



BCT Bachelor of Science (Computer Science & Information Technology)

Livestock Management with OpenCV and a UAV (drone)

Final Year Project Report

Student:

Lorcan Creedon

16422524

l.creedon3@nuigalway.ie

Project Supervisor

Enda Barrett

enda.barrett@nuigalway.ie

Table of Contents

Table of Contents	2
Table of Figures	4
Table of Tables.....	4
Abbreviations	4
1. Introduction.....	5
1.1. Project Abstract	5
1.2. Agricultural Drones	5
1.3. Object Detection	6
2. Technical Review:.....	7
2.1. Tools & Technologies Used	7
2.1.1. Labellmg & Image Downloader	7
2.1.2. Google Open Images Dataset & OIdv4 Tool Kit	7
2.1.3. YOLO	8
2.1.4. Google Colab GPU	9
2.1.5. Python.....	9
2.1.6. OpenCV & Other Python Libraries	9
2.1.7. NginX RTMP & DJI GO 4	10
2.2. Relevance of BCT Course Content to Project.....	10
2.2.1. Machine Learning	11