 YOLOv5n confusion matrix 66

39 Sample YOLOv4-tiny testing inference compared with YOLOv5n 69

CHAPTER I

Introduction

Problem Context

Monitoring the habitats and populations of endangered species is critical to the conservation and measuring conservation programs' efficacy. Birds, in particular, have a wide distribution, excellent mobility, and high sensitivity to environmental changes; therefore, they have become significant groups for monitoring (Gregory & van Strien, 2010). However, surveying a species across its geographic range can be complex and resource-intensive (Neate-Clegg, Horns, Aytekin, & Şekercioğlu, 2020). To reduce economic cost, labor, and logistics, conservation efforts are increasingly automating the sampling of natural environments (Pimm, Alibhai, Dehgan, & Giri, 2015). Collecting habitat data with unmanned aerial vehicles (UAVs) provides a minimally invasive approach to obtaining relative habitat densities and estimating population trends over time. These devices can survey large, forested areas that humans cannot easily access or stay in for long periods of time.

Additionally, captured imagery can be stored in convenient formats for analysis. By analyzing the habitat elements in these images, conservationists can accurately record nesting locations and cluster patterns (Carrascal, Galván, Sánchez-Oliver, & Rey Benayas, 2013). Although UAVs are increasingly common devices and help collect large-scale data, the ability to process and analyze huge amounts of images remains challenging when turning data into information on animal presence, nesting patterns, and behavior (Weinstein, 2017). To address challenges in large dataset analysis for the preservation of species, ecologists are frequently turning to automation for accurate and efficient processing of large image datasets. This work aims to propose, describe, and implement an experimental system using the latest advances in computing capability to analyze large UAV-generated datasets to identify red-cockaded woodpecker habitats.

Red-Cockaded Woodpecker

Current Status

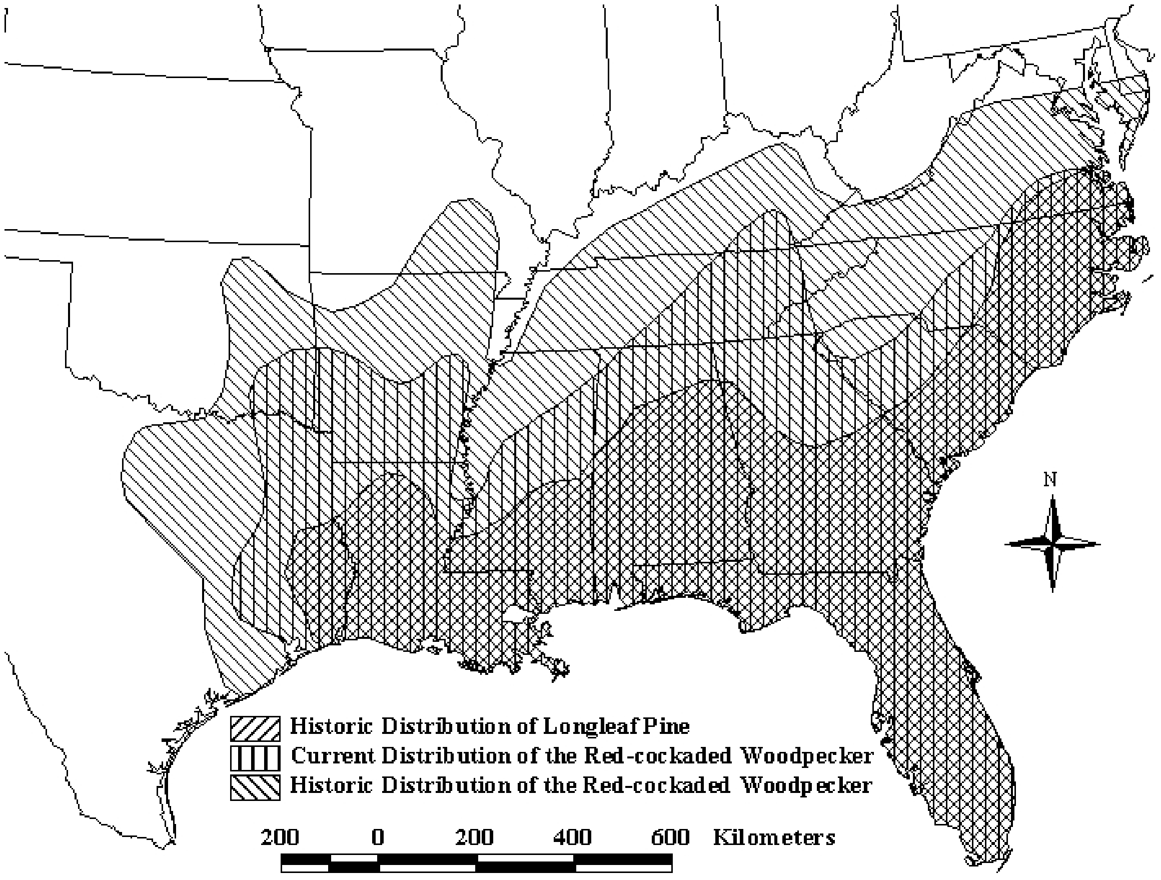
The primary subject of this study is the red-cockaded woodpecker (RCW). RCWs are a species endemic across the southeastern United States (U.S. Fish and Wildlife Service, 2003). RCWs were listed as endangered in 1970 and received federal protection with the passage of the Endangered Species Act in 1973 (U.S. Fish and Wildlife Service, 2003). However, by the time of its endangered classification, RCWs had declined to fewer than 10,000 individuals in scattered and isolated habitats and this figure represents less than 3 percent estimated abundance at the time of European colonialization (Jackson, 1971).

Habitat Description

According to the U.S. Fish and Wildlife Service, the staggering decline was caused by an almost complete loss of habitats. Southern longleaf pine once dominated the southeastern United States and may have totaled over 800 million hectares during pre-European colonialization; however, today, less than 1.2 million hectares remains (Conner et al. 2001, Landers et al. 1995). Figure 1 visualizes the historical distribution of longleaf pine and the historical and present-day distribution of red-cockaded woodpeckers (Butler, 2001).

Figure 1

Historic Red-cockaded Woodpecker Habitat Distribution


▨ Historic Distribution of Longleaf Pine
▥ Current Distribution of the Red-cockaded Woodpecker
▧ Historic Distribution of the Red-cockaded Woodpecker
200 0 200 400 600 Kilometers
N

Note. Historical distribution of longleaf pine and historical and current distribution of red-cockaded woodpecker (Butler, 2001). Courtesy of Dr. Mathew J. Butler.

The original pine ecosystem was lost due to intense logging during the 19th – 20th century, exploitation of pine resins, and grazing done by livestock and free-ranging hogs (U.S. Fish and Wildlife Service, 2003). RCWs are habitat specialists, strongly tied to old-growth pine forests that burn frequently (Cornell Labs, 2019). RCWs exploit the ability of live pines to produce large amounts of pine resin by creating cavity wounds known as resin wells (U.S. Fish and Wildlife Service, 2003). Resin is a natural, effective barrier against climbing snakes and is best created by old-growth longleaf pine. Regrowth of these habitats has been severely restricted since the pre-colonial era due to free-ranging hogs and fire suppression (U.S. Fish and Wildlife Service, 2003).

Threats

The primary threat to RCW viability is a lack of suitable habitat. On public and private lands, the quantity and quality of RCW habitats are affected by past and present policies on fire suppression and silvicultural practices. These policies and procedures have led to an insufficient number of cavities, net loss of suitable cavity trees, habitat fragmentation leading to isolation and genetic issues, lack of suitable foraging habitats, and risk of extinction through genetic, environmental, and catastrophic events. For these reasons, RCWs are now among the most endangered species on earth (Simberloff 1993, Ware et al. 1993). RCW population monitoring is a critical component of their conservation and recovery. Traditional RCW surveying is done by personnel experienced in managing and monitoring the species. Potential nesting habitat is identified and surveyed by visually inspecting all medium-size and large pine trees for cavity excavation evidence (Environmental Analysis Unit, 2018). If cavities are located, more intense land surveying is performed within 457 meters of each cavity (Environmental Analysis Unit, 2018). This type of land surveying is costly, inefficient, and dangerous to personnel.

Document Outline

The rest of the thesis is organized as follows. In Chapter II, the related work of various interdisciplinary studies is reviewed. Chapter III is an in-depth discussion and review of object recognition, including traditional and modern computer vision approaches, and You Only Look Once (YOLO) is introduced. In Chapter IV, the proposed method is discussed, where a detailed rationale of available techniques is provided, and the available dataset and tools are thoroughly described. In Chapter V, the experimental results are detailed, along with optimal settings and configurations. Finally, Chapter VI concludes with summaries and suggestions for possible improvements.

CHAPTER II

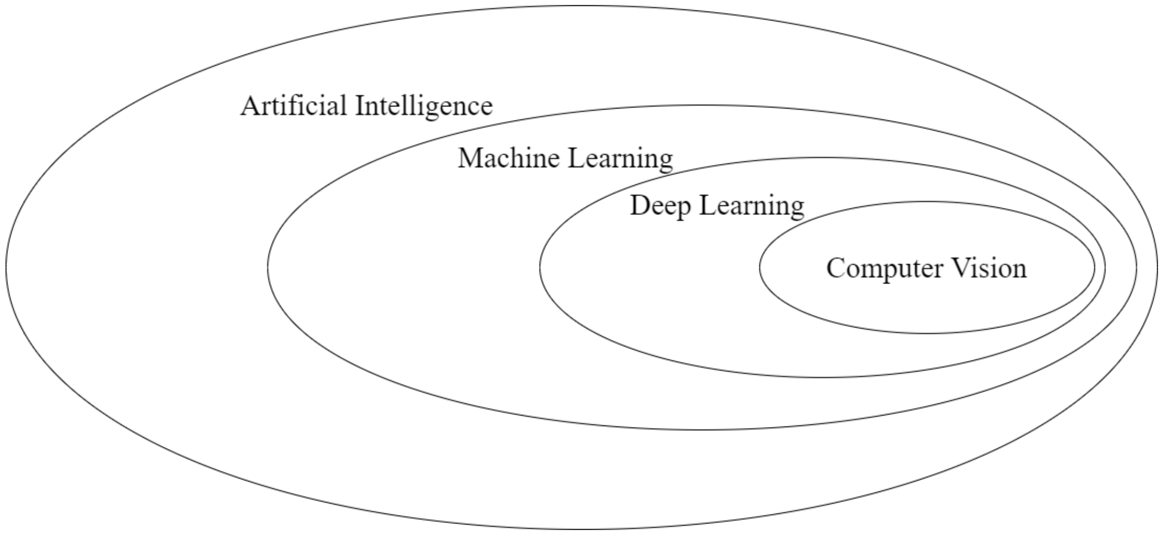
Related Research

Computer Vision and Ecology

Computer vision (CV) is a subset field of machine learning, as summarized in Figure 2, that is focused on mimicking the natural way human and animal visual systems work (Krohn, Beyleveld, & Bassens, 2019). Computer vision was dominated by traditional machine learning algorithms before the advent of deep learning (DL) innovations and is now considered a unique DL area of study that uses pixel values to infer image content (LeCun, Bengio, & Hinton, 2015). This allows computing systems to process and identify objects in images and videos faster and more accurately than humans can (Mahajlovic, 2019). Using this cutting-edge technology, CV can assist conservation efforts by increasing the scope, duration, and repeatability of image-based ecological studies through automated image analysis (Weinstein, 2017).

Figure 2

Relationship among Subfields in Artificial Intellence



Note. Figure 2 visualizes the relationship among artificial intelligence, machine learning, deep learning and computer vision. Redrawn from (Mellit et al., 2020).

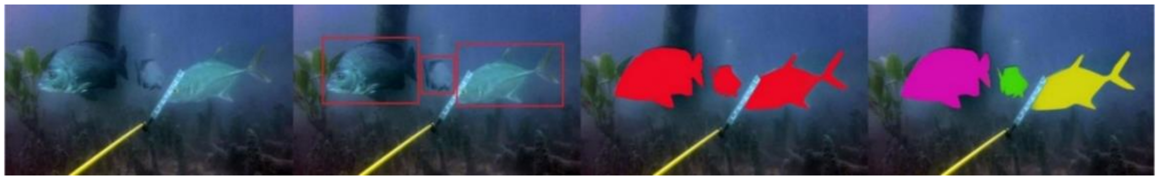
Particularly, ecologists choose CV algorithms as content analysis methodologies due to the explosive volume of data generated by deployed UAVs rapidly surpasses the capacity of human video viewers, making video analysis prohibitively expensive (Konovalov et al., 2019). Humans may be good at inferring objects in an image by manual visual inspection or have excellent knowledge on identifying habitats; however, when confronted with thousands of data points, it is difficult to find the time, organization, and concentration to validate each image manually (Berg et al., 2014).

Computer vision has been highly developed and frequently updated with new models. Therefore, CV methodologies have been successfully adapted for numerous ecological applications, including species identification, detection, and enumeration of animals within an image and description of animal coloration, patterns, and relative sizes (Weinstein, 2017). However, to date, CV has never been applied to the detection of RCW habitats; thus, no direct references to the subject are available. Furthermore, no benchmark image datasets on RCW habitats are available in the public domain. Accordingly, this project addresses the need to collect a benchmark image dataset on RCW habitats and offers a timely opportunity to create a CV application to detect RCW habitats.

Ecologists often concentrate on three common tasks for ecological computer vision: physical description of species such as coloration, patterns, background relationships, and relative sizes, enumeration of animals in a target area, and identification of species (Weinstein, 2017). Figure 3 demonstrates the typical CV tasks related to ecology.

Figure 3

Illustrating the Four Typical Computer Vision Tasks



Note. From left: Image Classification (i.e., is there a fish in this image, or what type of fish is in the image?), Object Detection/Localization (i.e., where are the fish in the image located, if at all?), Semantic Segmentation (i.e., what is the relationship between the fish and their surroundings?), Instance Segmentation (i.e., how many instances of fish are in this image?). From (Saleh, Sheaves, & Azghadi, 2022).

For example, to correctly outline a flying bird, computer vision algorithms often search for important pixels within images. (Atanbori, Duan, Murray, Appiah, & Dickinson, 2016) was successfully able to detect distant birds in flight by looking at pixels where wings intersect in the sky with an average accuracy of 0.89 using a modified SVM classifier. Beyond detecting animals in a specific environment, (Mahndahar et al., 2018) demonstrated the ability to classify potential coastal habitats for both marine and terrestrial animals in satellite imagery using AlexNet with an average accuracy of .95. (Wilber et al., 2013) implemented a variety of computer vision models and modified SVMs to classify seven desert species given a large image. Table 1 lists previous relevant research with applications of computer vision to animal ecology.

Table 1

Existing Works Combining Computer Vision and Ecology

Reference	Technique	Model	Training Images	Habitat	Taxa	Species	AP%
Wilber et al., 2013	Traditional ML	V1-like	5,362	Desert	Mammals, Reptiles	7	76.4
Yu et al., 2013	Traditional ML	SVM	22,533	Tropical Rainforest, Temperate Forest	Mammals	18	83.8
Atanbori et al., 2016	Traditional ML	Normal Bayes Classifier, SVM	-	-	Birds	7	89.0
Feng et al., 2016	Traditional ML	Semantic Related Visual	4,530	-	Moths	50	53.12
Jin & Liang, 2017	DL	Custom CNN	27,000	Tropical	Fish	23	96.27
Mahndahar et al., 2018	DL	AlexNet	4096	Coastal	-	-	95.12
Tamou et al., 2018	DL	AlexNet	27,000	Tropical	Fish	23	99.45
Miao et al., 2019	DL	VGG19, ResNet-50	111,467	Savannah	Varied	21	87.5
Iqbal et al., 2021	DL	AlexNet derived	13,200 + 20 videos	Tropical	Fish	6	90.48

Note. Table 1 summarizes a collection of existing works combining computer vision and ecology. An important difference to notice between traditional ML and DL techniques is the improved performance, average precision (AP), with DL techniques. Adapted and expanded from Weinstein, 2017.

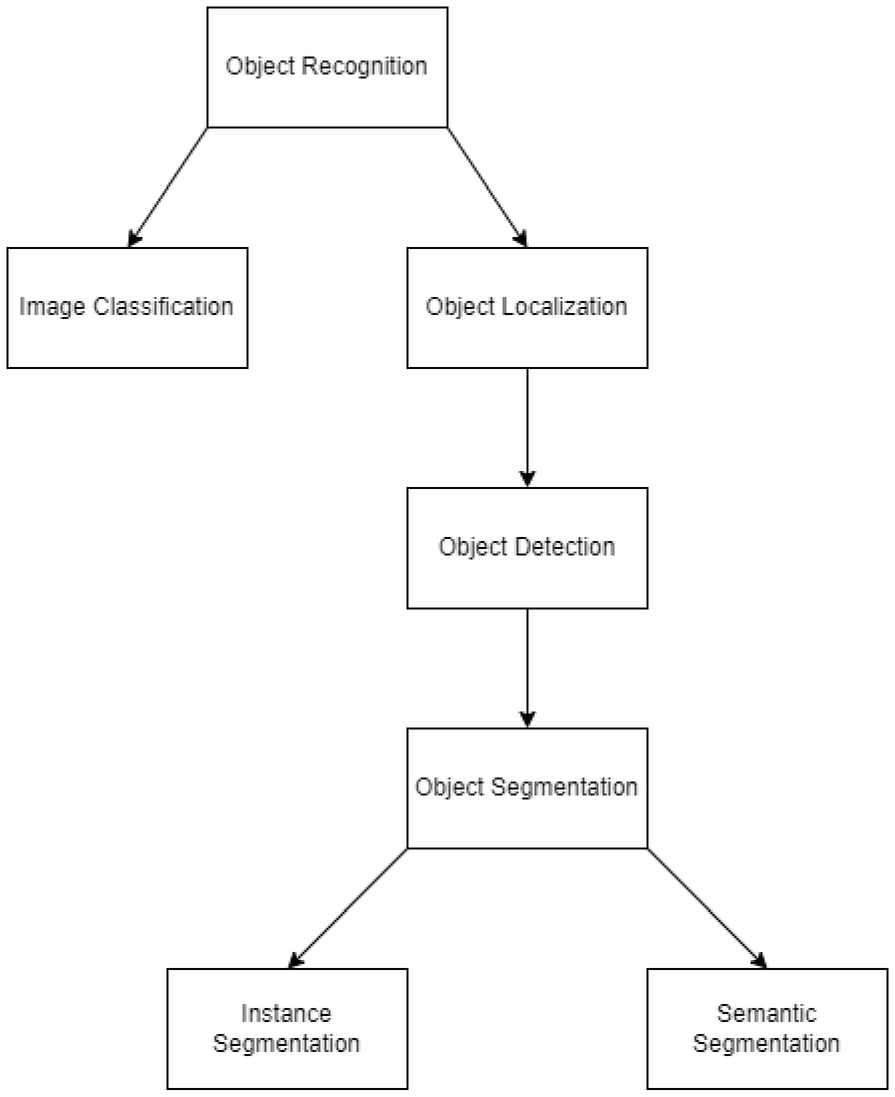
CHAPTER III

Object Recognition

Object recognition is a broad term to describe a collection of methodologies that involve identifying objects within digital images. Object recognition can be split into two distinct branches: image classification and object localization. Image classification predicts a class given a single input image with a single object within the image, see Figure 5. Object localization is the methodology used to locate the pixel presence of an object(s) in an image and denote the object's location with a drawn bounding box. Object localization can be further decomposed into object detection, where objects are drawn with bounding boxes and labels, and object segmentation, where instances are highlighted and separated from their background, see Figure 4. According to (Zhao et al., 2019), object recognition can be traced back to the 1940s, when (Pitts & McCulloch, 1947) first described an artificial neural mechanism that solved simple, general learning problems. (Hinton et al., 1986, Zhao et al., 2019) renewed interest in neural structures and has remained consistently popular since 2006. The emergence of large-scale annotated image datasets like ImageNet, the broader availability of high-performance computing, and significant improvements in neural network structure designs such as AlexNet, GoogLeNet, VGG, ResNet, and YOLO have attributed to huge impacts in object recognition.

Figure 4

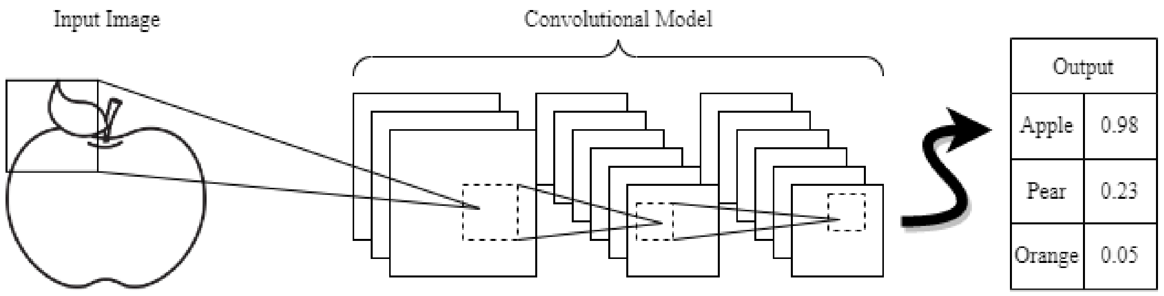
Classification of Object Recognition Tasks



Note. Object recognition is often mistaken for object detection. However, object recognition broadly encompasses image classification and object detection (Russakovsky et al., 2015). This work focuses on object localization and object detection. Adapted from (Brownlee, 2019).

Figure 5

The Elementary Process of an Image Classifier



Note. Adapted from (Redmond et al., 2016).

Traditional Computer Vision

The research community achieves a new state-of-the-art benchmark in object recognition every year. These immense achievements would not be possible without seminal historical works and the evolution of Deep Neural Networks and NVIDIA GPUs. The history of modern object detection can be split into two distinct eras. Traditional machine learning relied heavily on extracting handcrafted features (Krohn, Beyleveld, & Bassens, 2020). Edges, corners, gradients, and object components were all manually identified, such as the Deformable Parts Model (University of Central Florida, 2016). Viola-Jones created the first real-time face detector commercially applied to Kodak cameras (Krohn, Beyleveld, & Bassens, 2020); however, it struggled to detect faces sideways or upside down due to the nature of these angles not being considered (Sharma, 2022). Similar to traditional machine learning, these early detectors relied heavily on feature-engineered modifications on training data and less time modeling data through a detector (Krohn, Beyleveld, & Bassens, 2020). Figure 6 provides a timeline for seminal traditional computer vision methodologies.

Historically, traditional computer vision is derived from how natural neurons are arranged in the visual systems of animals. Hierarchical processing of vision was first discovered by (Hubel & Weisel, 1959). Images are first detected by visual receptors and processed by simple neurons that feed more complex neurons that can process more complex shapes, colors, and features of an object (Krohn, Beyleveld, & Bassens, 2020). Early versions of neural networks are referred to as Artificial Neural Networks (ANNs) or Multi-Layer Perceptrons (MLPs). At first, they were a shallow collection of neurons that were stacked on top of each other with weight connections as illustrated in Figure 7 (Elgendy, 2020). Traditional MLP architecture consists of an input layer, one or more hidden layers, and an output layer and its input layer only takes one-dimensional vectors; thus, for the process of complex imagery, two-dimensional images must be transformed into a one-dimensional vector. This required transformation is a process called image flattening demonstrated in Figure 8.

Figure 6

Traditional Computer Vision Approaches

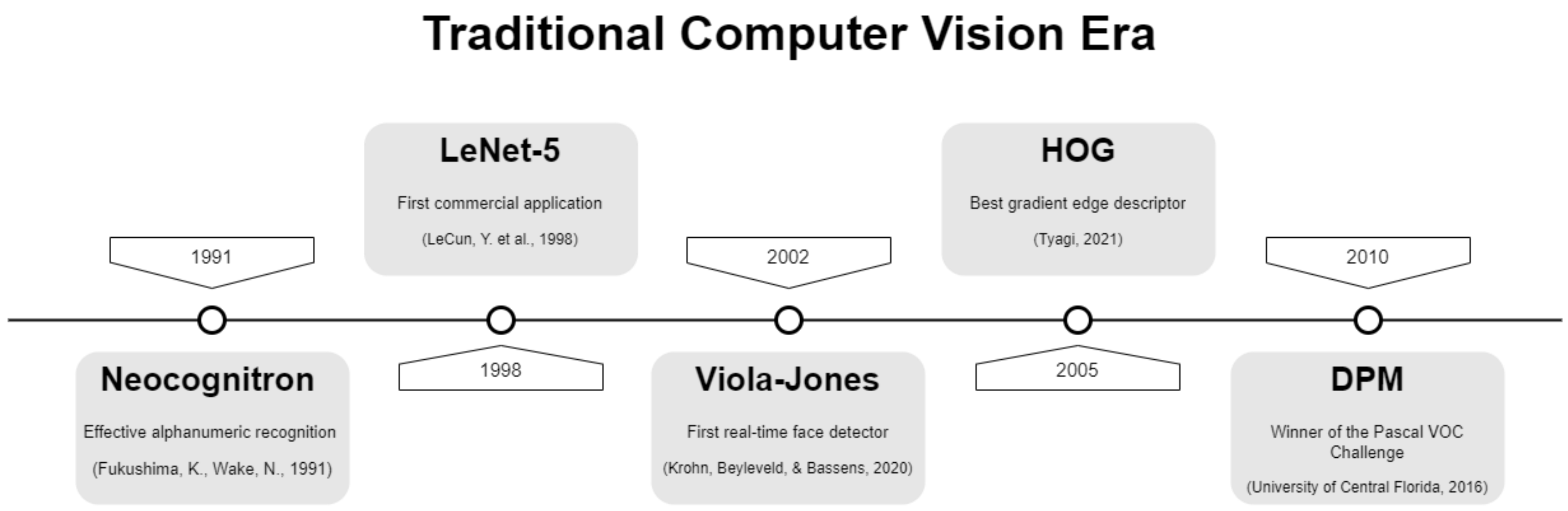
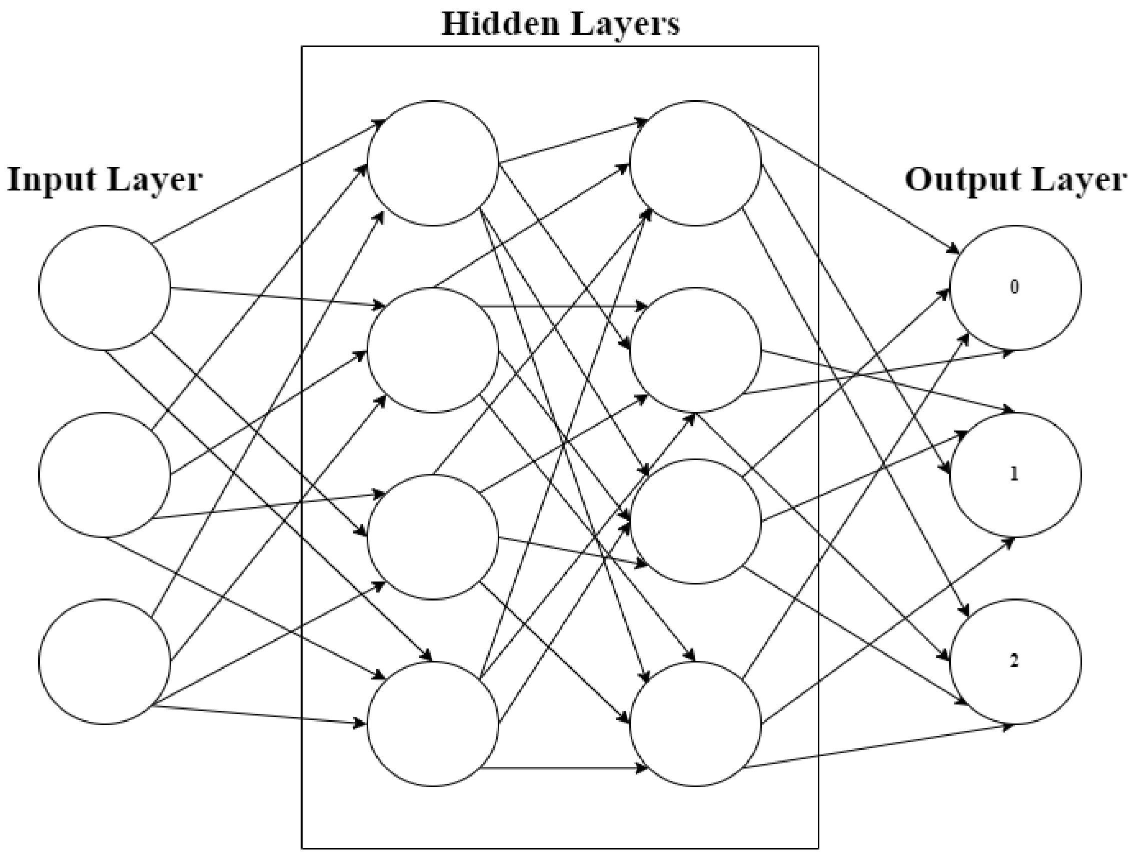

Traditional Computer Vision Era
1991
Neocognitron
Effective alphanumeric recognition
(Fukushima, K., Wake, N., 1991)
LeNet-5
First commercial application
(LeCun, Y. et al., 1998)
1998
2002
Viola-Jones
First real-time face detector
(Krohn, Beyleveld, & Bassens, 2020)
HOG
Best gradient edge descriptor
(Tyagi, 2021)
2005
2010
DPM
Winner of the Pascal VOC Challenge
(University of Central Florida, 2016)

Figure 7

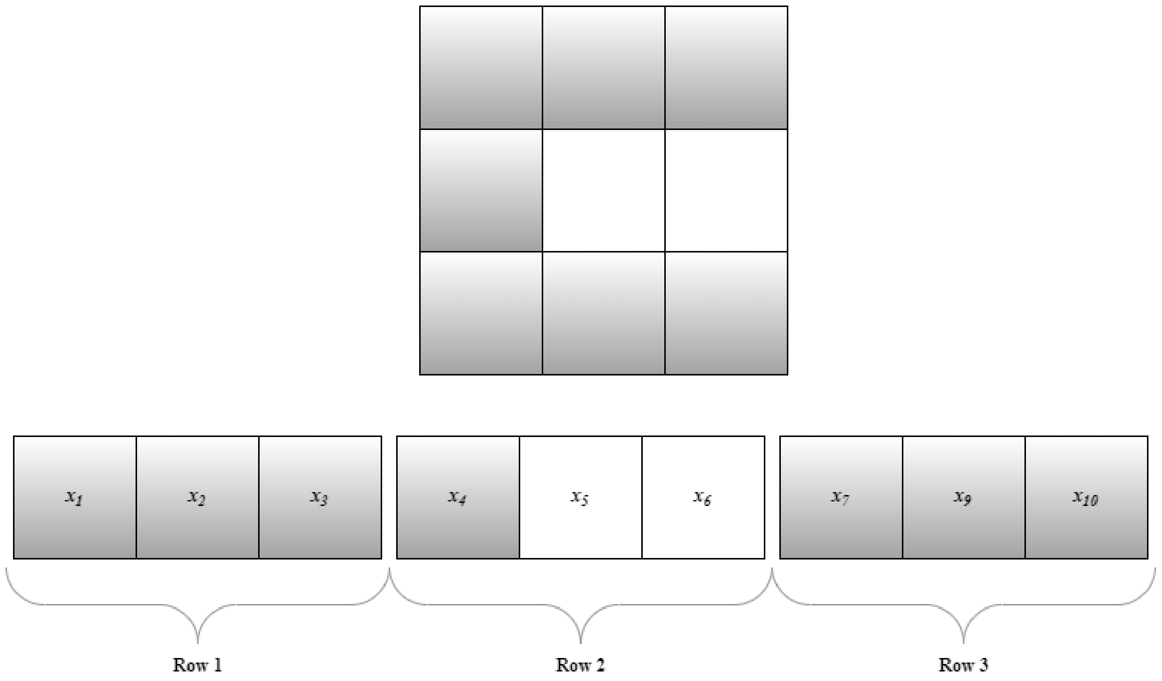
Multi-layer Perceptron Standard Architecture



Note. MLP architecture consists of layers of neurons connected by weights. Input layer may contain one or more neurons that take an input and forward-feed it to one or more layers of hidden neurons. Denoted by this graphic, hidden layers are often considered a "black-box" that observers do not access. These layers perform feature extraction of an image and then connect to the output layer that classifies an input. Adapted from (Elgendy, 2022).

Figure 8

Flattening of a 2D Image to a 1D Input Vector



Note. This figure visualizes the necessary conversion from a 2D image (top) to a 1D input vector (bottom). This is called image flattening and is required for MLP processing. In this example, a pixelated letter 'C' is converted from a two-dimensional image into a 'flattened' one-dimensional vector. Adapted from (Elgendy, 2022).

Modern Computer Vision

In 2012, a new era of neural networks began with the advent of convolutional neural networks (CNN), as summarized in Figure 9. Alex Krizhevsky and Illya Sutskever created a CNN called AlexNet and presented it at the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), and it beat long-standing traditional machine learning techniques by a massive margin (Krohn, Beyleveld, & Bassens, 2020). Modern-era CNN models eliminated the long-standing method of handcrafted feature engineering and focused more on neural network architecture optimization. Feature engineering is now mainly done by a neural network (Krohn, Beyleveld, & Bassens, 2020). CNNs achieve automatic feature extraction by expanding on traditional MLP architectures. Three main layers are prevalent in almost every CNN: convolutional layer, pooling layer, and fully connected layer (Elgendy, 2020). As shown in Figure 10, the fully connected layer borrows heavily from traditional MLP architectures. Modern CNNs expand previous generations of neural networks by focusing heavily on automatic feature extraction. Most CNNs share this formula; however, they all differ in the feature extraction layers. These advancements lead to the creation of single-stage object detectors (SSD) that treat object detection as a simple regression problem (Sharma, 2022). Treating object detection as a regression problem allows a much simpler network structure that accepts a variable-sized input image and, without prior feature engineering, is processed directly by the neural network. The output of this process is the class probability, along with bounding box coordinates (Sharma, 2022). In addition, SSDs can skip the region proposal stage and automatically detect areas of an image that could contain an object, as shown in Figure 11 (Sharma, 2022). This ability to skip a second stage allows SSDs like You Only Look Once (YOLO) to achieve much faster real-time detection speeds, albeit with an accuracy penalty.

Figure 9

Modern Computer Vision Approaches

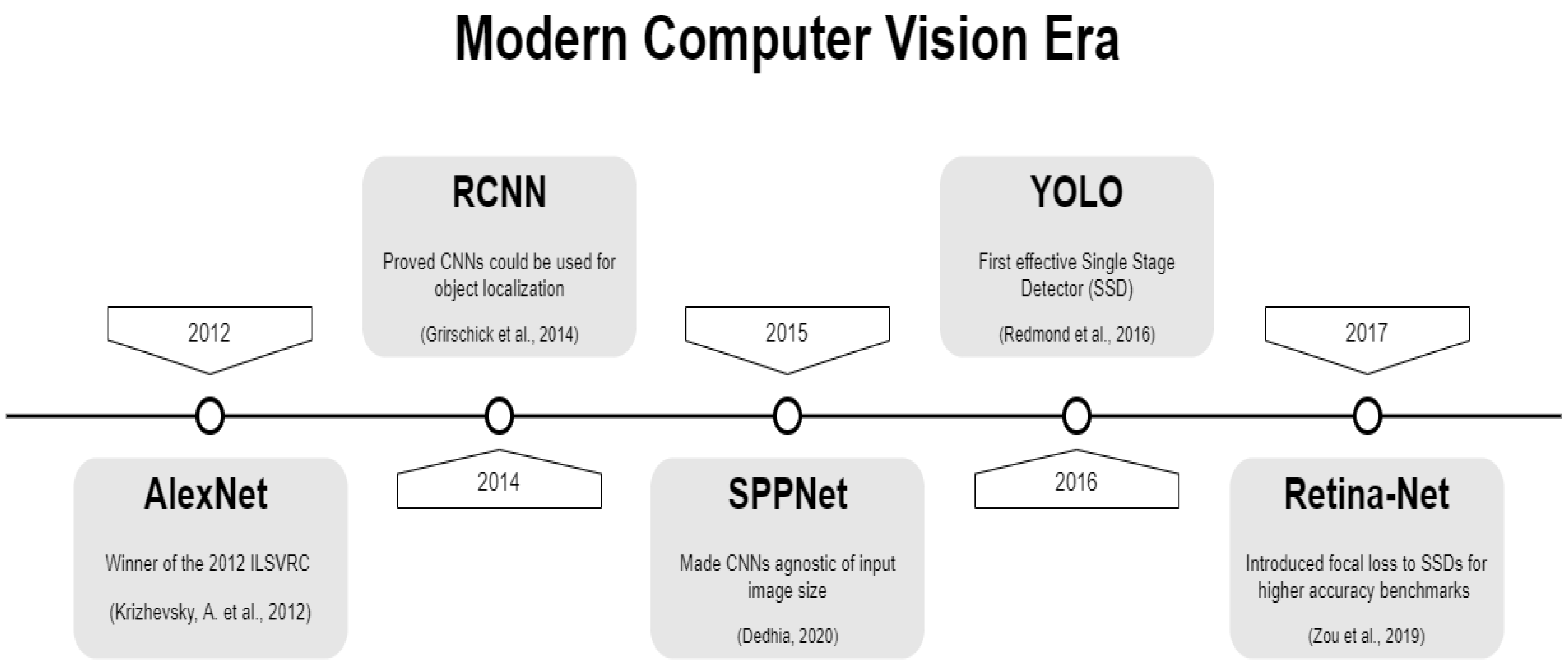

Modern Computer Vision Era
2012
AlexNet
Winner of the 2012 ILSVRC
(Krizhevsky, A. et al., 2012)
RCNN
Proved CNNs could be used for object localization
(Grirschick et al., 2014)
2014
2015
SPPNet
Made CNNs agnostic of input image size
(Dedhia, 2020)
YOLO
First effective Single Stage Detector (SSD)
(Redmond et al., 2016)
2016
2017
Retina-Net
Introduced focal loss to SSDs for higher accuracy benchmarks
(Zou et al., 2019)

Figure 10

Basic Component of Convolutional Networks

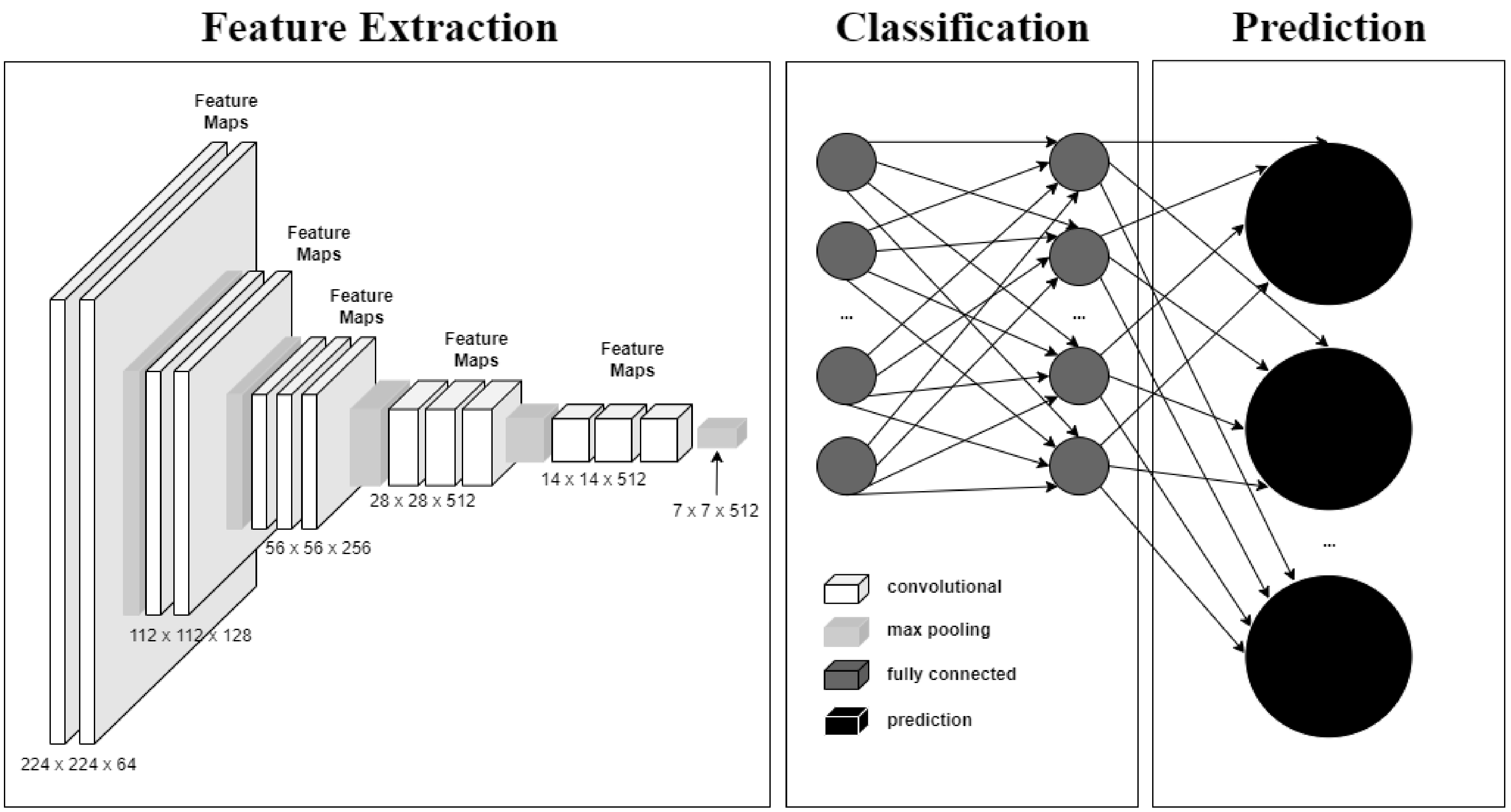
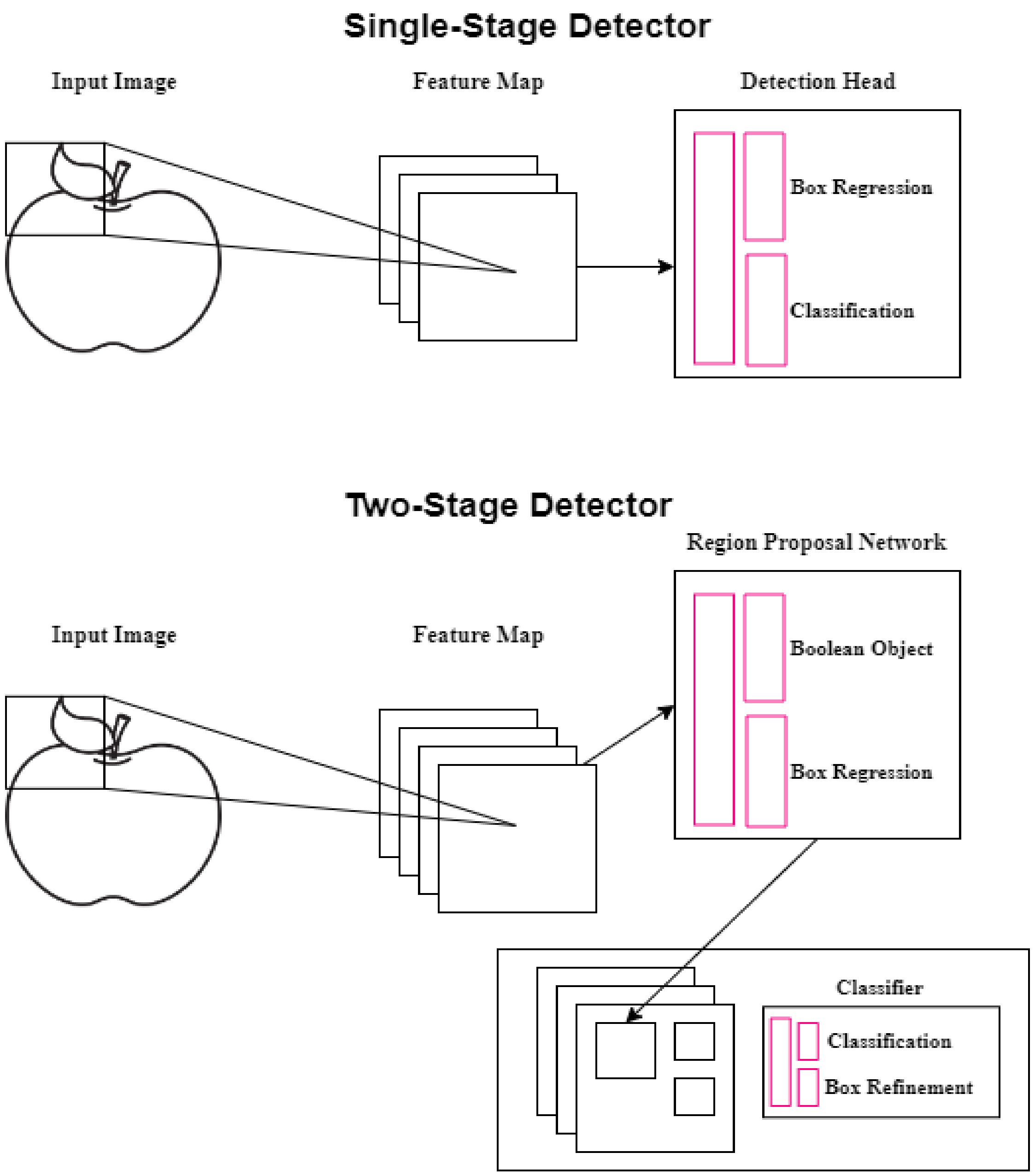


Figure 11

Difference between single-stage detectors and two-stage detectors



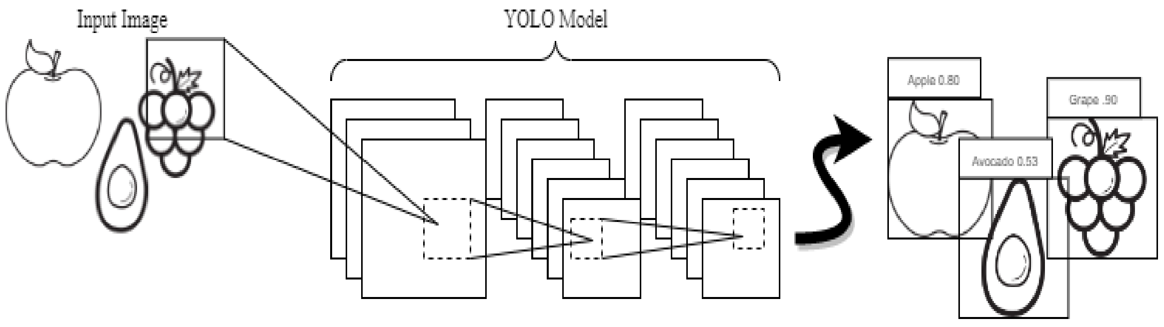
Note. Single-stage detectors skip the Region Proposal Network and this skip allows for much faster image processing than two-stage detectors. Albeit at a loss of accuracy penalty. Adapted from (Sharma, 2022).

You Only Look Once

The computer vision work featured in this thesis is over a popular single-stage detector family of object recognition models known as You Only Look Once (YOLO), developed by (Redmond et al., 2016). While other object recognition models such as ResNet, R-CNN, or Faster R-CNN may be more accurate, YOLO models are much faster in achieving real-time object recognition. Like many previous classifiers, YOLO takes an image input, breaks it into a grid, processes it through network layers, and directly predicts bounding boxes and class labels. For example, see Figures 12 and 13.

Figure 12

A Simplified View of You Only Look Once Model Processing

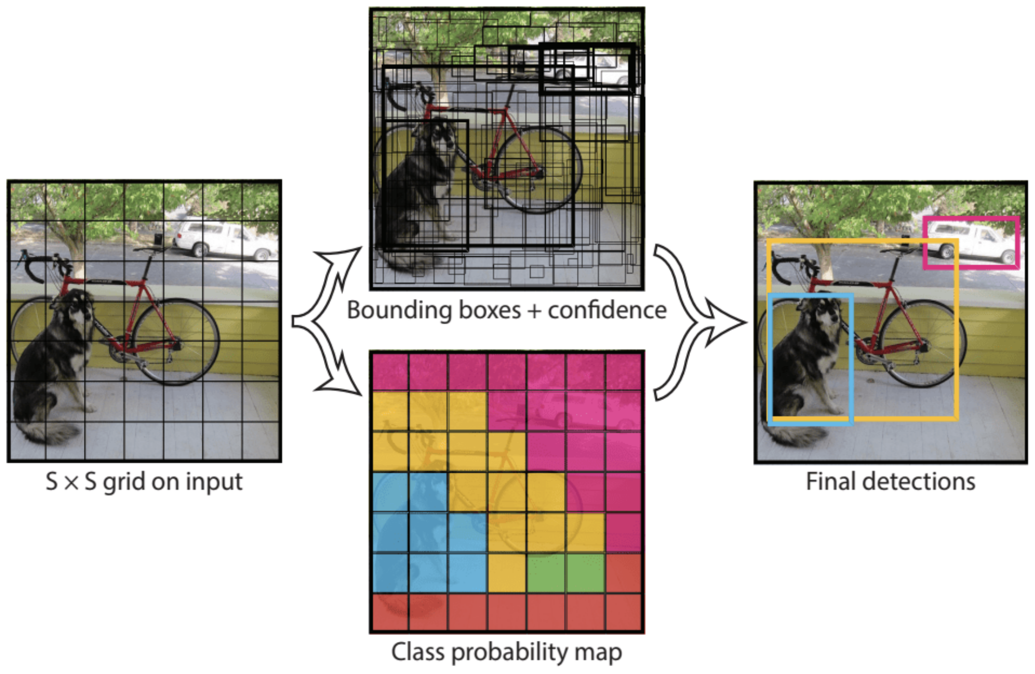


Note. Adapted from (Redmond et al., 2016).

The YOLO family is an end-to-end deep learning model that was one of the first attempts to successfully build a fast real-time object detector (Elgendy, 2020). It predicts over a limited number of probability areas by splitting an input image into a grid of cells, as summarized in Figure 13. Processing an image as a grid result in many potential bounding boxes consolidated into a final prediction using non-maximum suppression (Elgendy, 2020).

Figure 13

Summary of the Detection Process Used by YOLO



Note. An input image is broken up into a grid, and each cell is processed for potential bounding boxes using learned image features. As seen in the upper-central image, this generates many possible bounding boxes. To reduce all possible bounding boxes, YOLO implements non-maximum suppression to consolidate probabilities into a final detection (Elgendy, 2020). Taken from You Only Look Once: Unified, Real-Time Object Detection (Redmond et al., 2016).

YOLO is a breakthrough and seminal work in object recognition because it treated object detection with an SSD approach and as a regression problem. As a result, you only look once was chosen as an appropriate name because the detection layer only looked at an image once to predict the objects' location and class labels (Redmond et al., 2016). Over the years, YOLO has significantly improved through various releases and continues to be a premier real-time detection neural network available, as summarized in Figure 14. Table 2 details mainline YOLO models.

Figure 14

History of the YOLO Model Family

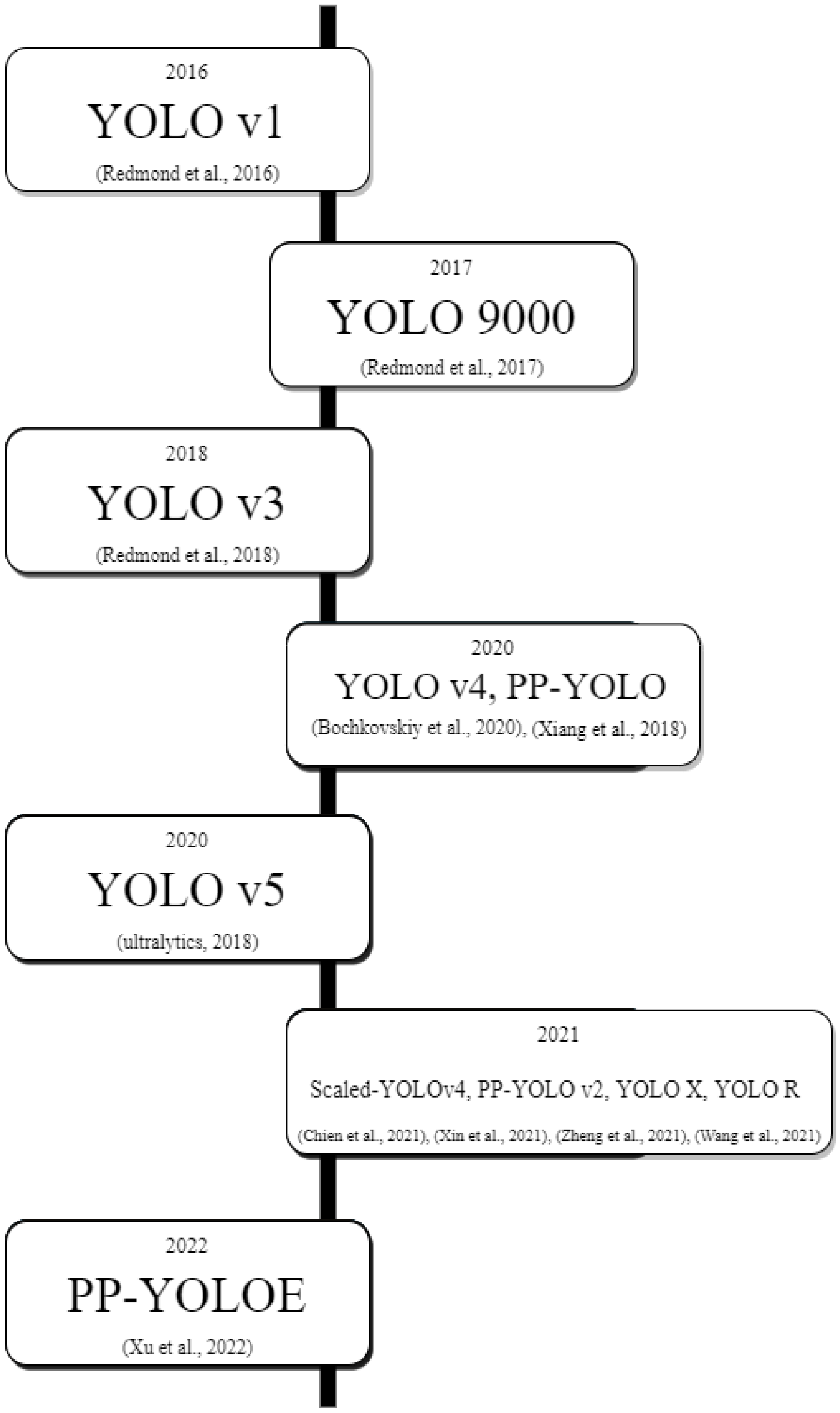


Table 2

Condensed model summary of YOLO

Reference	Model	Train Set	Conv. Layers	Total Layers (Conv + Filters)	Input size	mAP (%)	FPS
Redmon et al., 2016	YOLO	ImageNet	24	26	448x448	63.4	45
Redmon et al., 2016	Fast YOLO	ImageNet	9	11	448x448	52.7	155
Redmond & Farhadi, 2017	YOLO9000 Darknet-19	ImageNet & MS COCO	19	26	416x416	16.0	-
Redmond & Farhadi, 2017	YOLOv2 Darknet-19	VOC 2007 & MS COCO	19	26	288x288	69.0	91
Redmond & Farhadi, 2017	YOLOv2 Darknet-19	VOC 2007 & MS COCO	19	26	352x352	73.7	81
Redmond & Farhadi, 2017	YOLOv2 Darknet-19	VOC 2007 & MS COCO	19	26	416x416	76.8	67
Redmond & Farhadi, 2017	YOLOv2 Darknet-19	VOC 2007 & MS COCO	19	26	480x480	77.8	59
Redmond & Farhadi, 2017	YOLOv2 Darknet-19	VOC 2007 & MS COCO	19	26	544x544	78.6	40
Redmond & Farhadi, 2018	YOLOv3 Darknet-53	Darknet Framework	53	106	320x320	51.5	45
Redmond & Farhadi, 2018	YOLOv3 Darknet-53	Darknet Framework	53	106	416x416	55.3	35
Redmond & Farhadi, 2018	YOLOv3 Darknet-53	Darknet Framework	53	106	608x608	57.9	20
Bochkovskiy, Wang, & Liao, 2020	YOLOv4 Darknet-53	MS COCO	53	162	416x416	41.7	92

(continued)

Reference	Model	Train Set	Conv. Layers	Total Layers (Conv + Filters)	Input size	mAP (%)	FPS
Bochkovskiy, Wang, & Liao, 2020	YOLOv4	MS COCO	53	162	512x512	43.0	87
Bochkovskiy, Wang, & Liao, 2020	YOLOv4	MS COCO	53	162	608x608	44.0	68
ultralytics, 2020	YOLOv5n	MS COCO	53	Darknet-53+PANet+Yolov3	640x640	45.7	-
ultralytics, 2020	YOLOv5s	MS COCO	53	Darknet-53+PANet+Yolov3	640x640	56.8	-
ultralytics, 2020	YOLOv5m	MS COCO	53	Darknet-53+PANet+Yolov3	640x640	64.1	-
ultralytics, 2020	YOLOv5l	MS COCO	53	Darknet-53+PANet+Yolov3	640x640	67.3	-
ultralytics, 2020	YOLOv5x	MS COCO	53	Darknet-53+PANet+Yolov3	640x640	68.9	-
Long et al., 2020	PP-YOLO	MS COCO	50	ResNet50-vd-dcn	320x320	39.3	132
Long et al., 2020	PP-YOLO	MS COCO	50	ResNet50-vd-dcn	416x416	42.2	109
Long et al., 2020	PP-YOLO	MS COCO	50	ResNet50-vd-dcn	512x512	44.4	90
Long et al., 2020	PP-YOLO	MS COCO	50	ResNet50-vd-dcn	608x608	45.2	73
Zheng, et al., 2021	YOLOX-Nano	MS COCO	-	-	416x416	28.5	-
Zheng, et al., 2021	YOLOX-Tiny	MS COCO	-	-	416x416	32.8	-
Zheng, et al., 2021	YOLOX-S	MS COCO	-	-	640x640	40.5	-

(continued)

Reference	Model	Train Set	Conv. Layers	Total Layers (Conv + Filters)	Input size	mAP (%)	FPS
Zheng, et al., 2021	YOLOX-DarkNet53	MS COCO	53	Darknet-53+Yolov3	640x640	47.7	-
Zheng, et al., 2021	YOLOX-L	MS COCO	-	-	640x640	50.0	-
Wang et al., 2021	YOLOR-P6	MS COCO	-	YOLOv4-CSP	1280x1280	54.1	76
Wang et al., 2021	YOLOR-W6	MS COCO	-	YOLOv4-CSP	1280x1280	55.5	66
Wang et al., 2021	YOLOR-E6	MS COCO	-	YOLOv4-CSP	1280x1280	56.4	45

Note. Table 2 is compiled with information provided by Sharma, 2022, and each respective model reference. The models above are the ones listed upon release in their respective works. Nearly all models have updated versions, such as Scaled-YOLOv4 (Chien et al., 202), PP-YOLOE (Xu et al., 2022), or YOLOv5 ver. 6 (ultralytics, 2022). Listing all possible variations and configurations is irrelevant to this work, and not all updated versions are accompanied by peer-reviewed work. A vast majority of these models are based on Darknet-19 or Darknet-53 architectures. This means a model has 19 or 53 convolutional layers. Commonly, 19-layer or 53-layer notations do not include the varying filter, drop-out, or classification layers featured in Figure 10. Notably, the YOLOX family uses a primary baseline model, YOLOX-Darknet53, which uses the Darknet53 framework. However, its derived models, such as YOLOX-Nano or YOLOX-L, use a modification of the baseline architecture (Sharma, 2022). These modifications are not directly stated in the work by (Zheng et al. 2021). YOLOR, for instance, uses YOLOv4-CSP as a baseline model and Scaled-YOLOv4 training parameters along with the YOLOv5 training data directory structure. Table 2 summarizes the characteristics of each model in terms of model architecture size, input image size, and frames-per-second. The deeper the architecture and the larger the input image size, the greater mean average precision (mAP) that can be achieved, which usually comes at the cost of lower frames-per-second.

CHAPTER IV

Proposed Approach

The ultimate goal of this project is to lay down a foundation that will allow UAV-mounted real-time object detection of RCW habitats. Achieving this goal means considering the small memory availability on a UAV system. Considering this constraint, the proposed approach will use the smallest versions of YOLOv4 and YOLOv5. The benefit of using two models is to validate results real-time when performing testing inferencing. The smallest models, YOLOv4-tiny and YOLOv5n, are trained using the identical training data. However, due to the two models technically belonging to different families, their architectures and parameters are slightly different. These differences are reconciled and explained later. As shown in Table 2, models with fewer layers achieve faster object recognition times, and this is a benefit when flying a drone in real-time. Another justification, when choosing the two models, is on graphical processing unit (GPU) memory limitations. Despite achieving better mean average precision (mAP) results, larger versions of YOLOv4 and YOLOv5 are more memory intensive and have a slower identification time.

The GPU memory constraint is a universal issue in object recognition training, particularly with images. For example, given a relatively small input image of 416×416 pixel dimensions and using YOLOv4-tiny, it would take around 2 hours of training for an NVIDIA RTX 2080 GPU system to complete the standard 10,000 training iterations. In contrast, it may take an 8-core 3.40 GHz central processing unit (CPU) based system around six days to achieve the same result (ccoderun, 2018). Updated models such as YOLOv5 ver. 6 and Scaled-YOLOv4 require an image input dimension of 1280×1280.

i.e. the larger the image input dimension, the larger the GPU memory requirements. The maximum input size possible before running out of GPU memory is 738×738 pixels on the hardware available for this project. In essence, memory limitations severely hamper the number of possible models that can be trained. Cloud computing services such as Google's Google Colab, Amazon Web Services, Google Cloud Platform Deep Learning VM, Roboflow, or Kaggle are possible replacements for direct hardware training. However, these services require payment, credit limitations, or purchasing an upgrade tier to receive faster computing speeds, and are still not as fast as training on a local computer.

Dataset Description

The entire dataset used in this study was created by wildlife biologists, Brett Lawrence and Jesse Exum, at Raven Environmental Services, Inc. All images were photographed at Cook's Branch Conservancy (CBC) and various parts of the greater Sam Houston National Forest (SHNF) located in Montgomery County, Texas, as illustrated in Figures 15 and 16. The dataset was collected between March 2020 and March 2021 and split into two batches. The imagery was captured with a DJI Mavic Pro Platinum and a DJI Mavic Pro in the red-green-blue color bands with approximate flight times of 4-5 hours (Lawrence, 2022). See Table 6 for more details on drone hardware. The spring season is the best time to capture imagery as it is the dormant season when deciduous trees are in a leaf-off state (Lawrence, 2022). Less foliage allows for more direct discrimination between hardwood and pine species and easier detection of red-cockaded woodpecker resin markers that leaves may obscure. This dataset is publicly available at

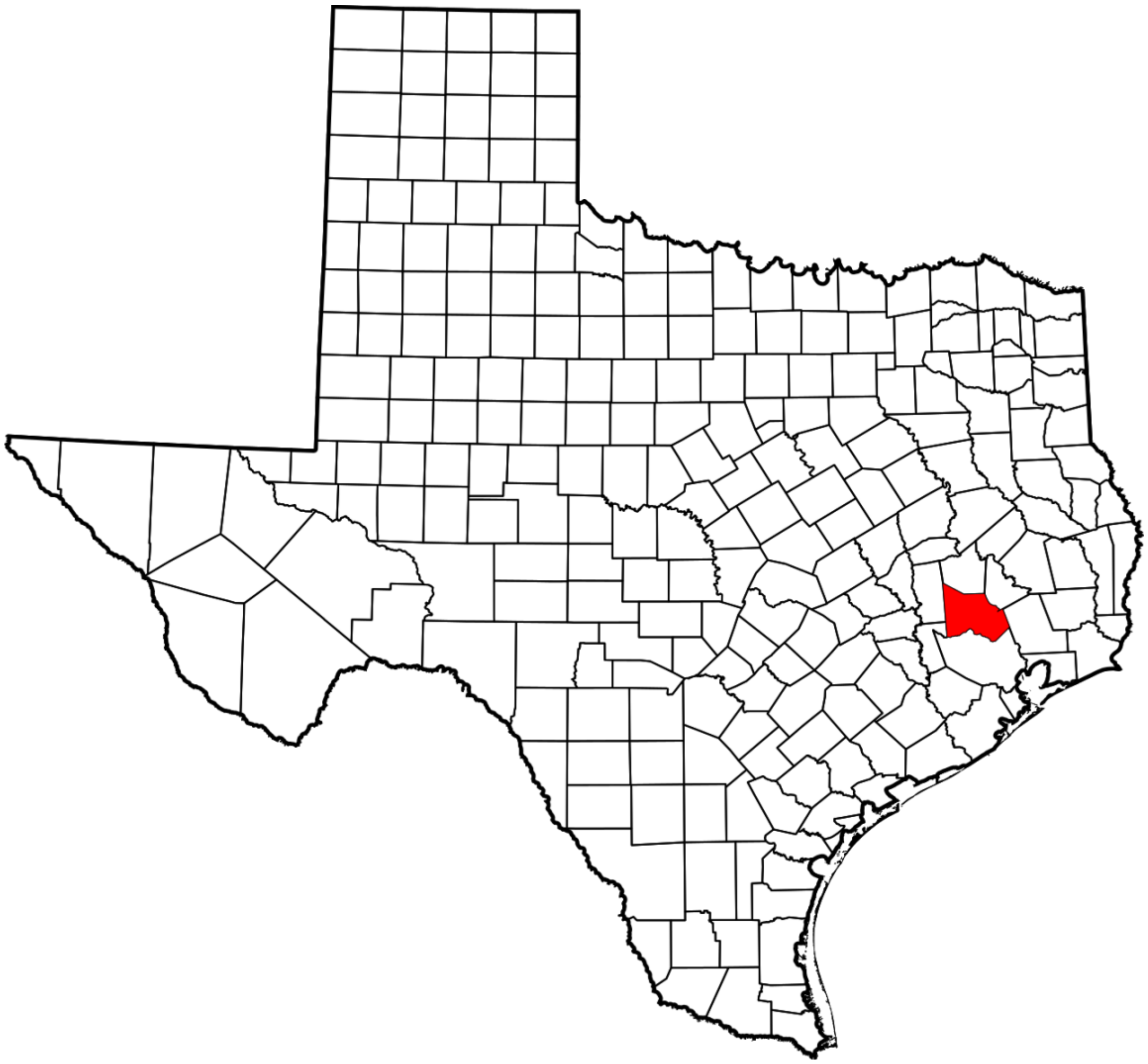
Roboflow.com courtesy of Raven Environmental Services, the Mitchell Foundation, the U.S. Fish and Wildlife Service, and the U.S. Forest Service.

Geographic Location

As shown in Figure 16, the image data for this project was captured in the Sam Houston National Forest (SHNF) and Cook's Branch Conservancy (CBC), located in Montgomery County, Texas.

Figure 15

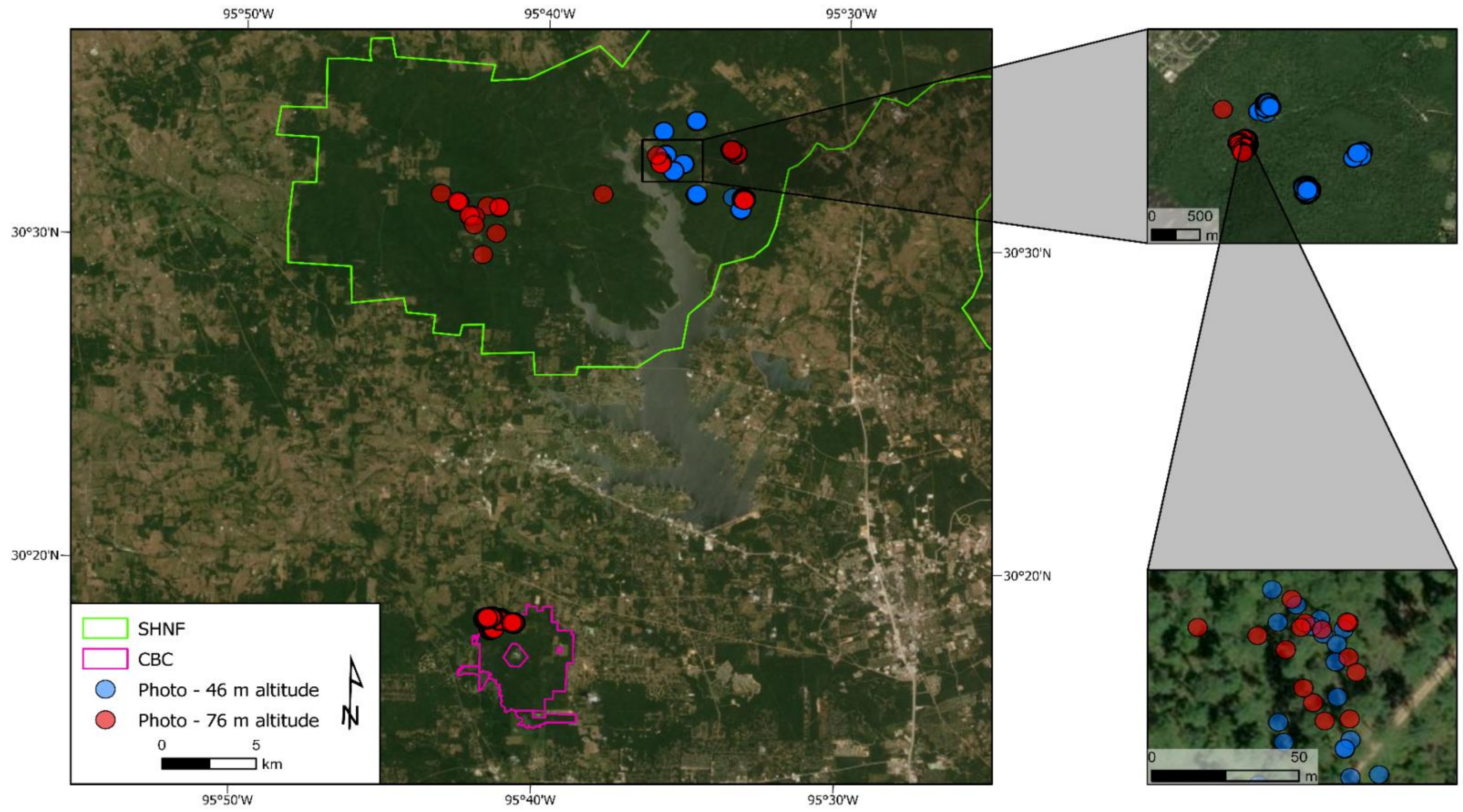
Montgomery County, Texas



Note. Highlighted in red is Montgomery County, Texas. This study focuses on this region of East Texas. Adapted from nationalatlas.gov.

Figure 16

Map Detailing all Locations where the Dataset was Gathered



Dataset (1)

Brett Lawrence and Jesse Exum collected the first part of the dataset. The dataset collection occurred in the SHNF and CBC during the spring 2022 season. UAVs were utilized for data collection, and the dataset is further described in Table 3 with a featured annotation heatmap in Figure 17.

Table 3

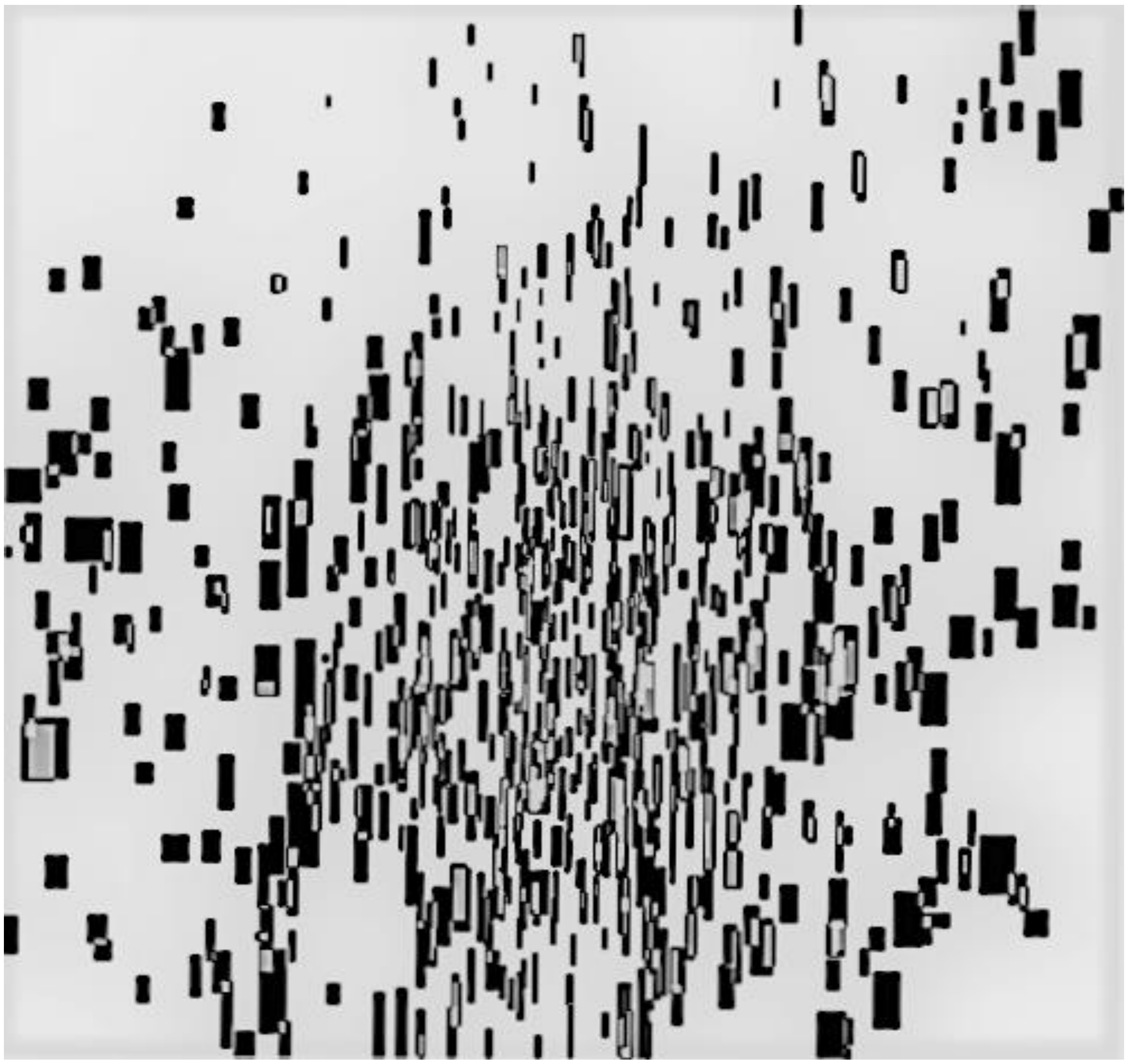
534 Images Taken at 76m Elevation

Total Images	Class	Train	Valid	Test	Total Annotations	Null Images	Average Annotation per Image	Average Image Size	Average Image Ratio
534	1	428	53	53	765	94	1.4	16.84mp	5472✕3078

Note. The total image count is split 80% training, 10% validation, and 10% test. This split is standard practice; however other divisions can be possible, such as 80% training and validation and 20% test. No test images contained annotations.

Figure 17

Annotation Heatmap of 534 Images Taken at 76m Elevation



Note. Figure 17 represents 765 annotations across the first batch of images. The image demonstrates the distribution of all annotation locations. Note the wide distribution of annotation box locations and annotation box size. Black annotation boxes denote the area does not overlap with other annotation boxes. White annotation regions indicate a high density of annotation overlapping. Generated with Roboflow.

Dataset (2)

Brett Lawrence and Jesse Exum also created the second dataset to provide a diverse approach. It was taken during the same period, spring 2022; however, forest imagery was captured at 46m altitude. This elevation difference is visualized in Figure 19. This batch of data contains 90 fewer images than the first batch. The dataset is further described in Table 4 with a featured annotation heatmap in Figure 18.

Table 4

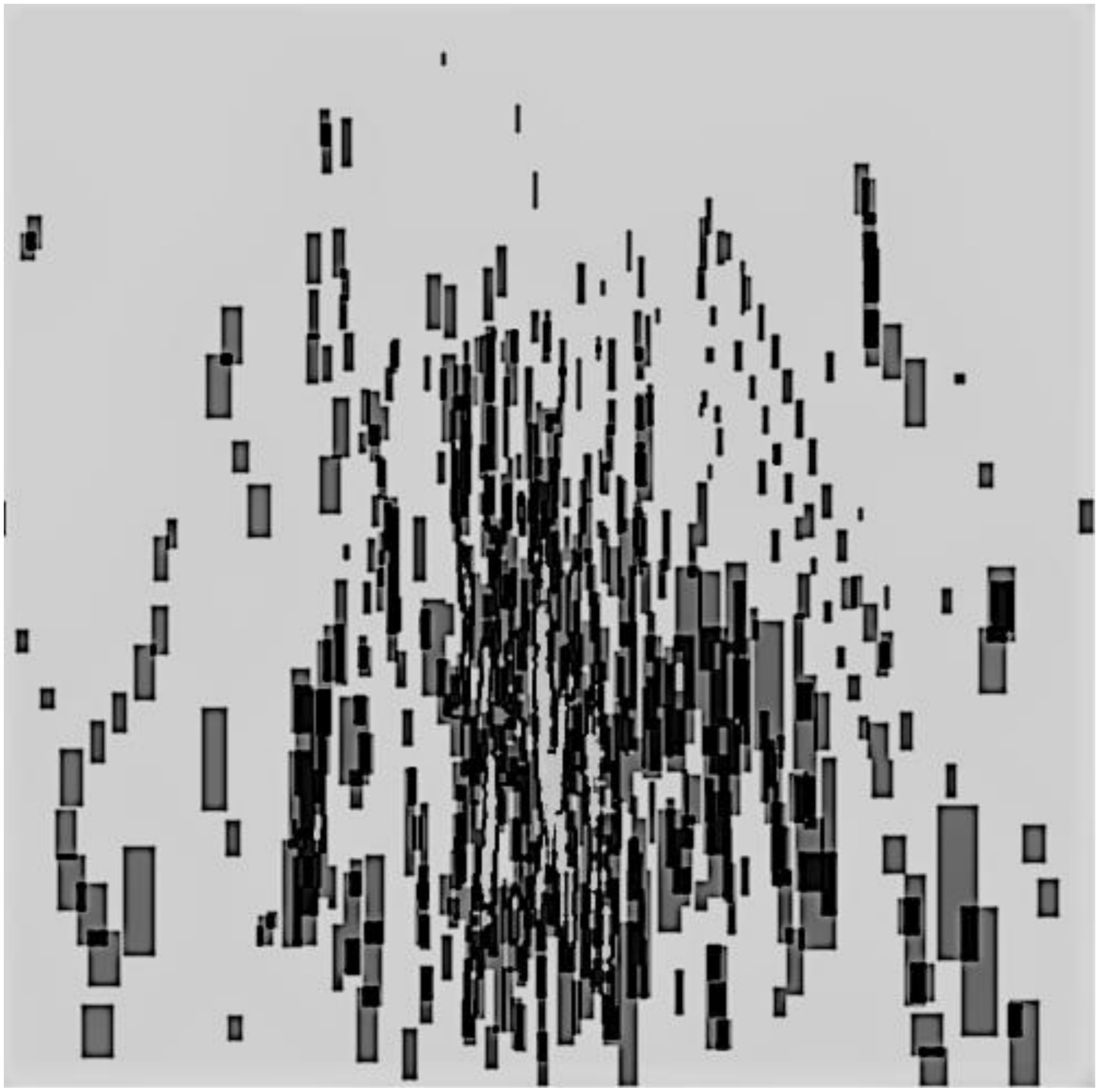
444 Images Taken at 46m Elevation

Total Images	Class	Train	Valid	Test	Total Annotations	Average Annotation per Image	Null Images	Average Image Size	Average Image Ratio
444	1	307	87	50	678	1.5	27	9.00mp	4000×2250

Note. The total image count is split 80% training, 10% validation, and 10% test. This dataset was taken at a much lower 46m elevation. Note that the average image ratio decreased by over 1,000 pixels in height and width compared to the first batch. This dataset batch introduces a closer look at the forest and provides more angle diversity when observing RCW sap signs.

Figure 18

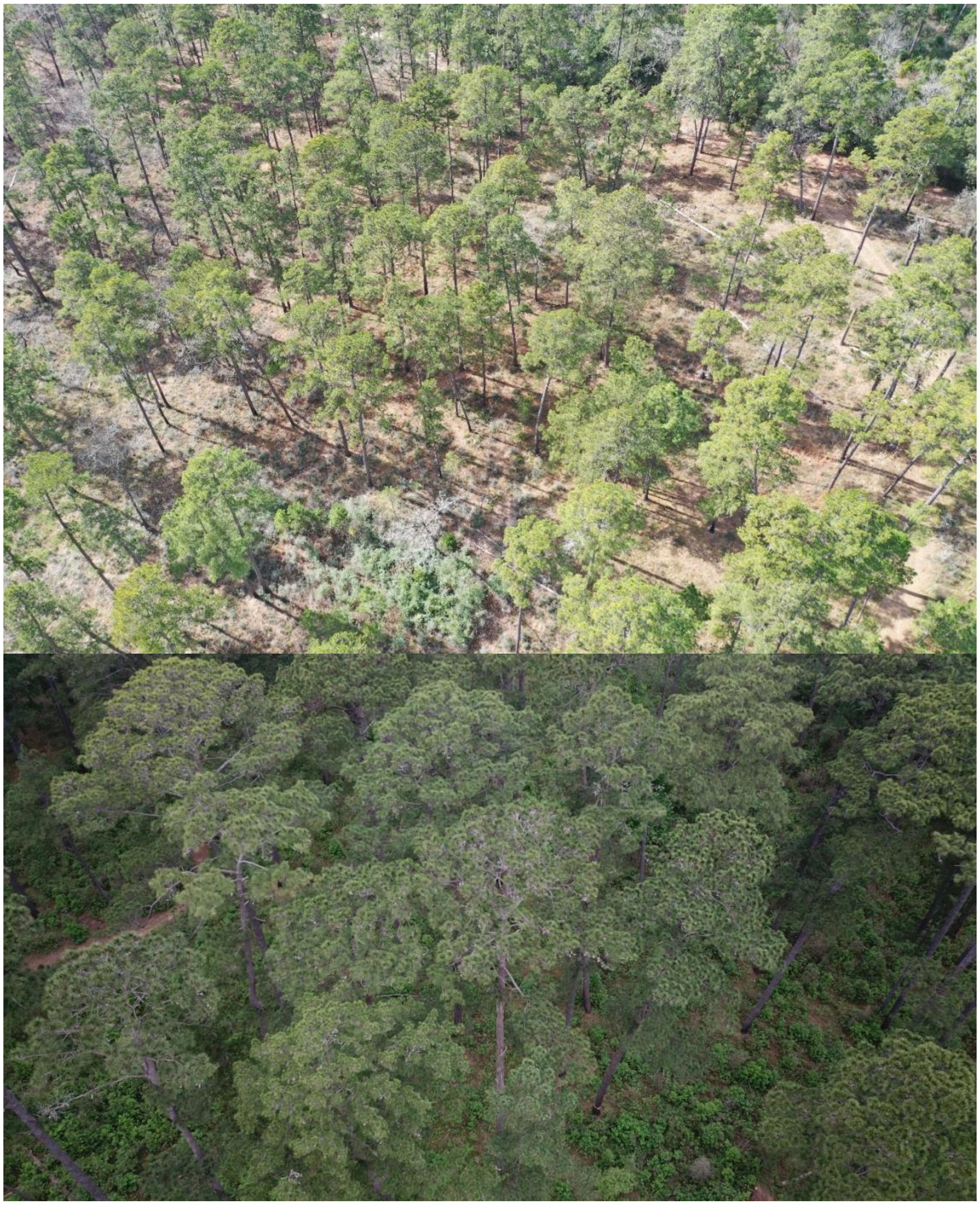
The Annotation Heatmap of 444 images Taken at 46m Elevation



Note. The image above represents 678 annotations across the second batch of images. This dataset displayed a different distribution of annotations compared to Figure 15. Generated with Roboflow.

Figure 19

Sample Images Demonstrating Altitude Differences



Note. Altitude matters when surveying tracts of forest. The top image was taken at 76m altitude, and the bottom image was at 46m altitude. Note the difference in forest coverage and potential nesting locations.

Hardware Tools

Hardware tools used for this study include both computational hardware and UAVs. Computational hardware is detailed in Table 5, and UAV hardware is detailed in Table 6. Raven Eviornmental Inc owns all UAV hardware used for this study.

Table 5

Computer Hardware Tools

GPU	GPU Memory	CPU	RAM
NVIDIA GeForce RTX 2080 Super Max Q (Notebook)	8 GB	Intel Core i9-10980HK @ 2.40 GHz x 16	32 GB
NVIDIA GeForce RTX 3080 (Notebook)	12 GB	AMD Ryzen 9 5900HK	32 GB

Table 6

UAVs Deployed

Model	DJI Mavic Pro Platinum	DJI Mavic Pro 2
Weight	734 g	907 g
Flight time	30 m	31 m
Sensor	1/2.3" (CMOS)	1" (CMOS)
Pixels	12.35 million	20 million
Lens	FOV 78.8°, 26mm (35mm format equivalent), aperture f/2.2, shooting range from 0.5 m to ∞	FOV about 77°, 28mm (35mm format equivalent), aperture f/2.8-f/11, shooting range from 1 m to ∞
ISO Range	100-1600	100-3200 (auto), 100-12800 (manual)
Electronic Shutter Speed	8-1/8000 s	8-1/8000 s
Size	4000x3000	5472x3648

Note. Information provided by Brett Lawrence at Raven Environmental Inc.

Software Tools

The software tools used for YOLOv4 and YOLOv5 are detailed in Tables 7 and 8, respectively.

Table 7

Software Tools for YOLOv4

Software	Version
Operating System	Ubuntu 20.04.4 LTS 64 Bit
Python	>=3.8
Github Package	github.com/AlexeyAB/darknet.git
CMake	3.18
CUDA	11.6
cuDNN	8.3
Darknet	AlexeyAB version
DarkHelp	darkhelp-1.1.18-3
DarkMark	darkmark-1.6.28-1
YOLO Model	yolov4-tiny.cfg (Bochovskiy, 2020)

Note. All necessary tools are outlined by (Bochkovskiy, 2020) and (Charette, 2020). Installation of the aforementioned software tools requires careful instruction, and environments such as Anaconda are recommended but not required. All the software listed is freely available to use as of this writing.

Table 8

Software Tools used for YOLOv5

Software	Version
Operating System	Microsoft Windows 10 Pro 10.0.19044 Build 19044
Python	>=3.8
Github Package	github.com/ultralytics/yolov5.git
Microsoft Visual Studio	2022 17.2
CMake	C++ CMake tools for Windows
CUDA	11.6
cuDNN	8.3
Darknet	-
DarkHelp	-
DarkMark	-
OpenCV	7.1.2
PyTorch	1.8.1
Tensorboard	2.4.1
Torchvision	0.8.1
Torch	1.7.0
YOLO Model	YOLOv5n (ultralytics, 2020)

Note. All necessary tools are outlined by (ultralytics, 2020). Installation of the software mentioned above tools requires careful instruction, and environments such as Anaconda are recommended but not required. All the software listed is freely available to use as of this writing. YOLOv5 is the most accessible model as it is compatible with cloud-based services like Google Colab, Kaggle, Docker, Amazon Web Services, and Google Cloud Deep Learning V.M. (ultralytics, 2020).

CHAPTER V

Experimental Study

Discussion

The experimental study to detect RCW habitats was performed with YOLOv4 tiny and YOLOv5n. The two YOLO models are chosen to verify each other's results in the form of a majority vote. Like ensemble modeling, two or more similar or diverse models are trained to predict or verify an outcome (Kotu & Deshpande, 2015). Ensembling aggregates each model's prediction and presents a result in one final prediction. Manual verification of ensembling can be used to compare two or more model output predictions and compare them to validate a result. Essentially, if more than one model produces a positive response for output, there is a higher likelihood of it being an accurate positive result. Due to the aforementioned hardware constraints, YOLOv4-tiny and YOLOv5n are the two appropriate YOLO models with the small layer configuration and the small input size requirement. YOLOv4-tiny is optimized at 416×416 pixel input, while YOLOv5n is optimized at 640x640 pixel input. Technically speaking, images larger than 736×736 image dimensions on an NVIDIA GeForce RTX 2080 Super Max Q notebook variant and images larger than 1,280×1,280 on an NVIDIA GeForce RTX 2080 notebook variant exceed the memory requirements. Therefore, more expensive hardware capacity is required for deep models and larger input image sizes. For instance, the famous repository created by (Bochkovskiy, 2020), popularly known as AlexeyAB, benchmarked YOLOv4 using a very high capacity, albeit expensive NVIDIA Tesla V100 GPU.

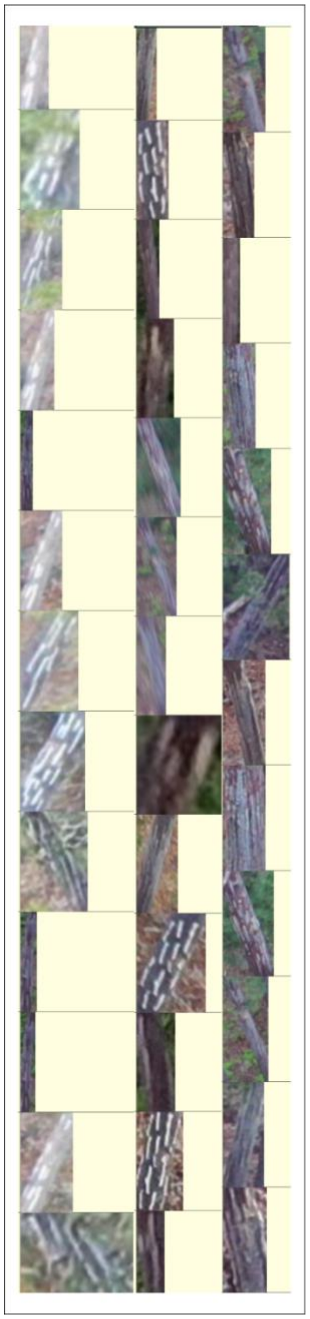
Training

The challenging aspect of training with red-cockaded woodpecker (RCW) habitats is the wide diversity of potential habitat signs, as shown in Figure 19. RCWs prefer nesting on the upper-third of old-growth pine trees (Lawrence, 2022) and keep pine resins constantly flowing by creating tree cavities. Therefore, pine resin is a highly visible and effective sign that denotes RCW presence. Due to camera angle, tree growth orientation, image quality, and extremely small object area, as shown in Figure 20, the resin signs are challenging to detect even with an expert's visual inspection. Furthermore, the signs are only partially consistent in their creamy white color that fades over time. Figure 21 also creates a perspective on the challenging training task. In addition, most object recognition models are trained on datasets such as MS COCO or ImageNet, where training data only contains highly defined objects with little ambiguity to their shape, color, or size. Therefore, strangely shaped objects such as the RCW pine resin signs require large amounts of training data to account for their diverse shape.

Initial training attempts on Roboflow yielded disastrous results due to a poor understanding of image augmentation techniques and image auto-resizing done by a neural network to fit its input dimension shape. Later attempts were made using DarkHelp and Darknet with much more successful results. Although Roboflow is very user-friendly, DarkHelp provides better technical control over data augmentation, data preparation, and configuration file generation on both Windows and Linux operating systems.

Figure 20

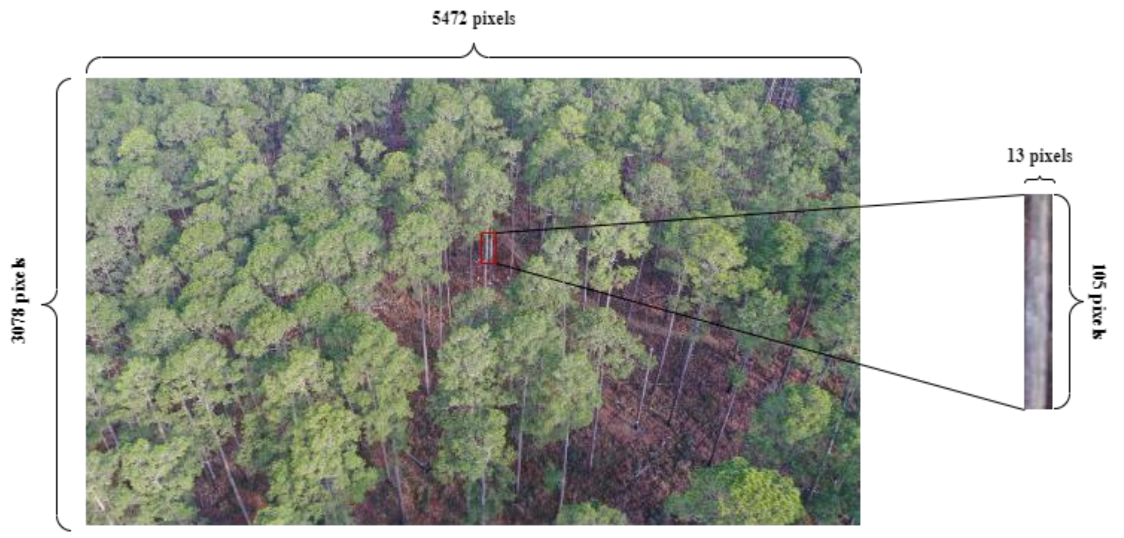
Sample Red-cockaded Woodpecker Habitat Signs



Note. These are sample bounding boxes from the training set for YOLOv4-tiny and YOLOv5n. To isolate the object of interest, bounding boxes are manually drawn using annotation software such as DarkMark, LabelImg, or Roboflow. The neural network model will then learn to identify the object of interest. Note the significant variance in image quality, angle, shape, and color. Screenshots were taken using DarkMark (ccoderun, 2018).

Figure 21

Illustration of a Small-size RCW Pine Resin Sign Compared to the Entire Image



Note. Pine resin is often no more than a smudgy, low-resolution object on average ~12,000 times smaller than its parent image. This significant difference leads to many challenges when detecting a successful nesting location.

YOLOv4-tiny requires an input size of 416×416 pixels, and YOLOv5n requires an input size of 640×640 pixels. As input sizes are variable, any multiple of 32 pixels is allowed; however, these models are optimized to work on these pre-defined dimensions. As known, increasing the image input size leads to more GPU memory requirements, and decreasing the image input size may lead to a loss in image quality due to compression. For example, taking an image of ~5,000×3,000 pixels and allowing YOLO to compress it down to 416×416 pixels to match its designed input size automatically leads to a significant loss of image quality, which makes the detection of any small object impossible. Table 9 demonstrates the average pixel sizes of all the annotations featured in the combined datasets. Note the sizes are in terms of pixels. See Figure 22 for an example of YOLOv4-tiny input dimension compression.

Table 9

Annotation Box Statistics

Class ID	Class Name	Images	Min Size	Avg Size	Max Size
0	cavity tree	612	15✕36	36.24✕115.80	149✕342
1	empty image	167	4000✕2250	5137.05✕2950.85	5472✕3648

Note. All dimensions are in terms of pixels.

Figure 22

Yolo Rescaling of an Image to Match Specified Input

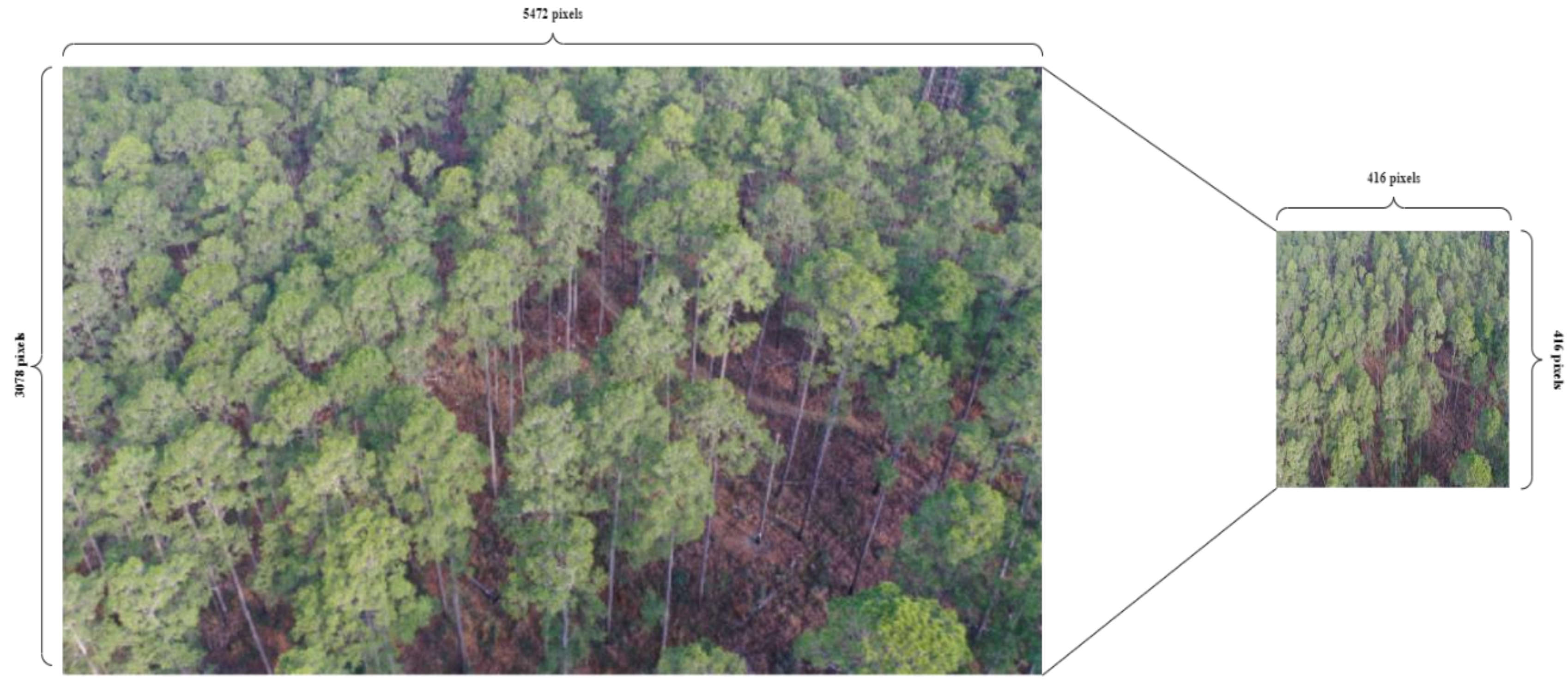
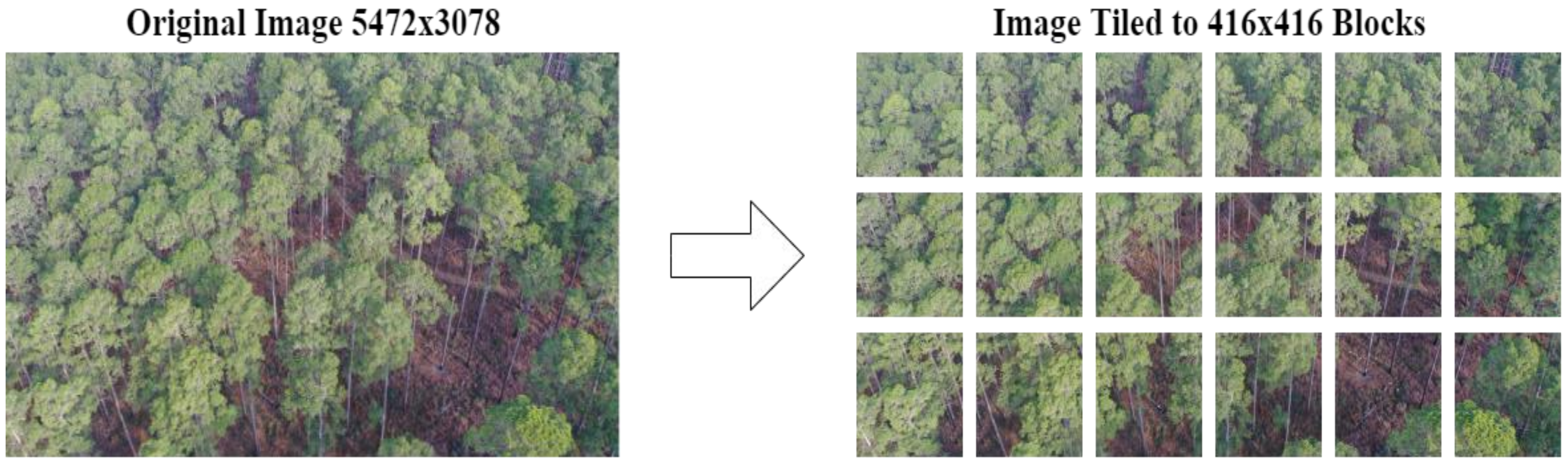


Image Tiling

Image tiling is a technique that prevents data loss when an image must match a neural network input dimension. Using high-resolution images assists in small object detection; however, it requires enormous GPU memory. Division of a large image into multiple small tiles guarantees no image quality is lost, as illustrated in Figure 23. If a training image approximately matches the network dimensions, then a user does not need to change anything. However, if an image exceeds 1.5 times the input dimensions and the object of interest is very small, then image tiling is necessary (Charette, 2020). For example, RCW pine resin is on average ~12,000× smaller than its parent image, and the object of interest only occupies approximately ~0.008% of the original image area. However, if we incorporate image tiling at 416×416 pixels, the object of interest occupies on average ~0.789% of the new image area. This noticeable increase in the apparent object-to-image ratio is beneficial when detecting smaller objects.

Figure 23

Image Tiling in Practice



Results

YOLOv4-tiny

YOLOv4 has the most significant amount of usage and online documentation to date. Unfortunately, YOLOv5 has no associated formal research paper, and it is difficult to reference a specific version due to constant Github updates. Thus, many of the experimental optimization for this novel object was done with YOLOv4-tiny using Darknet. Each complete run of 100,000 iterations took an average of 35 hours, and some experimental trials were cut short due to initial negative training results and the length of time required for completion. Once the dataset was slowly corrected and optimal parameters were found for a particular object, YOLOv5n was trained with very few scenarios.

Scenario 1

The first training scenario began with the first batch of 534 images and YOLOv4-tiny. 'Cavity' was the class name assigned to our object of interest. Any image with no cavities was deemed 'null'. To gain benchmark training mAP and preserve the original dataset provided, only 416×416 image tiling was applied to each training and validation image. No modifications to the original bounding boxes were applied. Darknet, by default, performs the following image augmentations 90-degree rotation, 15% zoom, and horizontal flipping. The total amount of images resulting from the default image augmentations was 146,040, of which 116,832 (i.e., 80%) images served as training data, and 29,208 (i.e., 20%) images served as validation data.

YOLOv4 requires a specific directory format that must be specified in its training path. Therefore, each image file must have a corresponding annotation text file matching the same name.

If this format is violated, training will not be executed. Test images are not necessary to start the training process. To avoid the risk of potentially exposing the model at learning step to unseen testing images, test data is stored in a separate directory and only to be used for inferencing. The following directory structure is required for YOLOv4 training.

```
../datasets/train/img1.jpg  # image
../datasets/train/img1.txt  # label
../datasets/valid/img1.jpg  # image
../datasets/valid/img1.txt  # label
```

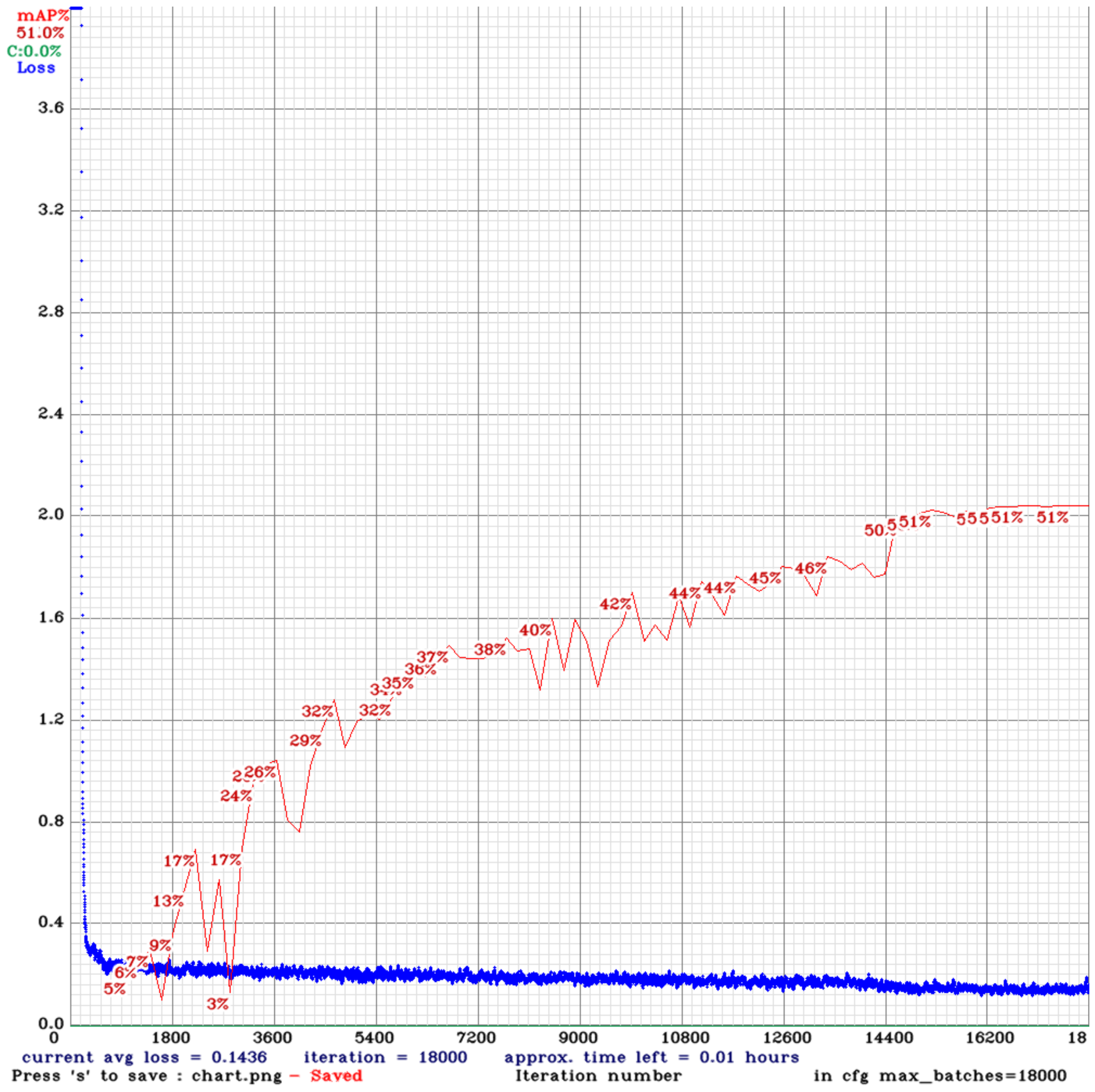
The following command-line tool argument started the training process for 18,000 iterations.

```
./darknet detector -map -dont_show train /home/usr/NeuralNets/YOLO/Yolov4-RCW.data
                    /home/usr/NeuralNets/YOLO/Yolov4-RCW.cfg
```

The following training scenarios are separated into individual sections. Figures 24 to 27 display training performance on individual parts of the dataset i.e., dataset (1) and dataset (2) separately, while Figures 30 and 32 display training performance on the merged dataset.

Figure 24

Scenario 1 using 534 Images taken at 76m Elevation

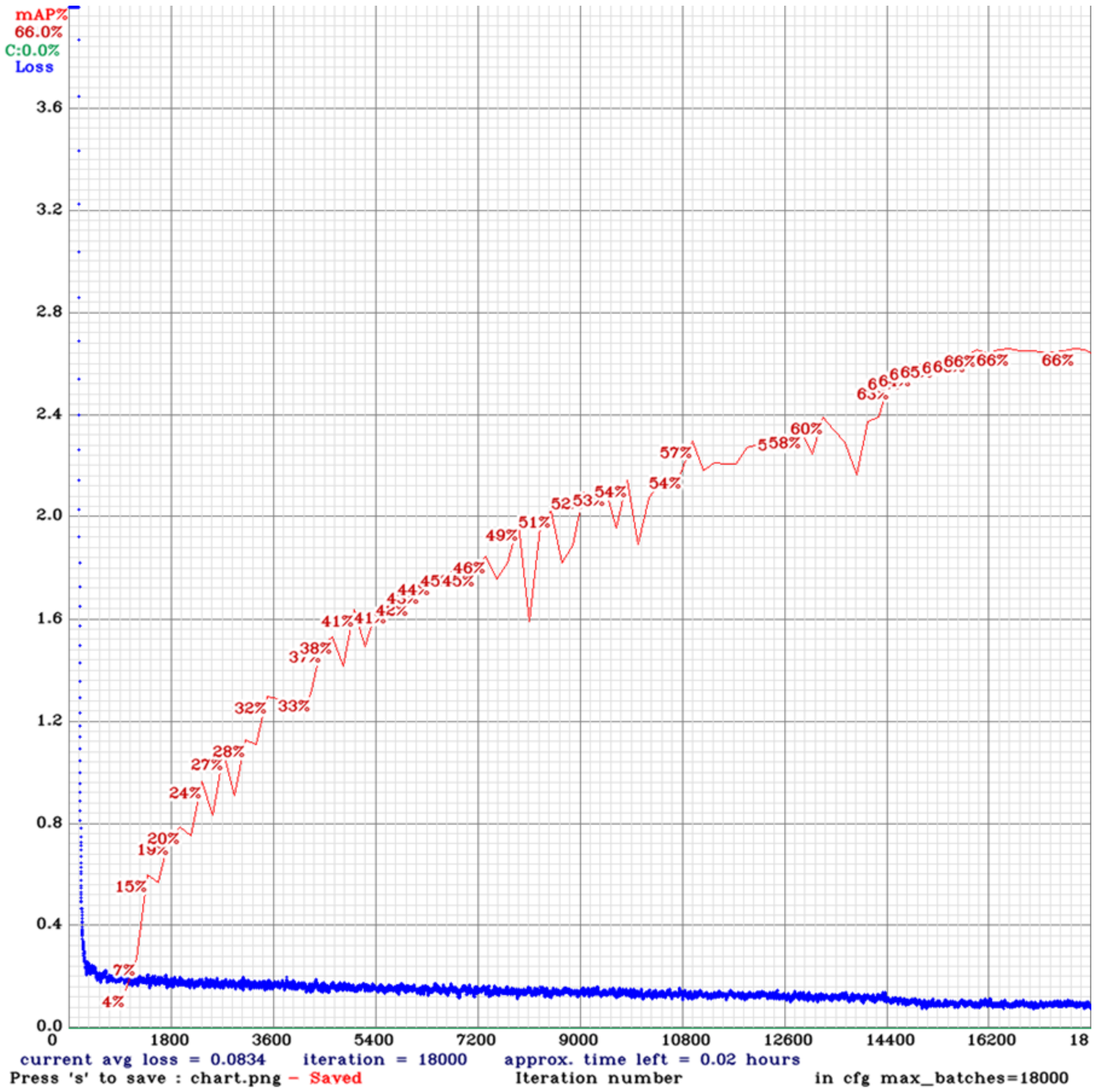


Note. The initial mAP, 0.51, was relatively poor and unsatisfying. The learning curve was far from smooth, and the training loss could be improved further. This result was due to a number of reasons, including relatively small training time, poor annotation box drawing, poor image quality, and small dataset size of 534 images with a large variance in object appearance.

Scenario 2

Figure 25

Scenario 2 using 534 Images taken at 76m Elevation

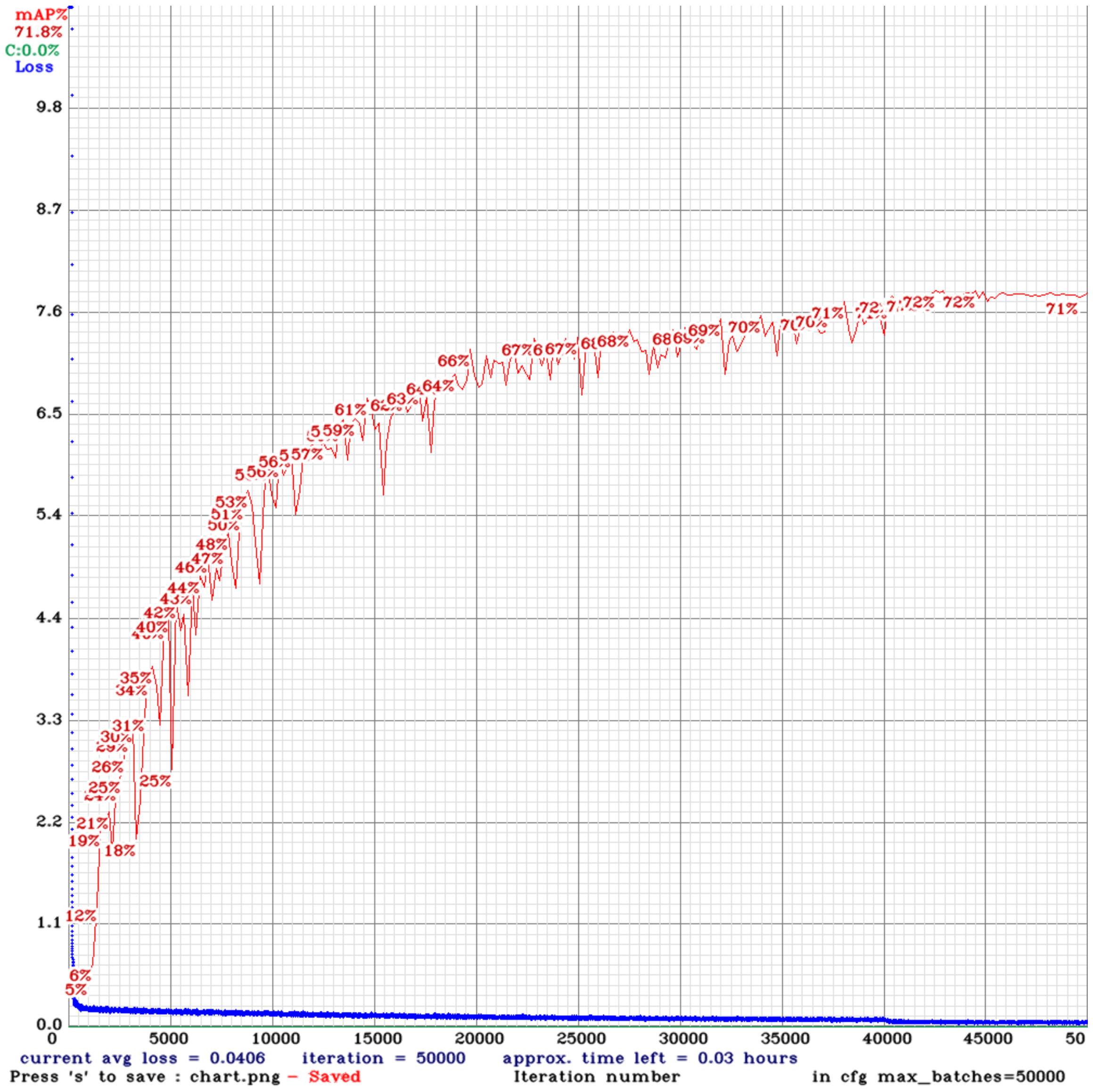


Note. To improve the results of scenario one, 23 relatively bad samples from the dataset were removed to enhance the mAP. The training parameters did not change for the second iteration. Removing bad examples worked to some extent, as it raised mAP by 14%. However, 'cherry-picking' is an inappropriate practice and should not be done. Diversity in object appearance is crucial to training a robust model.

Scenario 3

Figure 26

Scenario 3 using 534 Images taken at 76m Elevation



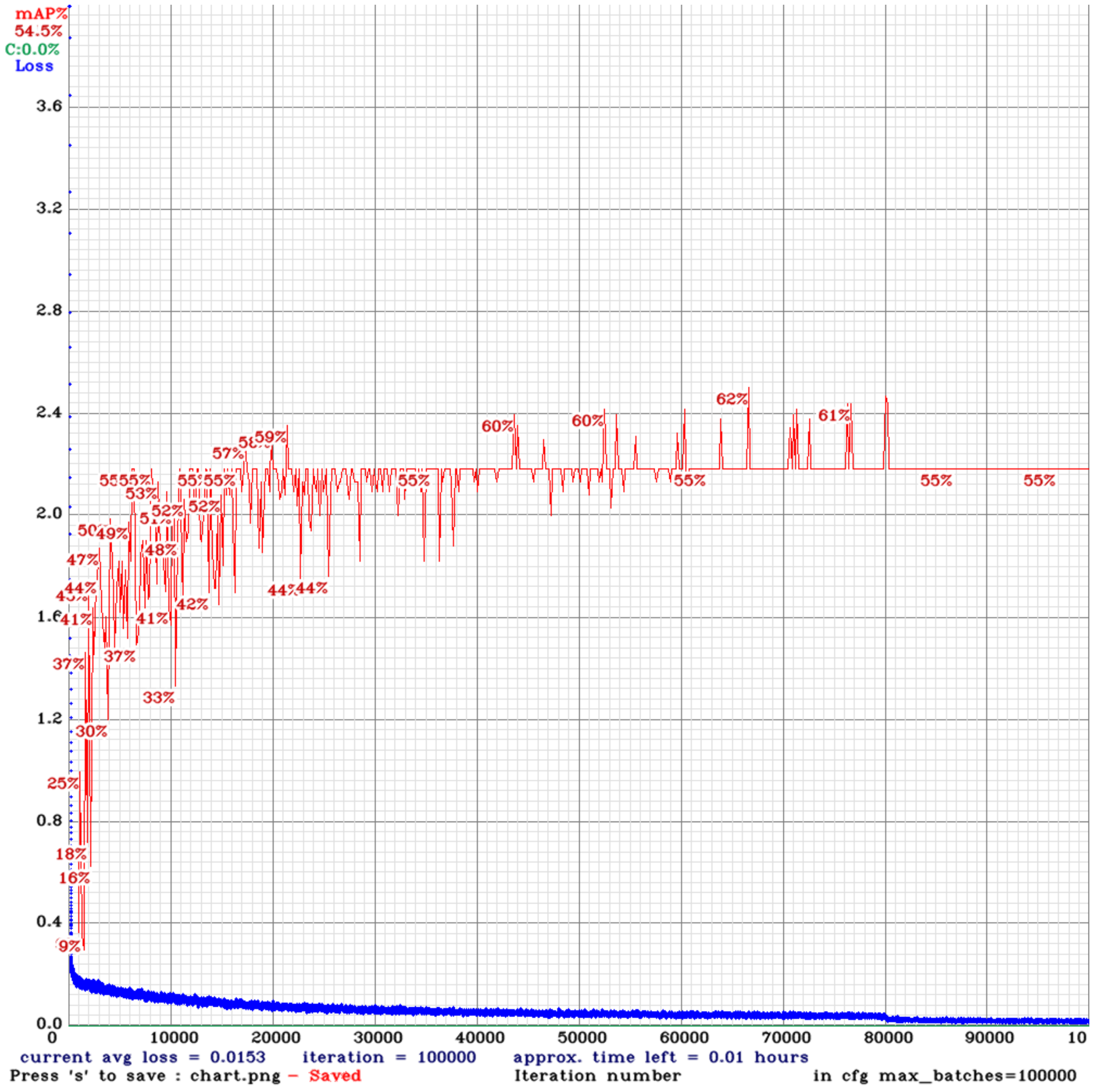
Note. Since removing bad quality samples is not a reasonable practice, they were put back in the pool annotated cavities. Another way to improve mAP is to increase the iteration number from 18,000 to 50,000 iterations. Most YOLOv4 training samples found by (Bochkovskiy, 2020) showed a standard iteration size of 10,000. Due to the nature of this object, training length reached 5× the normal range.

Scenario 4

The previous three scenarios demonstrated that all samples should be included in a training run, and better results were starting to appear at 100,000 iterations. The fourth scenario of training used only the second batch of 444 images taken at 46m altitude data. This scenario is designed to assess how altitude change affects accuracy performance. With a much lower altitude, better results were logically expected.

Figure 27

Scenario 4 using 444 Images taken at 46m Elevation

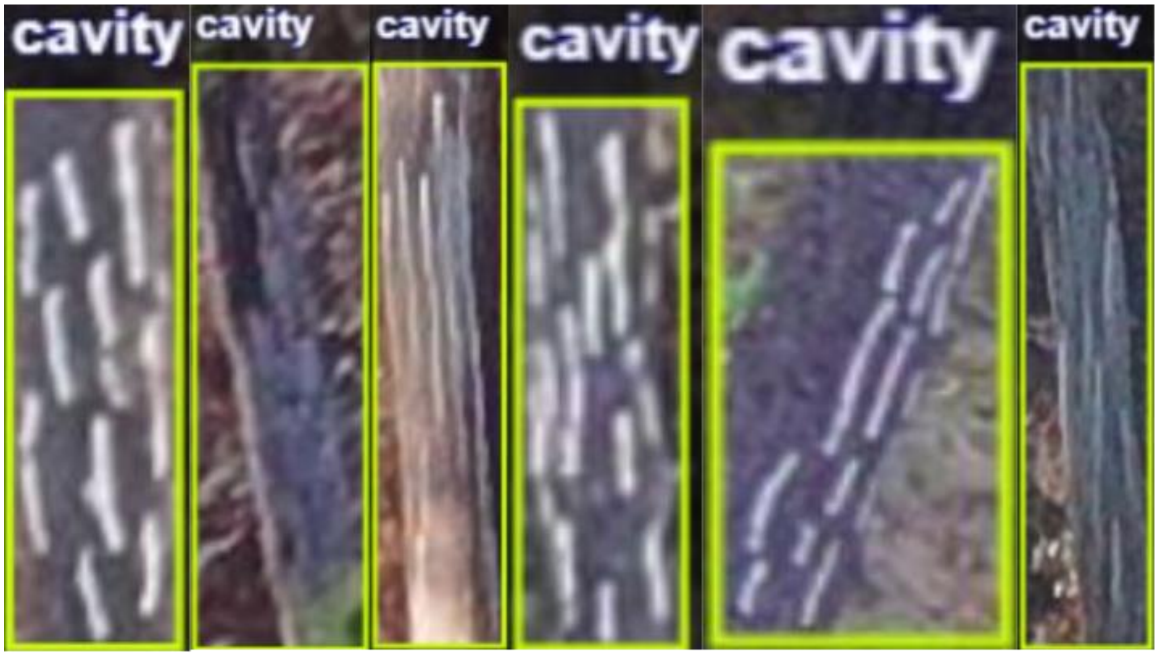


Annotation Technique

Contrary to our belief, as shown in Figure 27, the second batch of 46m altitude images performed worse than the first batch of 76m altitude images. This was caused by inefficient bounding box annotation and attempting to include objects that were not resin samples. Figure 28 displays the proper annotation technique, and Figure 29 displays the improper annotation technique.

Figure 28

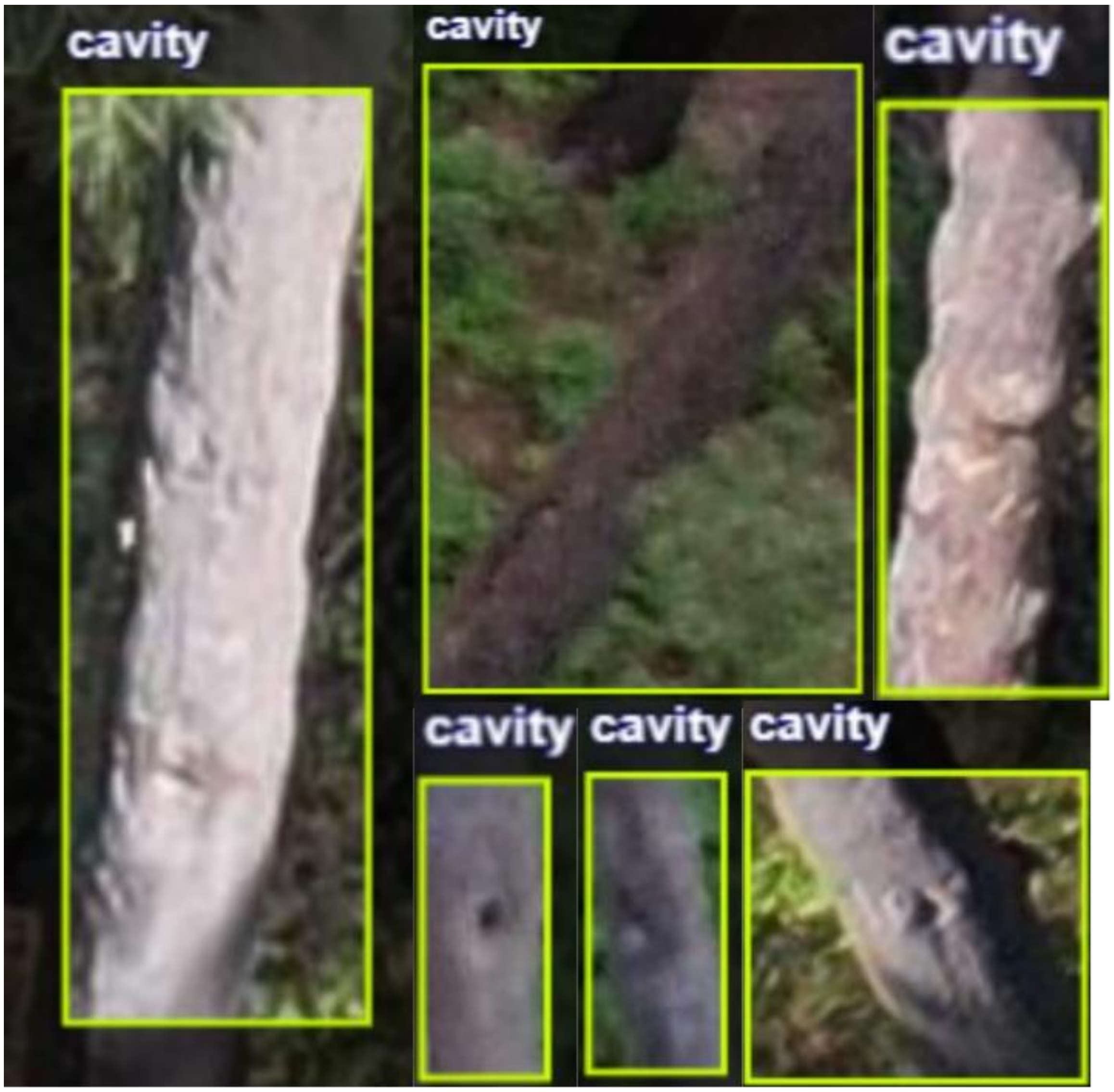
Proper Annotation to Capture Pine Resin Cavity Markers



Note. Above is an example of a proper annotation. Annotation boxes should be drawn with the tightest borders possible. Including too much background and irrelevant features will cause a model to learn unnecessary features and will skew its learning process. Note that Open Images Dataset V6 demonstrates more excellent examples of properly drawn annotation boxes.

Figure 29

Improper Annotation Featuring Incorrectly Labeled Objects



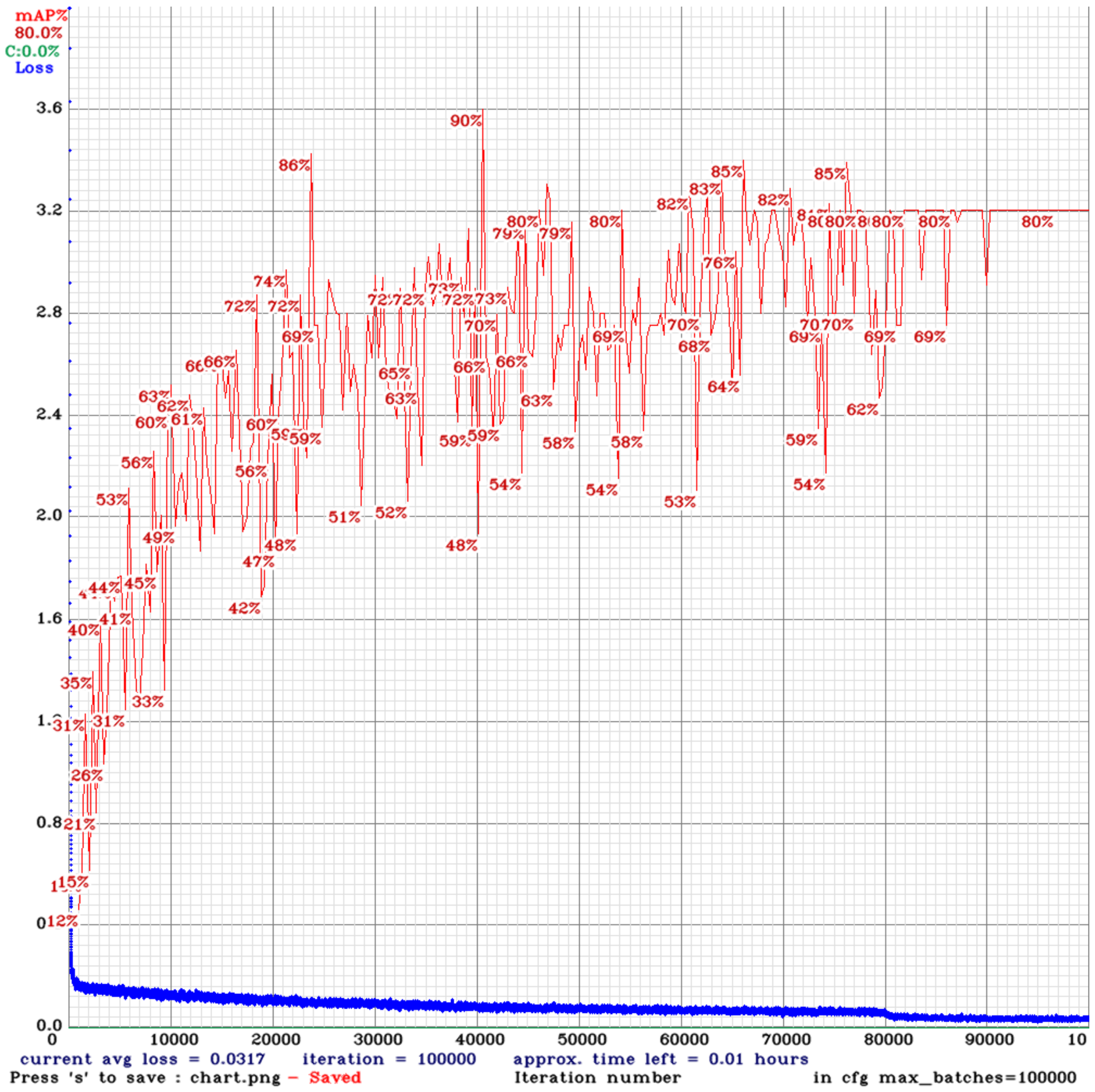
Note. These are ambiguous images that do not clearly illustrate what a pine resin cavity should look like. The bounding boxes do not tightly define the object of interest, and too many irrelevant features are present in these images for a model to fully learn. Bounding boxes like these were removed from the dataset altogether.

Merging Datasets

After observing the combined results of each iteration, the next step is to merge both datasets. Merging both datasets resulted in 978 images used for train and test. Figure 30 displays the training performance with the merged dataset.

Figure 30

Scenario 5 Iteration Merging Batch 1 and Batch 2 Datasets



```
current avg loss = 0.0317   iteration = 100000   approx. time left = 0.01 hours
Press 's' to save : chart.png - Saved            Iteration number        in cfg max_batches=100000
```

Note. Merging both datasets resulted in a classic example of overfitting. Plenty of abnormal outliers skew the training results. This image demonstrates why calculating of custom anchors is beneficial. The model is recorded as having the 'best' mAP of 0.90, however this is an outlier.

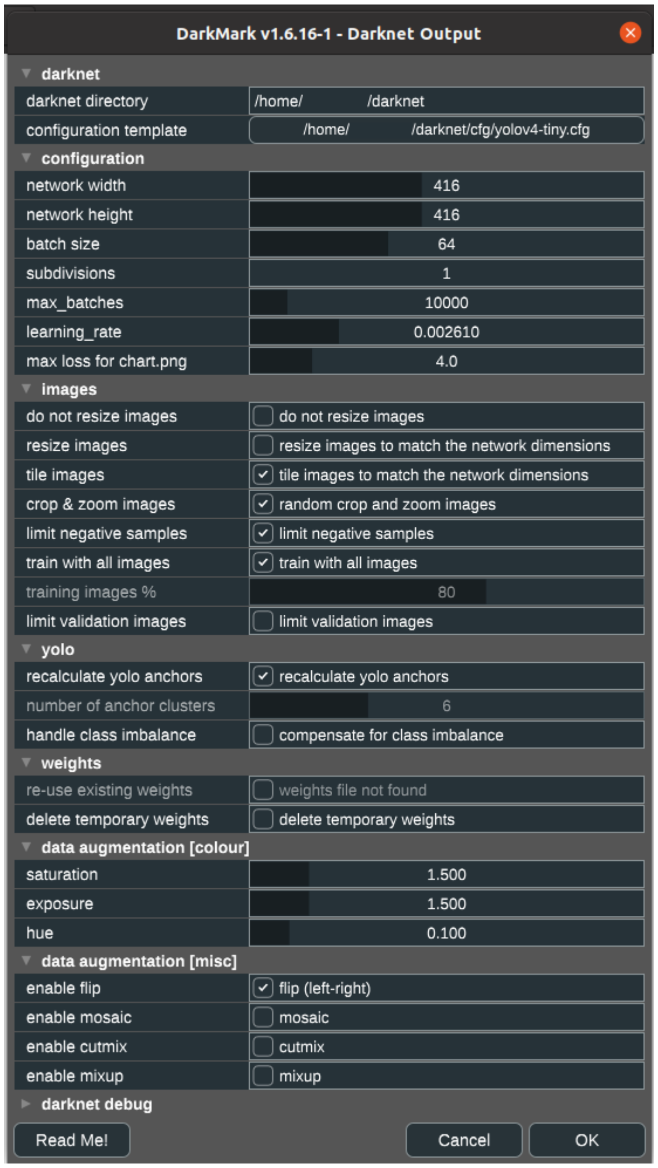
Final Optimal Scenario

The previous scenarios slowly updated the configuration file to find an optimal setting. Network input dimensions, batch size, subdivision, limiting validation images, recalculating bounding box anchors, and various miscellaneous data augmentations were performed and eventually led to finding the best results out of 100,000 iterations using the yolov4-tiny configuration.

A few important things to highlight with YOLOv4, batch size and subdivision size, require careful implementation. Ideally, batch size should be a multiple of subdivision size and should be slowly incremented to avoid GPU memory issues. With a small dataset size, i.e., under a few thousand samples, training with all images should be toggled on with a default 80% train, 20% validation split. Most importantly, when training with a novel object such as pine resin, recalculating yolo anchors must be enabled; this allows the model to adjust and fine-tune boundary boxes during training. The default anchor values will be used without recalculating these anchors, which are trained on MS COCO or ImageNet data. Data augmentation should be tested differently with each data set as there is no standard rule for using it. For this dataset, a vertical flip was most helpful as pine resin has some degree of vertical symmetry. Using the configuration settings shown in Figure 31, the best mAP of .95 was achieved; the final result is displayed in Figure 32.

Figure 31

Optimal Settings after Trial and Error



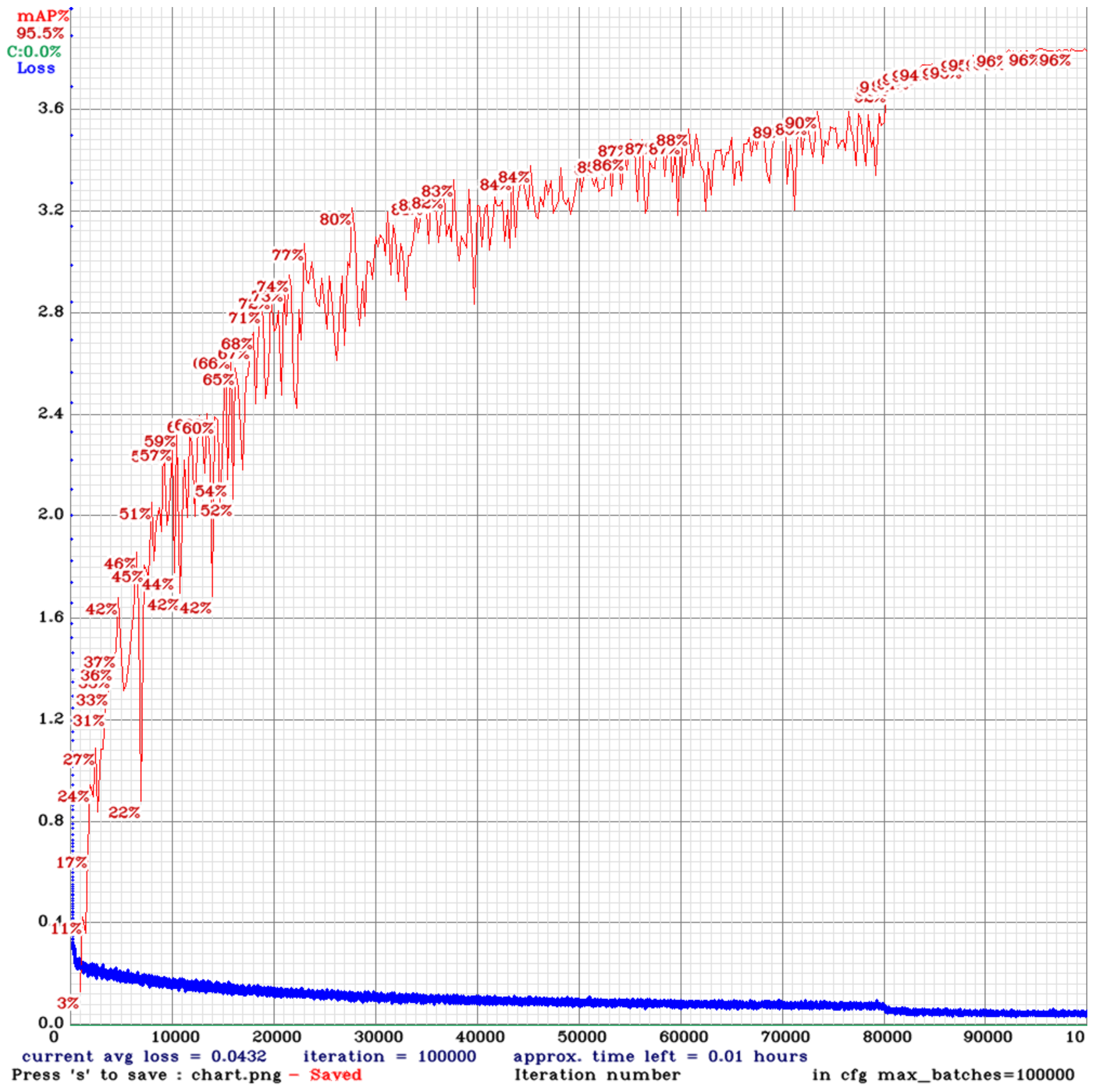
DarkMark v1.6.16-1 - Darknet Output

▼ darknet	
darknet directory	/home/ /darknet
configuration template	/home/ /darknet/cfg/yolov4-tiny.cfg
▼ configuration	
network width	416
network height	416
batch size	64
subdivisions	1
max_batches	10000
learning_rate	0.002610
max loss for chart.png	4.0
▼ images	
do not resize images	☐ do not resize images
resize images	☐ resize images to match the network dimensions
tile images	✓ tile images to match the network dimensions
crop & zoom images	✓ random crop and zoom images
limit negative samples	✓ limit negative samples
train with all images	✓ train with all images
training images %	80
limit validation images	☐ limit validation images
▼ yolo	
recalculate yolo anchors	✓ recalculate yolo anchors
number of anchor clusters	6
handle class imbalance	☐ compensate for class imbalance
▼ weights	
re-use existing weights	☐ weights file not found
delete temporary weights	☐ delete temporary weights
▼ data augmentation [colour]	
saturation	1.500
exposure	1.500
hue	0.100
▼ data augmentation [misc]	
enable flip	✓ flip (left-right)
enable mosaic	☐ mosaic
enable cutmix	☐ cutmix
enable mixup	☐ mixup
► darknet debug	

Note. Important things to note with this configuration. Recalculation of YOLO anchors was imperative to enable, either manually or through DarkMark, the software running this configuration, as it impacted training performance the most (Solawetz, 2020). Due to the small size of this dataset, training with all images and not limiting validation image augmentation was essential.

Figure 32

Optimal Performance using YOLOv4-tiny with the Merged Dataset



Note. The irregular object appearance of pine resin makes certain outliers inevitable. However, the training curve is relatively smooth compared to previous scenarios.

Test Results

The final optimal scenario of 0.95 training mAP produced the best model. Hence, this version of the trained model was used to inference test data. There are many techniques, commands, and software packages for inferencing test data. This particular model used DarknetServer API with 416x416 tiles. Figure 33 displays the confusion matrix created from this process.

Figure 33

Confusion Matrix of YOLOv4-tiny

		Actual Values	
		1	0
Predicted Values	1	130	37
	0	8	0

Note. This result is recorded using 103 testing images. The test images are all unique and unseen at the training step. The high false-positive cases (i.e., 37) can be explained by a relatively low threshold of 0.30. This is done to have the most bounding box predictions possible. In this particular problem, false positives are preferred because wildlife biologists would rather have more possible identifications instead of missing cased (i.e., false negatives).

Table 10 summarizes the performance of YOLOv4-tiny, and Figures 34 and 35 demonstrate the performance on 76m and 46m test data.

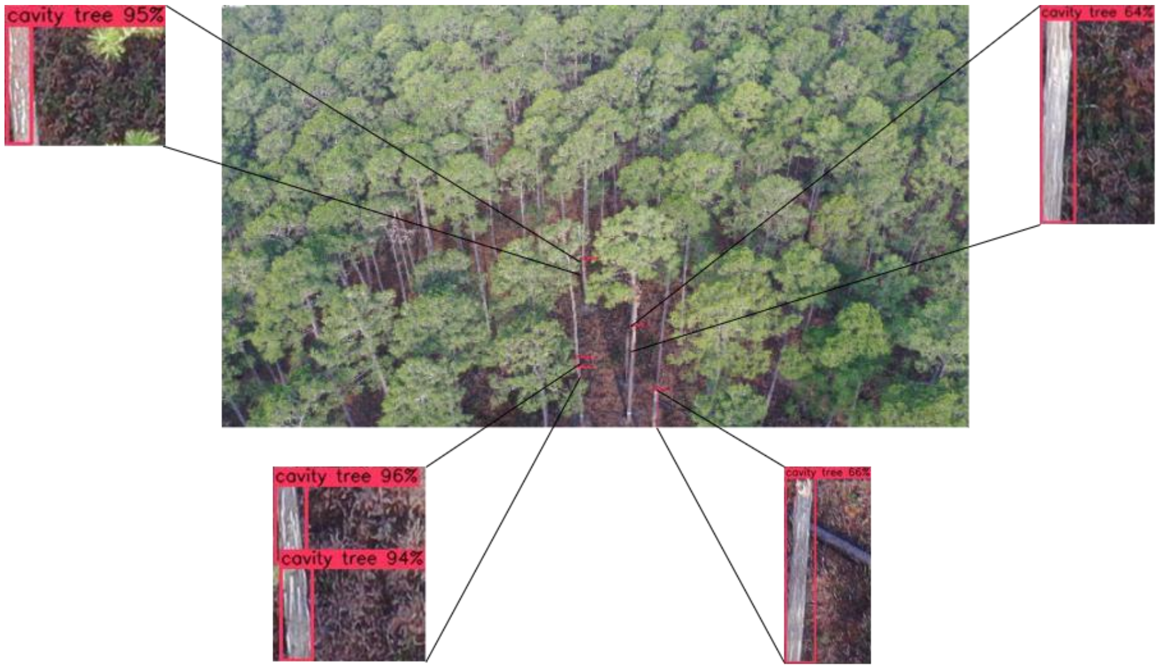
Table 10

Detailed Performance of YOLOv4-tiny

Metric	Definition	Equation	Result
TP	Positive values predicted positive	-	130
FP	Negative values predicted positive	-	37
FN	Positive values predicted negative	-	8
T.N.	Negative values predicted negative	-	0
Accuracy	Proportion of correct predictions out of all predictions made	(TP + TN) / (P + N)	0.7429
Precision	Proportion of correct predictions out of all positive predictions	(TP) / (TP+FP)	0.7784
Recall (Sensitivity or TPR)	Proportion of correct predictions out of all positive classes	(TP) / (TP + FN)	0.942
F1-Score	Test accuracy, weighted average of precision and recall. 1 best, 0 worst.	2*(Precision*Recall) / (Precision +Recall)	0.8525
Specificity (TNR)	Probability that a negative prediction will be true	(TN) / (TN+FP)	0
False Negative Rate (FNR)	Probability of labeling a negative class as positive	(FN) / (FN + TP)	0.05797
False Positive Rate (FPR)	Probability of labeling a positive class as negative	(FP) / (FP + TN)	1.00
False Discovery Rate (FDR)	Ratio of FP to total number of positive predictions	(FP) / (FP + TP)	0.2216

Figure 34

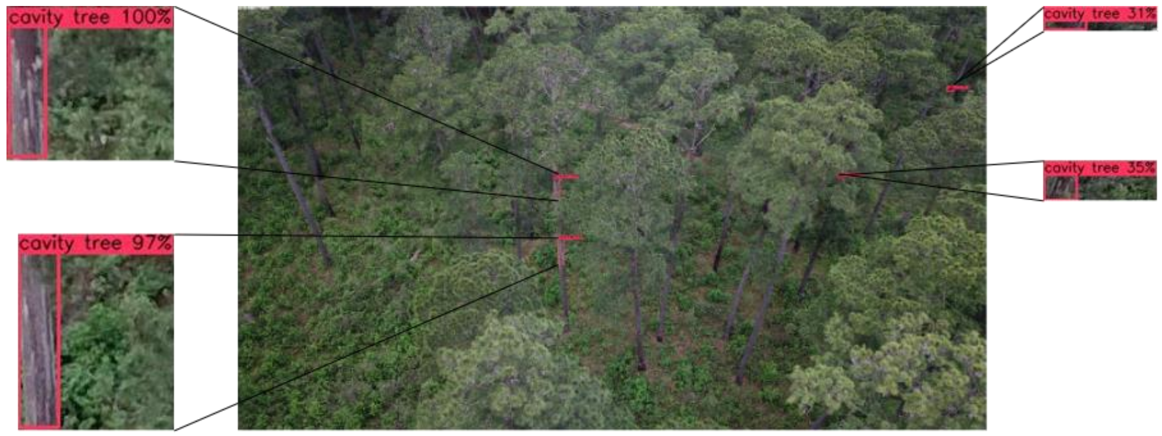
Sample Inference with 76m Altitude Test Data



Note. These inference results were performed with YOLOv4-tiny.weights realized with the final training scenario. To inference this, the following command was run: DarkHelp –keep –tiles on –autohide off – YOLOv4-tiny {.cfg, .names, best.weights} test_dir. Nearly all of the cavities in this image were detected with very high accuracy. Variance is normal as shown in the bottom right cavity with 66% confidence, which is expected with many single-shot detectors.

Figure 35

Sample Inferencing with 46m Altitude Test Data



Note. These inference results are gathered in the same fashion as Figure 34. Note two cavities on the right side were too small or obscured by foliage to be identified with large confidence. The detection threshold was set relatively low at 0.30 as there is a functional preference to consider false positives for potential cavities. These low-confidence cases can by unidentified by setting the threshold to be 0.50, which is the standard default detection threshold.

YOLOv5n

YOLOv5n is the second model of choice for this study. Due to technical constraints, this model did not perform as well as YOLOv4-tiny. All parameters for training YOLOv5n were the same as YOLOv4-tiny except for network input dimension. YOLOv5n requires a recommended input dimension of 640×640 pixels, which reduces the average annotation box apparent size to ~0.333% of the tiled from ~0.789% as featured in YOLOv4-tiny. Training for 300 epochs, YOLOv5n reached a training mAP of 78%, which is much lower than YOLOv4-tiny. Figure 36 suggests that the increased epoch size might achieve better mAP values. However, this 300 epoch run took 6 days and 13 hours to complete; thus, increasing the epoch length to 1,000 would be unfeasible and unreliable to keep a workstation online without interruption. Figure 37 visualizes the class distribution and the overall imprint of the average annotation box label.

The YOLOv5 family formats training inputs are slightly different compared to previous models. The expected format is as follows.

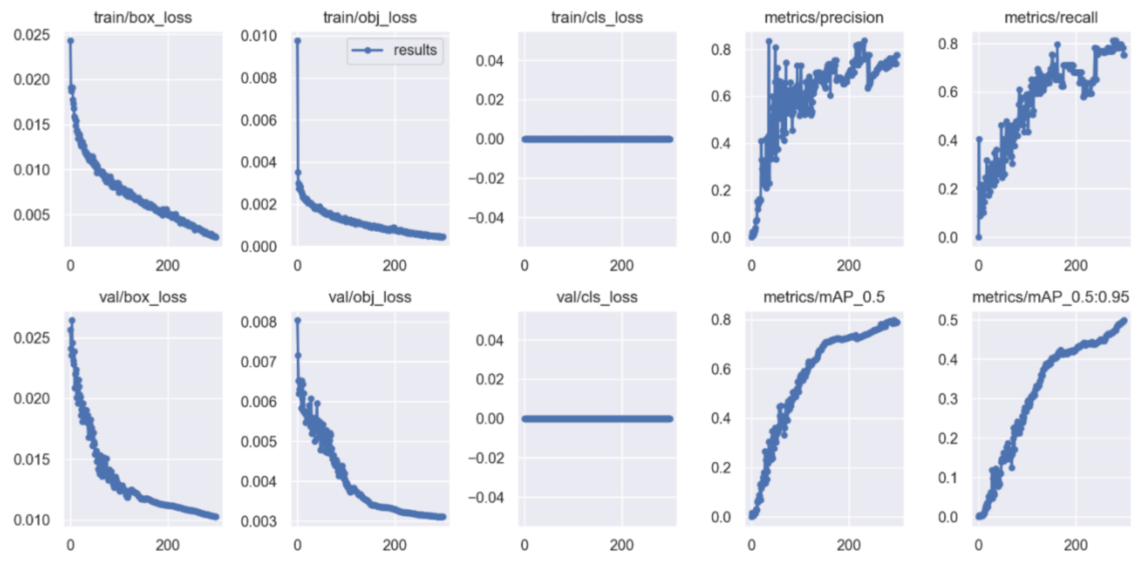
```
../datasets/train/images/img1.jpg  # image
../datasets/train/labels/img1.txt  # label
../datasets/valid/images/img1.jpg  # image
../datasets/valid/labels/img1.txt  # label
```

The corresponding training command was used to start the training process for 300 epochs in an Anaconda environment as follows.

```
python3 C:/yolov5-gpu/yolov5-master/train.py --img 640 --batch 3 --epochs 300 --data
  C:/yolov5-gpu/yolov5-master/data/RCW_custom.yaml --weights yolov5n.pt --device 0
```

Figure 36

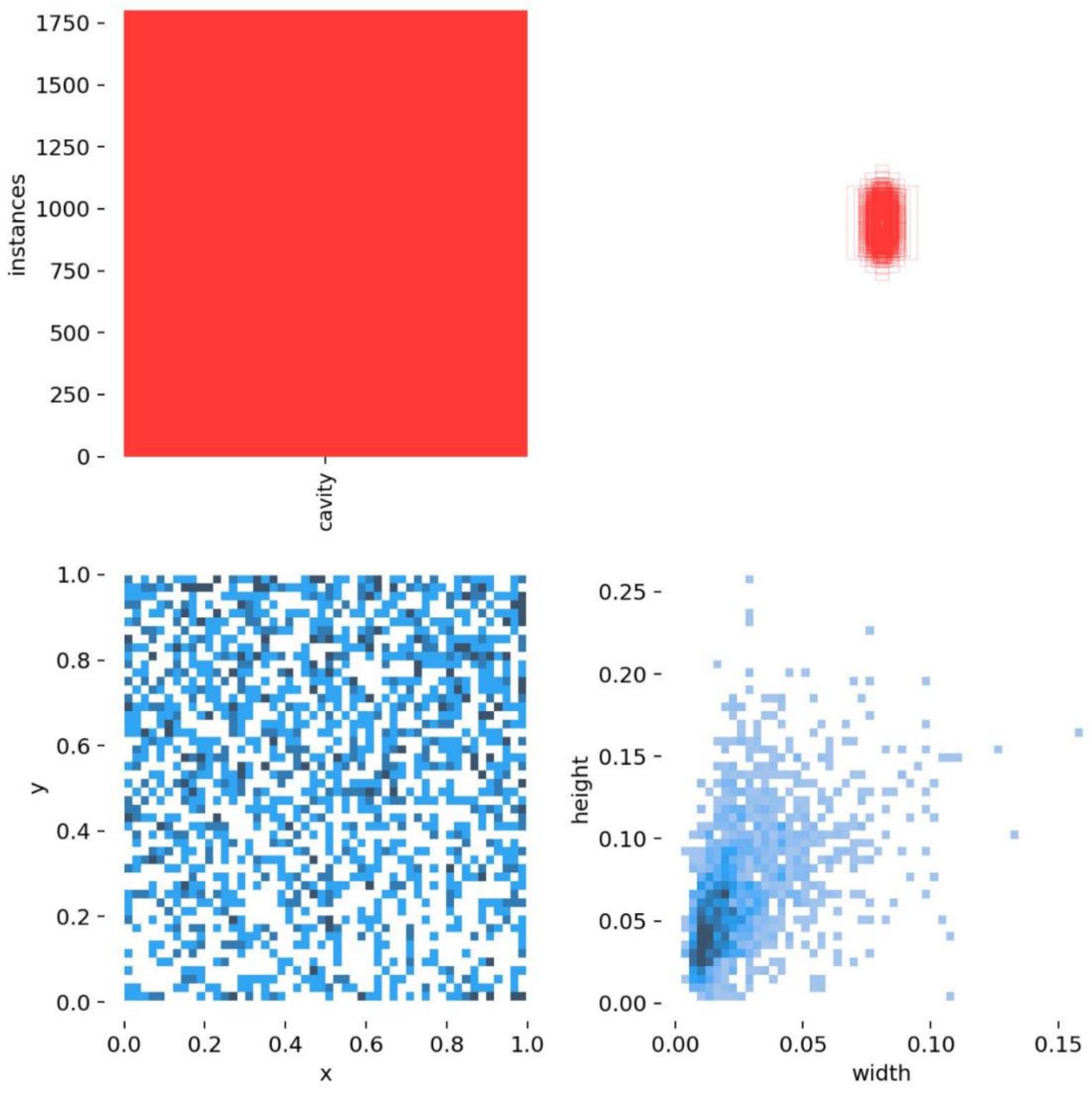
Performance of YOLOv5n with Batch 1 Data (Top) and Batch 2 Data (Bottom)



Note. YOLOv5 uses Results are generated in WandB.ai; the service YOLOv5 uses to automatically store results online. mAP_0.5 was 0.78.

Figure 37

Distribution of Class Instances in YOLOv5 Output Result



YOLOv5 automatically deploys its auto-anchoring during training. In addition, AutoAnchor enables the fine-tuning of the expected anchor box values. Figure 38 demonstrates the test confusion matrix recorded by YOLOv5, and Table 11 further defines the performance metrics.

Test Results

Figure 38

YOLOv5n Confusion Matrix

		Actual Values	
		1	0
Predicted Values	1	92	2
	0	39	0

Note. The same test images used by YOLOv4-tiny were also used for YOLOv5. The test images are all unique and unseen at the training step. This model displayed more false-negative identification and much fewer false positives.

Table 11

YOLOv5n Confusion Matrix Results

Metric	Definition	Equation	Result
TP	Positive values predicted positive	-	92
FP	Negative values predicted positive	-	2
FN	Positive values predicted negative	-	39
TN	Negative values predicted negative	-	0
Accuracy	Proportion of correct predictions out of all predictions made	(TP + TN) / (P + N)	0.6917
Precision	Proportion of correct predictions out of all positive predictions	(TP) / (TP+FP)	0.7023
Recall (Sensitivity or TPR)	Proportion of correct predictions out of all positive classes	(TP) / (TP + FN)	0.9787
F1-Score	Test accuracy, weighted average of precision and recall. 1 best, 0 worst.	2*(Precision*Recall) / (Precision +Recall)	0.8178
Specificity (TNR)	Probability that a negative prediction will be true	(TN) / (TN+FP)	0
False Negative Rate (FNR)	Probability of labeling a negative class as positive	(FN) / (FN + TP)	0.0212
False Positive Rate (FPR)	Probability of labeling a positive class as negative	(FP) / (FP + TN)	1.00
False Discovery Rate (FDR)	Ratio of FP to total number of positive predictions	(FP) / (FP + TP)	0.2977

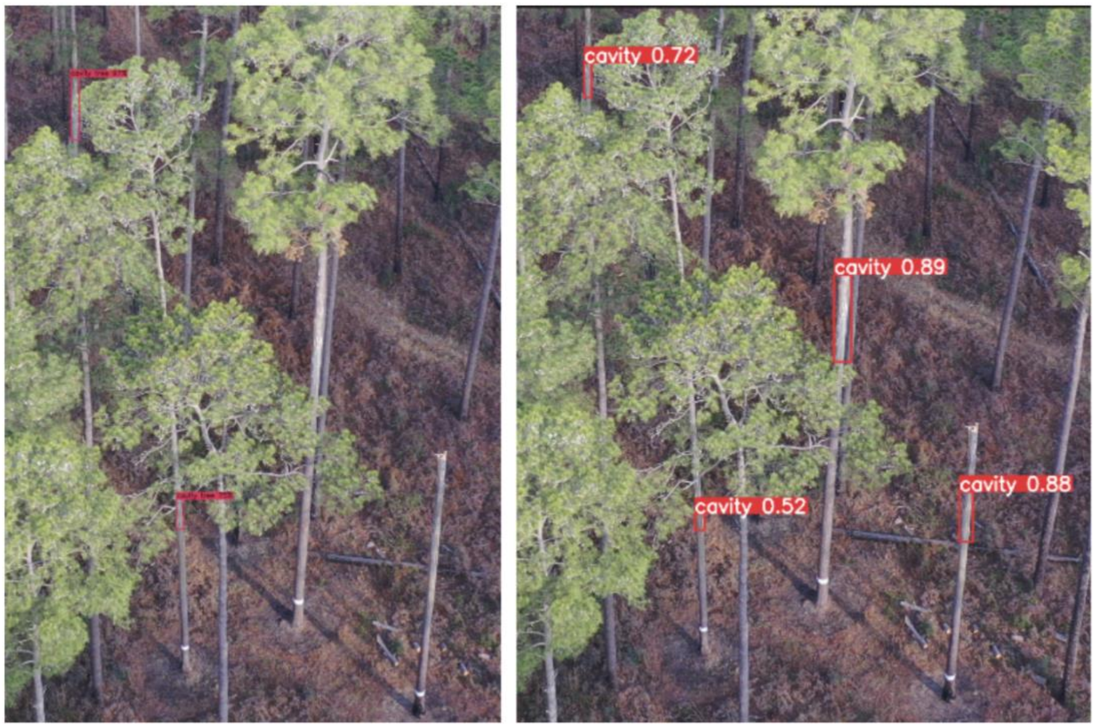
Comparison between YOLOv4-tiny and YOLOv5n

The F1-scores are used to assess a model's testing accuracy. Both models tiled images. The input image size is 416×416 for YOLOv4-tiny and 640×640 for YOLOv5n. YOLOv4-tiny ran inferencing using Darknet Detector, while YOLOv5n used Sliced Aided Hyper Inferencing (SAHI) created by (Akyon, 2021). Running both models at a threshold of 0.30. The F1-score for test data is 0.8525 for YOLOv4-tiny and 0.8178 for YOLOv5n. They are quite comparable in terms of overall performance. However, training mAP was quite different. YOLOv4-tiny achieved an mAP of 0.96, and YOLOv5n achieved an mAP of 0.78 for training data. A visual comparison is provided in Figure 39.

Numerically, YOLOv5n made 46 better predictions compared to YOLOv4-tiny. On the other hand, YOLOv4-tiny made 37 better predictions compared to YOLOv5n. However, test results showed that both models complemented each other. Various RCW markers that one model missed were detected by the other. Using both models for inferencing and combining their results, the cavity detection 1.0 (i.e., 100%) on the testing data was achieved. Not a single cavity was overlooked or missed. In practice, this is a massive benefit for wildlife ecologists to ensure a robust detection system. Below are a few samples that compare YOLOv4-tiny and YOLOv5n.

Figure 39

Sample YOLOv4-tiny Testing Inference Compared with YOLOv5n



Note. This figure clearly shows an example of difference in detecting RCW habitats between YOLOv4-tiny (left) and YOLOv5n (right). Using SAHI, which is only available for YOLOv5 and newer models, detections are made with any desired tiling size. Overall, YOLOv5n found 46 more cavity instances than YOLOv4-tiny.

CHAPTER VI

Conclusion and Future Work

For the preservation of red-cockaded woodpecker (RCW), this work implemented two computer vision models and performed a comparison to find the best solution to assist in automating the RCW habitat detection process. This process saves thousands of personnel hours and reduces dangers associated with pedestrian surveying in remote locations. Computer vision, a machine learning subfield, promises to expand the scope of many ecological conservation projects. YOLOv4-tiny reached a mean average precision (mAP) of 0.95 and test accuracy of 0.85, and YOLOv5n achieved a mAP of 0.78 and test accuracy of 0.82 using low fidelity data. Although the result is respectable, more can be done to improve testing performance and deploy a real-time detection system.

Data

The data set featured in this work is small in quantity, contains too much variation, and is not robust. The dataset needs to be increased by a few orders of magnitude to have better effects on model training. Simple models trained on large data sets generally perform better than complex models trained on small data sets. When expanding the dataset, care should be taken to include different kinds of pine resin appearances, seasonal variation, time of day, and increased geographic distribution. The current dataset suffers from a lack of reliability. A reliable RCW dataset should consider proper annotation technique, deletion of all duplicate, low-resolution images should be omitted, and creation of a multi-class structure if ecologists wish to classifiy other types of RCW habitats, i.e. non-sap.

Demanding Computing Hardware

The hardware used for this project had limitations. The maximum training image size was 736×736 pixels before running out of GPU memory. Increasing memory availability alleviates this problem however, GPUs are often costly and out of reach for most practitioners. An alternative to buying local hardware is using GPU cloud services that offer free training capabilities. However, they are often paywalled and are very slow compared to local hardware.

With more excellent hardware capabilities, larger models that promise better mAP values can be trained and tested. Unfortunately, YOLOR, YOLOX, YOLOv5, and YOLOv4 all have large or extra-large model configurations that require enormous memory requirements.

Model Ensembling

This work set out to perform model ensembling using the YOLO family. Using various models, working together helps validate the results. Model ensembling for real-time detection may be tricky but not impossible. In an ideal use case, the extra-large models that promise a higher mAP value should be used to infer one data point simultaneously.

The work featured in this study can be implemented in other ecological projects. Large and small animal species identification, animal population counting, habitat detection, territory segmentation, forest categorization, tree health measurements, etc. Implementing computer vision is becoming much more accessible in recent years due to artificial intelligence democratization and the streamlined approach model developers take when releasing the latest innovations.

REFERENCES

Akyon, F. C. (2022, January 13). *SAHI: A vision library for large-scale object detection & instance segmentation*. Medium. https://medium.com/codable/sahi-a-vision-library-for-performing-sliced-inference-on-large-images-small-objects-c8b086af3b80

Atanbori, J., Duan, W., Murray, J., Appiah, K., & Dickinson, P. (2016). Automatic classification of flying bird species using computer vision techniques. *Pattern Recognition Letters*, *81*, 53–62. https://doi.org/10.1016/j.patrec.2015.08.015

Berg, T., Liu, J., Lee, S. W., Alexander, M. L., Jacobs, D. W., & Belhumeur, P. N. (2014). Birdsnap: Large-scale fine-grained visual categorization of birds. *2014 IEEE Conference on Computer Vision and Pattern Recognition*. https://doi.org/10.1109/cvpr.2014.259

Bochkovskiy, A., Wang, C., & Liao, H. (2020). YOLOv4: Optimal speed and accuracy of object detection. *arXiv.Org.* https://arxiv.org/abs/2004.10934?

Bochkovskiy, A. (2020a). *GitHub - AlexeyAB/darknet: YOLOv4 / Scaled-YOLOv4 / YOLO - Neural networks for object detection (Windows and Linux version of Darknet)*. GitHub. https://github.com/AlexeyAB/darknet

Butler, M. J., & Tappe, P. A. (2008). Relationships of red-cockaded woodpecker reproduction and foraging habitat characteristics in Arkansas and Louisiana. *European Journal of Wildlife Research*, *54*(4), 601–608. https://doi.org/10.1007/s10344-008-0184-9

Brownlee, J. (2021, January 26). *A gentle introduction to object recognition with deep learning*. Machine Learning Mastery. https://machinelearningmastery.com/object-recognition-with-deep-learning/

Carrascal, L. M., Galván, I., Sánchez-Oliver, J. S., & Rey Benayas, J. M. (2013). Regional distribution patterns predict bird occurrence in Mediterranean cropland afforestations. *Ecological Research*, *29*(2), 203–211. https://doi.org/10.1007/s11284-013-1114-1

Chien, W., Bochkovskiy, A., & Liao, H. (2020, November 16). *Scaled-YOLOv4: Scaling cross stage partial network*. arXiv.Org. https://arxiv.org/abs/2011.08036

Charette, S. (2020, August 22). *Darknet, DarkMark tutorial for Ubuntu* [video]. YouTube. https://www.youtube.com/watch?v=RcLL8Lea6Ec&t

Charette, S. (2020, August 22) *Programming comments - Darknet FAQ*. Ccoderun. https://www.ccoderun.ca/programming/darknet_faq/

Dedhia, P. R. (2022, March 30). *Understanding SPPNet for object classification and detection*. Medium. https://towardsdatascience.com/understanding-sppnet-for-object-detection-and-classification-682d6d2bdfb

Elgendy, M. (2020). *Deep learning for vision systems* (1st ed.). Manning.

Fukushima, K., & Wake, N. (1991). Handwritten alphanumeric character recognition by the neocognitron. *IEEE Transactions on Neural Networks, 2*(3), 355–365. https://doi.org/10.1109/72.97912

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*. https://doi.org/10.1109/cvpr.2014.81

Gregory, R. D., & Strien, A. V. (2010). Wild bird indicators: Using composite population trends of birds as measures of environmental health. *Ornithological Science*, *9*(1), 3–22. https://doi.org/10.2326/osj.9.3

Hubel, D. H., & Weisel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *The Journal of Physiology*. 148, 574-91. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1363130/

Jackson, J. A. (1971). The evolution, taxonomy, distribution, past populations, and current status of the red-cockaded woodpecker. *Symposium on the Red-Cockaded Woodpecker*. 4-29. U.S. Bureau of Sport Fisheries and Wildlife.

Jin, L., & Liang, H. (2017). Deep learning for underwater image recognition in small sample size situations. *OCEANS 2017 - Aberdeen*. https://doi.org/10.1109/oceanse.2017.8084645

Konovalov, D. A., Saleh, A., Bradley, M., Sankupellay, M., Marini, S., & Sheaves, M. (2019). Underwater fish detection with weak multi-domain supervision. *2019 International Joint Conference on Neural Networks (IJCNN)*. https://doi.org/10.1109/ijcnn.2019.8851907

Kotu, V., & Deshpande, B. (2015). *Predictive analytics and data mining*. Elsevier Gezondheidszorg.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386

Krohn, J., Beyleveld, G., & Bassens, A. (2020). *Deep learning illustrated: A visual, interactive guide to artificial intelligence (Addison-Wesley Data & Analytics Series)* (1st ed.). Addison-Wesley Professional.

Landers, J. L., Van Lear, D. H., & Boyer, W. D. (1995). *The longleaf pine forests of the southeast: requiem or renaissance? | Treesearch*. Journal of Forestry. https://www.fs.usda.gov/treesearch/pubs/980

Lawrence, B. (2022). Classifying forest structure of Red-Cockaded Woodpecker habitat using structure from motion elevation data derived from sUAS imagery. *Drones*, *6*(1), 26. https://doi.org/10.3390/drones6010026

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Leonard, D. L., & DeLotelle, R. S. (2003). [Review of *The Red-Cockaded Woodpecker: Surviving in a fire-maintained ecosystem*, by Richard N. Conner D. Craig Rudolph Jeffery R. Walters]. *The Auk*, *120*(4), 1201–1205. https://doi.org/10.2307/4090293

Mellit, A., Massi Pavan, A., Ogliari, E., Leva, S., & Lughi, V. (2020). Advanced methods for photovoltaic output power forecasting: A review. *Applied Sciences*, *10*(2), 487. https://doi.org/10.3390/app10020487

Miao, Z., Gaynor, K. M., Wang, J., Liu, Z., Muellerklein, O., Norouzzadeh, M. S., McInturff, A., Bowie, R. C. K., Nathan, R., Yu, S. X., & Getz, W. M. (2019). Insights and approaches using deep learning to classify wildlife. *Scientific Reports*, *9*(1). https://doi.org/10.1038/s41598-019-44565-w

Mihajlovic, I. (2022, March 29). *Everything you ever wanted to know about computer vision.* Medium. https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e

Neate-Clegg, M. H., Horns, J. J., Adler, F. R., Kemahlı Aytekin, M. I., & Şekercioğlu, A. H. (2020). Monitoring the world's bird populations with community science data. *Biological Conservation*, *248*, 108653. https://doi.org/10.1016/j.biocon.2020.108653

Pimm, S. L., Alibhai, S., Bergl, R., Dehgan, A., Giri, C., Jewell, Z., Joppa, L., Kays, R., & Loarie, S. (2015). Emerging technologies to conserve biodiversity. *Trends in Ecology & Evolution*, *30*(11), 685–696. https://doi.org/10.1016/j.tree.2015.08.008

Pitts, W., & McCulloch, W. S. (1947). How we know universals the perception of auditory and visual forms. *The Bulletin of Mathematical Biophysics*, *9*(3), 127–147. https://doi.org/10.1007/bf02478291

Red-cockaded Woodpecker Identification, All about birds, Cornell Lab of Ornithology. (2019). Cornell Labs. https://www.allaboutbirds.org/guide/Red-cockaded_Woodpecker/id#:%7E:text=Measurements&text=Despite%20their%20name%2C%20they%20are,horizontal%20black%2Dand%2Dwhite%20bars

Red-cockaded Woodpecker Survey Protocol. (2018). Environmental Analysis Unit. https://www.fws.gov/sites/default/files/documents/RCW_Survey_protocol.pdf

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr.2016.91

Redmon, J., & Farhadi, A. (2017). YOLO9000: Better, faster, stronger. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr.2017.690

Redmon, J. (2018, April 8). *YOLOv3: An incremental improvement*. arXiv.Org. https://arxiv.org/abs/1804.02767

Redmon, J [@pjreddie]. (2020, February 20). Twitter. https://twitter.com/pjreddie/status/1230524770350817280?ref_src=twsrc%5Etfw%7Ctwcamp%5Etweetembed%7Ctwterm%5E1230524770350817280%7Ctwgr%5E%7Ctwcon%5Es1_&ref_url=https%3A%2F%2Fsyncedreview.com%2F2020%2F02%2F24%2Fyolo-creator-says-he-stopped-cv-research-due-to-ethical-concerns%2F

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, *115*(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

Saleh, A., Sheaves, M., & Rahimi Azghadi, M. (2022). Computer vision and deep learning for fish classification in underwater habitats: A survey. *Fish and Fisheries*, *23*(4), 977–999. https://doi.org/10.1111/faf.12666

Sharma, A. (2022, July 14). *Introduction to the YOLO family*. PyImageSearch. https://pyimagesearch.com/2022/04/04/introduction-to-the-yolo-family/

Simberloff, D. (1993). Species-area and fragmentation effects on old growth forests: prospects for longleaf pine communities. *Proceedings of the Tall Timbers Fire Ecology Conference,* 18. Tall Timbers Research Station, Tallahassee, FL. http://talltimbers.org/wp-content/uploads/2014/03/Simberloff1993_op.pdf

Solawetz, J. (2022, May 11). *What are Anchor Boxes in object detection?* Roboflow Blog. https://blog.roboflow.com/what-is-an-anchor-box/#:%7E:text=In%20order%20to%20predict%20and,prior%2C%20and%20adjust%20from%20there.

Tamou, A. B., Benzinou, A., Nasreddine, K., & Ballihi, L. (2018). Underwater live fish recognition by deep learning. *Lecture Notes in Computer Science*, 275–283. https://doi.org/10.1007/978-3-319-94211-7_30

Tyagi, M. (2022, January 6). *HOG (Histogram of Oriented Gradients): An overview - Towards data science*. Medium. https://towardsdatascience.com/hog-histogram-of-oriented-gradients-67ecd887675f

Ultralytics. (2020). U. (2020). *GitHub - ultralytics/yolov5: YOLOv5 🚀 in PyTorch > ONNX > CoreML > TFLite*. GitHub. https://github.com/ultralytics/yolov5

University of Central Florida. (2016). Computer vision. *Lecture 21-Deformable Part Model (DPM).* https://www.cs.ucf.edu/~bagci/teaching/computervision16/Lec21.pdf

Wang, C., Yeh, I., & Liao, H. (2021, May 10). *You only learn one representation: Unified network for multiple tasks*. arXiv.Org. https://arxiv.org/abs/2105.04206

Ware, S., Frost C., & Doerr, P. D. (1993). Southern mixed hardwood forest: the former longleaf pine forest. *Biodiversity of the southeastern United States: Lowland terrestrial communities.* New York, J. Wiley, p. 447-493.

Weinstein, B. G. (2017). A computer vision for animal ecology. *Journal of Animal Ecology*, *87*(3), 533–545. https://doi.org/10.1111/1365-2656.12780

Wilber, M. J., Scheirer, W. J., Leitner, P., Heflin, B., Zott, J., Reinke, D., Delaney, D. K., & Boult, T. E. (2013). Animal recognition in the Mojave Desert: Vision tools for field biologists. *2013 IEEE Workshop on Applications of Computer Vision (WACV)*. https://doi.org/10.1109/wacv.2013.6475020

U.S. Fish and Wildlife Service. (2003). Recovery plan for the Red-Cockaded Woodpecker (Picoides borealis): Second revision. *U.S. Fish and Wildlife Service.* Atlanta, GA, USA. https://www.nrc.gov/docs/ML1119/ML111920406.pdf

Xiang, L., Deng, K., Wang, G., Zhang, Y., Dang, Q., Gao, Y., Shen, H., Ren, J., Han, S., Ding, E., & Wen, S. (2020, July 23). *PP-YOLO: An effective and efficient implementation of object detector*. arXiv.Org. https://arxiv.org/abs/2007.12099

Xin, H., Wang, X., Lv, W., Bai, X., Long, X., et al., & Yoshie, O. (2021, April 21). *PP-YOLOv2: A practical object detector*. arXiv.Org. https://arxiv.org/abs/2104.10419

Xu, S., Wang, X., Lv, W., Chang, Q., Ciu, C., et al., & Lai, B. (2022*). PP-YOLOE: An evolved version of YOLO*. arXiv.Org. https://arxiv.org/pdf/2203.16250v2.pdf

Zhao, Z. (2018, July 15). *Object detection with deep learning: A review*. arXiv.Org. https://arxiv.org/abs/1807.05511

Zheng, G., Liu, S., Wang, F., Li, Zeming., & Sun, J. (2021, July 18). *YOLOX: Exceeding YOLO Series in 2021*. arXiv.Org. https://arxiv.org/abs/2107.08430

Zou, Z., Shi, Z., Guo, Y., & Ye, J. (2019, May 13). *Object detection in 20 years: A survey*. arXiv.Org. https://arxiv.org/abs/1905.05055

APPENDIX

YOLOv4-tiny.cfg

```
# DarkMark v1.6.21-1 output for Darknet
# Project .... /home/emkun/Downloads/855_FinalIteration_v4tiny/YoloV4
# Config .....
/home/emkun/Downloads/855_FinalIteration_v4tiny/YoloV4/YoloV4.cfg
# Template ... /home/emkun/src/darknet/cfg/yolov4-tiny.cfg
# Username ... emkun@emkun-ROG-Zephyrus-GX550LXS-GX550LXS
# Timestamp .. Wed 2022-06-29 23:18:24 CDT
#
# WARNING:  If you re-generate the darknet files for this project you'll
#  lose any customizations you are about to make in this file!


[net]
# Testing
#batch=1
#subdivisions=1
# Training
batch=1
subdivisions=1
width=416
height=416
```

```
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation=1.500000
exposure=1.500000
hue=0.100000

learning_rate=0.002610
burn_in=1000

max_batches=100000
policy=steps
steps=80000,90000
scales=.1,.1


#weights_reject_freq=1001
#ema_alpha=0.9998
#equidistant_point=1000
#num_sigmas_reject_badlabels=3
#badlabels_rejection_percentage=0.2
cutmix=0
```

```
flip=1

max_chart_loss=4.000000

mixup=0

mosaic=0

use_cuda_graph=0



[convolutional]

batch_normalize=1

filters=32

size=3

stride=2

pad=1

activation=leaky


[convolutional]

batch_normalize=1

filters=64

size=3

stride=2

pad=1

activation=leaky
```

```
[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[route]
layers=-1
groups=2
group_id=1

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
```

```
filters=32
size=3
stride=1
pad=1
activation=leaky

[route]
layers = -1,-2

[convolutional]
batch_normalize=1
filters=64
size=1
stride=1
pad=1
activation=leaky

[route]
layers = -6,-1

[maxpool]
size=2
stride=2
```

```
[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[route]
layers=-1
groups=2
group_id=1

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[convolutional]
```

```
batch_normalize=1

filters=64

size=3

stride=1

pad=1

activation=leaky


[route]

layers = -1,-2


[convolutional]

batch_normalize=1

filters=128

size=1

stride=1

pad=1

activation=leaky


[route]

layers = -6,-1


[maxpool]

size=2
```

```
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[route]
layers=-1
groups=2
group_id=1

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky
```

```
[convolutional]

batch_normalize=1

filters=128

size=3

stride=1

pad=1

activation=leaky


[route]

layers = -1,-2


[convolutional]

batch_normalize=1

filters=256

size=1

stride=1

pad=1

activation=leaky


[route]

layers = -6,-1


[maxpool]
```

```
size=2

stride=2


[convolutional]

batch_normalize=1

filters=512

size=3

stride=1

pad=1

activation=leaky


##################################


[convolutional]

batch_normalize=1

filters=256

size=1

stride=1

pad=1

activation=leaky


[convolutional]

batch_normalize=1
```

```
filters=512

size=3

stride=1

pad=1

activation=leaky


[convolutional]

size=1

stride=1

pad=1

filters=18

activation=linear




[yolo]

mask = 3,4,5

anchors=5, 14, 10, 32, 27, 45, 26, 113, 51, 74, 60, 182

classes=1

num=6

jitter=.3

scale_x_y = 1.05

cls_normalizer=1.0
```

```
iou_normalizer=0.07

iou_loss=ciou

ignore_thresh = .7

truth_thresh = 1

random=0

resize=1.5

nms_kind=greedynms

beta_nms=0.6

#new_coords=1

#scale_x_y = 2.0


[route]

layers = -4


[convolutional]

batch_normalize=1

filters=128

size=1

stride=1

pad=1

activation=leaky


[upsample]
```

```
stride=2

[route]
layers = -1, 23

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear

[yolo]
mask = 0,1,2
anchors=5, 14, 10, 32, 27, 45, 26, 113, 51, 74, 60, 182
```

```
classes=1

num=6

jitter=.3

scale_x_y = 1.05

cls_normalizer=1.0

iou_normalizer=0.07

iou_loss=ciou

ignore_thresh = .7

truth_thresh = 1

random=0

resize=1.5

nms_kind=greedynms

beta_nms=0.6

#new_coords=1

#scale_x_y = 2.0
```

YOLOv4.data

```
classes = 1

train = /home/emkun/Downloads/855_FinalIteration_v4tiny/YoloV4/YoloV4_train.txt

valid = /home/emkun/Downloads/855_FinalIteration_v4tiny/YoloV4/YoloV4_valid.txt

names = /home/emkun/Downloads/855_FinalIteration_v4tiny/YoloV4/YoloV4.names

backup = /home/emkun/Downloads/855_FinalIteration_v4tiny/YoloV4
```

YOLOv4.names

```
Cavity tree
```

YOLOv5n train.py

```
# YOLOv5 🚀 by Ultralytics, GPL-3.0 license
"""
Train a YOLOv5 model on a custom dataset.

Models and datasets download automatically from the latest YOLOv5 release.
Models: https://github.com/ultralytics/yolov5/tree/master/models
Datasets: https://github.com/ultralytics/yolov5/tree/master/data
Tutorial: https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data

Usage:
    $ python path/to/train.py --data coco128.yaml --weights yolov5s.pt --img 640  # from
pretrained (RECOMMENDED)
    $ python path/to/train.py --data coco128.yaml --weights '' --cfg yolov5s.yaml --img
640  # from scratch
"""

import argparse
import math
import os
import random
import sys
import time
from copy import deepcopy
from datetime import datetime
from pathlib import Path

import numpy as np
import torch
import torch.distributed as dist
import torch.nn as nn
import yaml
from torch.cuda import amp
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.optim import SGD, Adam, AdamW, lr_scheduler
from tqdm import tqdm

FILE = Path(__file__).resolve()
ROOT = FILE.parents[0]  # YOLOv5 root directory
if str(ROOT) not in sys.path:
    sys.path.append(str(ROOT))  # add ROOT to PATH
ROOT = Path(os.path.relpath(ROOT, Path.cwd()))  # relative

import val  # for end-of-epoch mAP
from models.experimental import attempt_load
from models.yolo import Model
from utils.autoanchor import check_anchors
from utils.autobatch import check_train_batch_size
from utils.callbacks import Callbacks
from utils.datasets import create_dataloader
from utils.downloads import attempt_download
from utils.general import (LOGGER, check_dataset, check_file, check_git_status,
check_img_size, check_requirements,
                           check_suffix, check_yaml, colorstr, get_latest_run, increment_path,
init_seeds,
                           intersect_dicts, labels_to_class_weights, labels_to_image_weights,
methods, one_cycle,
                           print_args, print_mutation, strip_optimizer)
from utils.loggers import Loggers
from utils.loggers.wandb.wandb_utils import check_wandb_resume
from utils.loss import ComputeLoss
from utils.metrics import fitness
from utils.plots import plot_evolve, plot_labels
from utils.torch_utils import EarlyStopping, ModelEMA, de_parallel, select_device,
torch_distributed_zero_first

LOCAL_RANK = int(os.getenv('LOCAL_RANK', -1))  #
https://pytorch.org/docs/stable/elastic/run.html
```

```
RANK = int(os.getenv('RANK', -1))
WORLD_SIZE = int(os.getenv('WORLD_SIZE', 1))


def train(hyp, opt, device, callbacks):  # hyp is path/to/hyp.yaml or hyp dictionary
    save_dir, epochs, batch_size, weights, single_cls, evolve, data, cfg, resume, noval,
nosave, workers, freeze = \
        Path(opt.save_dir), opt.epochs, opt.batch_size, opt.weights, opt.single_cls,
opt.evolve, opt.data, opt.cfg, \
        opt.resume, opt.noval, opt.nosave, opt.workers, opt.freeze

    # Directories
    w = save_dir / 'weights'  # weights dir
    (w.parent if evolve else w).mkdir(parents=True, exist_ok=True)  # make dir
    last, best = w / 'last.pt', w / 'best.pt'

    # Hyperparameters
    if isinstance(hyp, str):
        with open(hyp, errors='ignore') as f:
            hyp = yaml.safe_load(f)  # load hyps dict
    LOGGER.info(colorstr('hyperparameters: ') + ', '.join(f'{k}={v}' for k, v in
hyp.items()))
```

```
    # Save run settings
    if not evolve:
        with open(save_dir / 'hyp.yaml', 'w') as f:
            yaml.safe_dump(hyp, f, sort_keys=False)
        with open(save_dir / 'opt.yaml', 'w') as f:
            yaml.safe_dump(vars(opt), f, sort_keys=False)

    # Loggers
    data_dict = None
    if RANK in [-1, 0]:
        loggers = Loggers(save_dir, weights, opt, hyp, LOGGER)  # loggers instance
        if loggers.wandb:
            data_dict = loggers.wandb.data_dict
            if resume:
                weights, epochs, hyp, batch_size = opt.weights, opt.epochs, opt.hyp,
opt.batch_size

        # Register actions
        for k in methods(loggers):
            callbacks.register_action(k, callback=getattr(loggers, k))

    # Config
    plots = not evolve  # create plots
    cuda = device.type != 'cpu'
    init_seeds(1 + RANK)
    with torch_distributed_zero_first(LOCAL_RANK):
        data_dict = data_dict or check_dataset(data)  # check if None
    train_path, val_path = data_dict['train'], data_dict['val']
    nc = 1 if single_cls else int(data_dict['nc'])  # number of classes
    names = ['item'] if single_cls and len(data_dict['names']) != 1 else data_dict['names']  #
class names
    assert len(names) == nc, f'{len(names)} names found for nc={nc} dataset in {data}'  #
check
    is_coco = isinstance(val_path, str) and val_path.endswith('coco/val2017.txt')  # COCO
dataset

    # Model
    check_suffix(weights, '.pt')  # check weights
    pretrained = weights.endswith('.pt')
    if pretrained:
        with torch_distributed_zero_first(LOCAL_RANK):
            weights = attempt_download(weights)  # download if not found locally
        ckpt = torch.load(weights, map_location='cpu')  # load checkpoint to CPU to avoid
CUDA memory leak
        model = Model(cfg or ckpt['model'].yaml, ch=3, nc=nc,
anchors=hyp.get('anchors')).to(device)  # create
```

```
        exclude = ['anchor'] if (cfg or hyp.get('anchors')) and not resume else []  # exclude
keys
        csd = ckpt['model'].float().state_dict()  # checkpoint state_dict as FP32
        csd = intersect_dicts(csd, model.state_dict(), exclude=exclude)  # intersect
        model.load_state_dict(csd, strict=False)  # load
        LOGGER.info(f'Transferred {len(csd)}/{len(model.state_dict())} items from
{weights}')  # report
    else:
        model = Model(cfg, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create

    # Freeze
    freeze = [f'model.{x}.' for x in (freeze if len(freeze) > 1 else range(freeze[0]))]  #
layers to freeze
    for k, v in model.named_parameters():
        v.requires_grad = True  # train all layers
        if any(x in k for x in freeze):
            LOGGER.info(f'freezing {k}')
            v.requires_grad = False

    # Image size
    gs = max(int(model.stride.max()), 32)  # grid size (max stride)
    imgsz = check_img_size(opt.imgsz, gs, floor=gs * 2)  # verify imgsz is gs-multiple
    # Batch size
    if RANK == -1 and batch_size == -1:  # single-GPU only, estimate best batch size
        batch_size = check_train_batch_size(model, imgsz)
        loggers.on_params_update({"batch_size": batch_size})

    # Optimizer
    nbs = 64  # nominal batch size
    accumulate = max(round(nbs / batch_size), 1)  # accumulate loss before optimizing
    hyp['weight_decay'] *= batch_size * accumulate / nbs  # scale weight_decay
    LOGGER.info(f"Scaled weight_decay = {hyp['weight_decay']}")

    g0, g1, g2 = [], [], []  # optimizer parameter groups
    for v in model.modules():
        if hasattr(v, 'bias') and isinstance(v.bias, nn.Parameter):  # bias
            g2.append(v.bias)
        if isinstance(v, nn.BatchNorm2d):  # weight (no decay)
            g0.append(v.weight)
        elif hasattr(v, 'weight') and isinstance(v.weight, nn.Parameter):  # weight (with
decay)
            g1.append(v.weight)

    if opt.optimizer == 'Adam':
        optimizer = Adam(g0, lr=hyp['lr0'], betas=(hyp['momentum'], 0.999))  # adjust beta1
to momentum
    elif opt.optimizer == 'AdamW':
        optimizer = AdamW(g0, lr=hyp['lr0'], betas=(hyp['momentum'], 0.999))  # adjust
beta1 to momentum
    else:
        optimizer = SGD(g0, lr=hyp['lr0'], momentum=hyp['momentum'], nesterov=True)

    optimizer.add_param_group({'params': g1, 'weight_decay': hyp['weight_decay']})  #
add g1 with weight_decay
    optimizer.add_param_group({'params': g2})  # add g2 (biases)
    LOGGER.info(f"{colorstr('optimizer:')} {type(optimizer).__name__} with parameter
groups "
                f"{len(g0)} weight (no decay), {len(g1)} weight, {len(g2)} bias")
    del g0, g1, g2

    # Scheduler
    if opt.cos_lr:
        lf = one_cycle(1, hyp['lrf'], epochs)  # cosine 1->hyp['lrf']
    else:
        lf = lambda x: (1 - x / epochs) * (1.0 - hyp['lrf']) + hyp['lrf']  # linear
    scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)  #
plot_lr_scheduler(optimizer, scheduler, epochs)
```

```
    # EMA
    ema = ModelEMA(model) if RANK in [-1, 0] else None

    # Resume
    start_epoch, best_fitness = 0, 0.0
    if pretrained:
        # Optimizer
        if ckpt['optimizer'] is not None:
            optimizer.load_state_dict(ckpt['optimizer'])
            best_fitness = ckpt['best_fitness']

        # EMA
        if ema and ckpt.get('ema'):
            ema.ema.load_state_dict(ckpt['ema'].float().state_dict())
            ema.updates = ckpt['updates']

        # Epochs
        start_epoch = ckpt['epoch'] + 1
        if resume:
            assert start_epoch > 0, f'{weights} training to {epochs} epochs is finished,
nothing to resume.'
        if epochs < start_epoch:
            LOGGER.info(f"{weights} has been trained for {ckpt['epoch']} epochs. Fine-
tuning for {epochs} more epochs.")
            epochs += ckpt['epoch']  # finetune additional epochs

        del ckpt, csd

    # DP mode
    if cuda and RANK == -1 and torch.cuda.device_count() > 1:
        LOGGER.warning('WARNING: DP not recommended, use torch.distributed.run for
best DDP Multi-GPU results.\n'
                       'See Multi-GPU Tutorial at https://github.com/ultralytics/yolov5/issues/475
to get started.')
        model = torch.nn.DataParallel(model)

    # SyncBatchNorm
    if opt.sync_bn and cuda and RANK != -1:
        model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model).to(device)
        LOGGER.info('Using SyncBatchNorm()')

    # Trainloader
    train_loader, dataset = create_dataloader(train_path,
                                              imgsz,
                                              batch_size // WORLD_SIZE,
                                              gs,
                                              single_cls,
                                              hyp=hyp,
                                              augment=True,
                                              cache=None if opt.cache == 'val' else opt.cache,
                                              rect=opt.rect,
                                              rank=LOCAL_RANK,
                                              workers=workers,
                                              image_weights=opt.image_weights,
                                              quad=opt.quad,
                                              prefix=colorstr('train: '),
                                              shuffle=True)
    mlc = int(np.concatenate(dataset.labels, 0)[:, 0].max())  # max label class
    nb = len(train_loader)  # number of batches
    assert mlc < nc, f'Label class {mlc} exceeds nc={nc} in {data}. Possible class labels
are 0-{nc - 1}'

    # Process 0
    if RANK in [-1, 0]:
        val_loader = create_dataloader(val_path,
                                       imgsz,
                                       batch_size // WORLD_SIZE * 2,
                                       gs,
                                       single_cls,
                                       hyp=hyp,
                                       cache=None if noval else opt.cache,
                                       rect=True,
                                       rank=-1,
                                       workers=workers * 2,
                                       pad=0.5,
                                       prefix=colorstr('val: '))[0]

        if not resume:
            labels = np.concatenate(dataset.labels, 0)
            # c = torch.tensor(labels[:, 0])  # classes
            # cf = torch.bincount(c.long(), minlength=nc) + 1.  # frequency
            # model._initialize_biases(cf.to(device))
            if plots:
                plot_labels(labels, names, save_dir)

            # Anchors
            if not opt.noautoanchor:
                check_anchors(dataset, model=model, thr=hyp['anchor_t'], imgsz=imgsz)
            model.half().float()  # pre-reduce anchor precision

        callbacks.run('on_pretrain_routine_end')
```

```
    # DDP mode
    if cuda and RANK != -1:
        model = DDP(model, device_ids=[LOCAL_RANK],
output_device=LOCAL_RANK)

    # Model attributes
    nl = de_parallel(model).model[-1].nl  # number of detection layers (to scale hyps)
    hyp['box'] *= 3 / nl  # scale to layers
    hyp['cls'] *= nc / 80 * 3 / nl  # scale to classes and layers
    hyp['obj'] *= (imgsz / 640) ** 2 * 3 / nl  # scale to image size and layers
    hyp['label_smoothing'] = opt.label_smoothing
    model.nc = nc  # attach number of classes to model
    model.hyp = hyp  # attach hyperparameters to model
    model.class_weights = labels_to_class_weights(dataset.labels, nc).to(device) * nc  #
attach class weights
    model.names = names

    # Start training
    t0 = time.time()
    nw = max(round(hyp['warmup_epochs'] * nb), 100)  # number of warmup iterations,
max(3 epochs, 100 iterations)
    # nw = min(nw, (epochs - start_epoch) / 2 * nb)  # limit warmup to < 1/2 of training
    last_opt_step = -1
    maps = np.zeros(nc)  # mAP per class
    results = (0, 0, 0, 0, 0, 0, 0)  # P, R, mAP@.5, mAP@.5-.95, val_loss(box, obj, cls)
    scheduler.last_epoch = start_epoch - 1  # do not move
    scaler = amp.GradScaler(enabled=cuda)
    stopper = EarlyStopping(patience=opt.patience)
    compute_loss = ComputeLoss(model)  # init loss class
    LOGGER.info(f'Image sizes {imgsz} train, {imgsz} val\n'
                f'Using {train_loader.num_workers * WORLD_SIZE} dataloader workers\n'
                f"Logging results to {colorstr('bold', save_dir)}\n"
                f'Starting training for {epochs} epochs...')
    for epoch in range(start_epoch, epochs):  # epoch -------------------------------------------
----------------------
        model.train()

        # Update image weights (optional, single-GPU only)
        if opt.image_weights:
            cw = model.class_weights.cpu().numpy() * (1 - maps) ** 2 / nc  # class weights
            iw = labels_to_image_weights(dataset.labels, nc=nc, class_weights=cw)  # image
weights
            dataset.indices = random.choices(range(dataset.n), weights=iw, k=dataset.n)  #
rand weighted idx
```

```
        # Update mosaic border (optional)
        # b = int(random.uniform(0.25 * imgsz, 0.75 * imgsz + gs) // gs * gs)
        # dataset.mosaic_border = [b - imgsz, -b]  # height, width borders

        mloss = torch.zeros(3, device=device)  # mean losses
        if RANK != -1:
            train_loader.sampler.set_epoch(epoch)
        pbar = enumerate(train_loader)
        LOGGER.info(('\n' + '%10s' * 7) % ('Epoch', 'gpu_mem', 'box', 'obj', 'cls', 'labels',
'img_size'))
        if RANK in [-1, 0]:
            pbar = tqdm(pbar, total=nb, bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}')  #
progress bar
        optimizer.zero_grad()
        for i, (imgs, targets, paths, _) in pbar:  # batch --------------------------------------------
---------------
            ni = i + nb * epoch  # number integrated batches (since train start)
            imgs = imgs.to(device, non_blocking=True).float() / 255  # uint8 to float32, 0-255
to 0.0-1.0

            # Warmup
            if ni <= nw:
                xi = [0, nw]  # x interp
                # compute_loss.gr = np.interp(ni, xi, [0.0, 1.0])  # iou loss ratio (obj_loss = 1.0
or iou)
                accumulate = max(1, np.interp(ni, xi, [1, nbs / batch_size]).round())
                for j, x in enumerate(optimizer.param_groups):
                    # bias lr falls from 0.1 to lr0, all other lrs rise from 0.0 to lr0
                    x['lr'] = np.interp(ni, xi, [hyp['warmup_bias_lr'] if j == 2 else 0.0,
x['initial_lr'] * lf(epoch)])
                    if 'momentum' in x:
                        x['momentum'] = np.interp(ni, xi, [hyp['warmup_momentum'],
hyp['momentum']])

            # Multi-scale
            if opt.multi_scale:
                sz = random.randrange(imgsz * 0.5, imgsz * 1.5 + gs) // gs * gs  # size
                sf = sz / max(imgs.shape[2:])  # scale factor
                if sf != 1:
                    ns = [math.ceil(x * sf / gs) * gs for x in imgs.shape[2:]]  # new shape
(stretched to gs-multiple)
                    imgs = nn.functional.interpolate(imgs, size=ns, mode='bilinear',
align_corners=False)

            # Forward
            with amp.autocast(enabled=cuda):
```

```
            pred = model(imgs)  # forward
            loss, loss_items = compute_loss(pred, targets.to(device))  # loss scaled by
batch_size
            if RANK != -1:
                loss *= WORLD_SIZE  # gradient averaged between devices in DDP mode
            if opt.quad:
                loss *= 4.

        # Backward
        scaler.scale(loss).backward()

        # Optimize
        if ni - last_opt_step >= accumulate:
            scaler.step(optimizer)  # optimizer.step
            scaler.update()
            optimizer.zero_grad()
            if ema:
                ema.update(model)
            last_opt_step = ni

        # Log
        if RANK in [-1, 0]:
            mloss = (mloss * i + loss_items) / (i + 1)  # update mean losses
```

```
                mem = f'{torch.cuda.memory_reserved() / 1E9 if torch.cuda.is_available() else
0:.3g}G'  # (GB)
                pbar.set_description(('%10s' * 2 + '%10.4g' * 5) %
                                     (f'{epoch}/{epochs - 1}', mem, *mloss, targets.shape[0],
imgs.shape[-1]))
                callbacks.run('on_train_batch_end', ni, model, imgs, targets, paths, plots,
opt.sync_bn)
                if callbacks.stop_training:
                    return
            # end batch ------------------------------------------------------------------------------------
------------

        # Scheduler
        lr = [x['lr'] for x in optimizer.param_groups]  # for loggers
        scheduler.step()

        if RANK in [-1, 0]:
            # mAP
            callbacks.run('on_train_epoch_end', epoch=epoch)
            ema.update_attr(model, include=['yaml', 'nc', 'hyp', 'names', 'stride',
'class_weights'])
            final_epoch = (epoch + 1 == epochs) or stopper.possible_stop
            if not noval or final_epoch:  # Calculate mAP
            results, maps, _ = val.run(data_dict,
                                       batch_size=batch_size // WORLD_SIZE * 2,
                                       imgsz=imgsz,
                                       model=ema.ema,
                                       single_cls=single_cls,
                                       dataloader=val_loader,
                                       save_dir=save_dir,
                                       plots=False,
                                       callbacks=callbacks,
                                       compute_loss=compute_loss)

        # Update best mAP
        fi = fitness(np.array(results).reshape(1, -1))  # weighted combination of [P, R,
mAP@.5, mAP@.5-.95]
        if fi > best_fitness:
            best_fitness = fi
        log_vals = list(mloss) + list(results) + lr
        callbacks.run('on_fit_epoch_end', log_vals, epoch, best_fitness, fi)

        # Save model
        if (not nosave) or (final_epoch and not evolve):  # if save
            ckpt = {
                'epoch': epoch,
                'best_fitness': best_fitness,
                'model': deepcopy(de_parallel(model)).half(),
                'ema': deepcopy(ema.ema).half(),
                'updates': ema.updates,
                'optimizer': optimizer.state_dict(),
                'wandb_id': loggers.wandb.wandb_run.id if loggers.wandb else None,
                'date': datetime.now().isoformat()}

            # Save last, best and delete
            torch.save(ckpt, last)
            if best_fitness == fi:
                torch.save(ckpt, best)
            if (epoch > 0) and (opt.save_period > 0) and (epoch % opt.save_period == 0):
                torch.save(ckpt, w / f'epoch{epoch}.pt')
            del ckpt
            callbacks.run('on_model_save', last, epoch, final_epoch, best_fitness, fi)

        # Stop Single-GPU
        if RANK == -1 and stopper(epoch=epoch, fitness=fi):
            break

        # Stop DDP TODO: known issues
shttps://github.com/ultralytics/yolov5/pull/4576
```

```
            # stop = stopper(epoch=epoch, fitness=fi)
            # if RANK == 0:
            #    dist.broadcast_object_list([stop], 0)  # broadcast 'stop' to all ranks

        # Stop DPP
        # with torch_distributed_zero_first(RANK):
        # if stop:
        #    break  # must break all DDP ranks

        # end epoch ----------------------------------------------------------------------------------
--------------
    # end training -----------------------------------------------------------------------------------
--------------
    if RANK in [-1, 0]:
        LOGGER.info(f'\n{epoch - start_epoch + 1} epochs completed in {(time.time() - t0)
/ 3600:.3f} hours.')
        for f in last, best:
            if f.exists():
                strip_optimizer(f)  # strip optimizers
                if f is best:
                    LOGGER.info(f'\nValidating {f}...')
                    results, _, _ = val.run(
                        data_dict,
                        batch_size=batch_size // WORLD_SIZE * 2,
                        imgsz=imgsz,
                        model=attempt_load(f, device).half(),
                        iou_thres=0.65 if is_coco else 0.60,  # best pycocotools results at 0.65
                        single_cls=single_cls,
                        dataloader=val_loader,
                        save_dir=save_dir,
                        save_json=is_coco,
                        verbose=True,
                        plots=True,
                        callbacks=callbacks,
                        compute_loss=compute_loss)  # val best model with plots
                    if is_coco:
                        callbacks.run('on_fit_epoch_end', list(mloss) + list(results) + lr, epoch,
best_fitness, fi)

        callbacks.run('on_train_end', last, best, plots, epoch, results)
        LOGGER.info(f"Results saved to {colorstr('bold', save_dir)}")

    torch.cuda.empty_cache()
    return results
```

```
def parse_opt(known=False):
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', type=str, default=ROOT / 'yolov5s.pt', help='initial
weights path')
    parser.add_argument('--cfg', type=str, default='', help='model.yaml path')
    parser.add_argument('--data', type=str, default=ROOT / 'data/coco128.yaml',
help='dataset.yaml path')
    parser.add_argument('--hyp', type=str, default=ROOT / 'data/hyps/hyp.scratch-
low.yaml', help='hyperparameters path')
    parser.add_argument('--epochs', type=int, default=300)
    parser.add_argument('--batch-size', type=int, default=16, help='total batch size for all
GPUs, -1 for autobatch')
    parser.add_argument('--imgsz', '--img', '--img-size', type=int, default=640, help='train,
val image size (pixels)')
    parser.add_argument('--rect', action='store_true', help='rectangular training')
    parser.add_argument('--resume', nargs='?', const=True, default=False, help='resume
most recent training')
    parser.add_argument('--nosave', action='store_true', help='only save final checkpoint')
    parser.add_argument('--noval', action='store_true', help='only validate final epoch')
    parser.add_argument('--noautoanchor', action='store_true', help='disable AutoAnchor')
    parser.add_argument('--evolve', type=int, nargs='?', const=300, help='evolve
hyperparameters for x generations')
    parser.add_argument('--bucket', type=str, default='', help='gsutil bucket')
    parser.add_argument('--cache', type=str, nargs='?', const='ram', help='--cache images in
"ram" (default) or "disk"')
    parser.add_argument('--image-weights', action='store_true', help='use weighted image
selection for training')
    parser.add_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')
    parser.add_argument('--multi-scale', action='store_true', help='vary img-size +/-
50%%')
    parser.add_argument('--single-cls', action='store_true', help='train multi-class data as
single-class')
    parser.add_argument('--optimizer', type=str, choices=['SGD', 'Adam', 'AdamW'],
default='SGD', help='optimizer')
    parser.add_argument('--sync-bn', action='store_true', help='use SyncBatchNorm, only
available in DDP mode')
    parser.add_argument('--workers', type=int, default=8, help='max dataloader workers
(per RANK in DDP mode)')
    parser.add_argument('--project', default=ROOT / 'runs/train', help='save to
project/name')
    parser.add_argument('--name', default='exp', help='save to project/name')
    parser.add_argument('--exist-ok', action='store_true', help='existing project/name ok,
do not increment')
    parser.add_argument('--quad', action='store_true', help='quad dataloader')
    parser.add_argument('--cos-lr', action='store_true', help='cosine LR scheduler')
    parser.add_argument('--label-smoothing', type=float, default=0.0, help='Label
smoothing epsilon')
    parser.add_argument('--patience', type=int, default=100, help='EarlyStopping patience
(epochs without improvement)')
    parser.add_argument('--freeze', nargs='+', type=int, default=[0], help='Freeze layers:
backbone=10, first3=0 1 2')
    parser.add_argument('--save-period', type=int, default=-1, help='Save checkpoint every
x epochs (disabled if < 1)')
    parser.add_argument('--local_rank', type=int, default=-1, help='DDP parameter, do not
modify')

    # Weights & Biases arguments
    parser.add_argument('--entity', default=None, help='W&B: Entity')
    parser.add_argument('--upload_dataset', nargs='?', const=True, default=False,
help='W&B: Upload data, "val" option')
    parser.add_argument('--bbox_interval', type=int, default=-1, help='W&B: Set
bounding-box image logging interval')
    parser.add_argument('--artifact_alias', type=str, default='latest', help='W&B: Version
of dataset artifact to use')

    opt = parser.parse_known_args()[0] if known else parser.parse_args()
    return opt
```

```
def main(opt, callbacks=Callbacks()):
    # Checks
    if RANK in [-1, 0]:
        print_args(vars(opt))
        check_git_status()
        check_requirements(exclude=['thop'])

    # Resume
    if opt.resume and not check_wandb_resume(opt) and not opt.evolve:  # resume an
interrupted run
        ckpt = opt.resume if isinstance(opt.resume, str) else get_latest_run()  # specified or
most recent path
        assert os.path.isfile(ckpt), 'ERROR: --resume checkpoint does not exist'
        with open(Path(ckpt).parent.parent / 'opt.yaml', errors='ignore') as f:
            opt = argparse.Namespace(**yaml.safe_load(f))  # replace
        opt.cfg, opt.weights, opt.resume = '', ckpt, True  # reinstate
        LOGGER.info(f'Resuming training from {ckpt}')
    else:
        opt.data, opt.cfg, opt.hyp, opt.weights, opt.project = \
            check_file(opt.data), check_yaml(opt.cfg), check_yaml(opt.hyp), str(opt.weights),
str(opt.project)  # checks
        assert len(opt.cfg) or len(opt.weights), 'either --cfg or --weights must be specified'
        if opt.evolve:
            if opt.project == str(ROOT / 'runs/train'):  # if default project name, rename to
runs/evolve
                opt.project = str(ROOT / 'runs/evolve')
            opt.exist_ok, opt.resume = opt.resume, False  # pass resume to exist_ok and
disable resume
        if opt.name == 'cfg':
            opt.name = Path(opt.cfg).stem  # use model.yaml as name
        opt.save_dir = str(increment_path(Path(opt.project) / opt.name,
exist_ok=opt.exist_ok))

    # DDP mode
    device = select_device(opt.device, batch_size=opt.batch_size)
    if LOCAL_RANK != -1:
        msg = 'is not compatible with YOLOv5 Multi-GPU DDP training'
        assert not opt.image_weights, f'--image-weights {msg}'
        assert not opt.evolve, f'--evolve {msg}'
        assert opt.batch_size != -1, f'AutoBatch with --batch-size -1 {msg}, please pass a
valid --batch-size'
        assert opt.batch_size % WORLD_SIZE == 0, f'--batch-size {opt.batch_size} must
be multiple of WORLD_SIZE'
        assert torch.cuda.device_count() > LOCAL_RANK, 'insufficient CUDA devices for
DDP command'
```

```
    torch.cuda.set_device(LOCAL_RANK)

    device = torch.device('cuda', LOCAL_RANK)

    dist.init_process_group(backend="nccl" if dist.is_nccl_available() else "gloo")


# Train

if not opt.evolve:

    train(opt.hyp, opt, device, callbacks)

    if WORLD_SIZE > 1 and RANK == 0:

        LOGGER.info('Destroying process group... ')

        dist.destroy_process_group()


# Evolve hyperparameters (optional)

else:

    # Hyperparameter evolution metadata (mutation scale 0-1, lower_limit, upper_limit)

    meta = {

        'lr0': (1, 1e-5, 1e-1),  # initial learning rate (SGD=1E-2, Adam=1E-3)

        'lrf': (1, 0.01, 1.0),  # final OneCycleLR learning rate (lr0 * lrf)

        'momentum': (0.3, 0.6, 0.98),  # SGD momentum/Adam beta1

        'weight_decay': (1, 0.0, 0.001),  # optimizer weight decay

        'warmup_epochs': (1, 0.0, 5.0),  # warmup epochs (fractions ok)

        'warmup_momentum': (1, 0.0, 0.95),  # warmup initial momentum

        'warmup_bias_lr': (1, 0.0, 0.2),  # warmup initial bias lr

        'box': (1, 0.02, 0.2),  # box loss gain
        'cls': (1, 0.2, 4.0),  # cls loss gain

        'cls_pw': (1, 0.5, 2.0),  # cls BCELoss positive_weight

        'obj': (1, 0.2, 4.0),  # obj loss gain (scale with pixels)

        'obj_pw': (1, 0.5, 2.0),  # obj BCELoss positive_weight

        'iou_t': (0, 0.1, 0.7),  # IoU training threshold

        'anchor_t': (1, 2.0, 8.0),  # anchor-multiple threshold

        'anchors': (2, 2.0, 10.0),  # anchors per output grid (0 to ignore)

        'fl_gamma': (0, 0.0, 2.0),  # focal loss gamma (efficientDet default gamma=1.5)

        'hsv_h': (1, 0.0, 0.1),  # image HSV-Hue augmentation (fraction)

        'hsv_s': (1, 0.0, 0.9),  # image HSV-Saturation augmentation (fraction)

        'hsv_v': (1, 0.0, 0.9),  # image HSV-Value augmentation (fraction)

        'degrees': (1, 0.0, 45.0),  # image rotation (+/- deg)

        'translate': (1, 0.0, 0.9),  # image translation (+/- fraction)

        'scale': (1, 0.0, 0.9),  # image scale (+/- gain)

        'shear': (1, 0.0, 10.0),  # image shear (+/- deg)

        'perspective': (0, 0.0, 0.001),  # image perspective (+/- fraction), range 0-0.001

        'flipud': (1, 0.0, 1.0),  # image flip up-down (probability)

        'fliplr': (0, 0.0, 1.0),  # image flip left-right (probability)

        'mosaic': (1, 0.0, 1.0),  # image mixup (probability)

        'mixup': (1, 0.0, 1.0),  # image mixup (probability)

        'copy_paste': (1, 0.0, 1.0)}  # segment copy-paste (probability)


    with open(opt.hyp, errors='ignore') as f:
        hyp = yaml.safe_load(f)  # load hyps dict
        if 'anchors' not in hyp:  # anchors commented in hyp.yaml
            hyp['anchors'] = 3
    opt.noval, opt.nosave, save_dir = True, True, Path(opt.save_dir)  # only val/save
final epoch
    # ei = [isinstance(x, (int, float)) for x in hyp.values()]  # evolvable indices
    evolve_yaml, evolve_csv = save_dir / 'hyp_evolve.yaml', save_dir / 'evolve.csv'
    if opt.bucket:
        os.system(f'gsutil cp gs://{opt.bucket}/evolve.csv {evolve_csv}')  # download
evolve.csv if exists

    for _ in range(opt.evolve):  # generations to evolve
        if evolve_csv.exists():  # if evolve.csv exists: select best hyps and mutate
            # Select parent(s)
            parent = 'single'  # parent selection method: 'single' or 'weighted'
            x = np.loadtxt(evolve_csv, ndmin=2, delimiter=',', skiprows=1)
            n = min(5, len(x))  # number of previous results to consider
            x = x[np.argsort(-fitness(x))][:n]  # top n mutations
            w = fitness(x) - fitness(x).min() + 1E-6  # weights (sum > 0)
            if parent == 'single' or len(x) == 1:
                # x = x[random.randint(0, n - 1)]  # random selection
                x = x[random.choices(range(n), weights=w)[0]]  # weighted selection
            elif parent == 'weighted':
```

```
            x = (x * w.reshape(n, 1)).sum(0) / w.sum()  # weighted combination

        # Mutate
        mp, s = 0.8, 0.2  # mutation probability, sigma
        npr = np.random
        npr.seed(int(time.time()))
        g = np.array([meta[k][0] for k in hyp.keys()])  # gains 0-1
        ng = len(meta)
        v = np.ones(ng)
        while all(v == 1):  # mutate until a change occurs (prevent duplicates)
            v = (g * (npr.random(ng) < mp) * npr.randn(ng) * npr.random() * s +
1).clip(0.3, 3.0)
        for i, k in enumerate(hyp.keys()):  # plt.hist(v.ravel(), 300)
            hyp[k] = float(x[i + 7] * v[i])  # mutate

    # Constrain to limits
    for k, v in meta.items():
        hyp[k] = max(hyp[k], v[1])  # lower limit
        hyp[k] = min(hyp[k], v[2])  # upper limit
        hyp[k] = round(hyp[k], 5)  # significant digits

    # Train mutation
    results = train(hyp.copy(), opt, device, callbacks)
```

```
            callbacks = Callbacks()
            # Write mutation results
            print_mutation(results, hyp.copy(), save_dir, opt.bucket)


        # Plot results
        plot_evolve(evolve_csv)
        LOGGER.info(f'Hyperparameter evolution finished {opt.evolve} generations\n'
                    f"Results saved to {colorstr('bold', save_dir)}\n"
                    f'Usage example: $ python train.py --hyp {evolve_yaml}')



def run(**kwargs):
    # Usage: import train; train.run(data='coco128.yaml', imgsz=320,
weights='yolov5m.pt')
    opt = parse_opt(True)
    for k, v in kwargs.items():
        setattr(opt, k, v)
    main(opt)
    return opt


if __name__ == "__main__":
    opt = parse_opt()
    main(opt)
```

YoloV5n_data.yaml

```
train: ../train/images

val: ../valid/images

test: ../test/images

nc: 1

names: ['cavity']
```

VITA

Emerson de Lemmus (He/Him/His)

EXPERIENCE

Machine Learning Engineer – Verity Integrated Systems/Lockheed Martin 2022 - Present

- Neural network creation and surrogate modeling optimization.

Graduate Assistant – Raven Environmental Services/U.S. Fish and Wildlife 2021 - 2022
Thesis: Detecting Endangered Red-cockaded Woodpecker Habitats using Computer Vision

- Enjoyed team collaboration with representatives from various government and local agencies to identify endangered Red-Coackaded Woodpecker habitats using drone-mounted computer vision.
- My contribution increased workflow performance and saved hundreds of yearly work hours, classifying habitats manually.
- Implemented real-time object recognition models.

Instructor – Department of Computer Science, Sam Houston State University 2020-2022

- Instructor for graduate-level Computer Science Core Topics. I successfully introduced CS concepts to students of non-technical backgrounds.
- Instructor for undergraduate Programming Fundamentals. 87% approval rating from students.
- GA for the NSF-funded REU Genome Science and Computational Biology. Worked with a team of faculty and students where my role was to represent the Department of Computer Science, coordinate and plan meetings, and contribute independent research for publication.

SKILLS

Software: Java, C++, Python, SQL, Darknet (Proficient). CUDA, TensorFlow (Conversant).
Interests: CS Education, Wildlife Preservation, Machine Learning, Ethical AI, Hebrew/Ladino.
Languages: English (fluent), Spanish (fluent).

EDUCATION

Sam Houston State University
Master of Science in Computing and Data Science 2022
Bachelor of Science in Computing and Information Science 2019

AWARDS & SERVICE

Awards: U.S. Department of Defense Cyber Scholarship Program Recipient; I received a full-ride scholarship for
a Ph.D. in Cyber and Digital Forensics (2022).
Enhancing Undergraduate Research Experiences and Creative Activities Recipient Winner for studying
vehicular ad hoc networks. (2018)

Service: Middle/ High School outreach for SHSU CS Department. (2020 - 2022)
Association for Computing Machinery SHSU Student Chapter, Leadership (2018-2019)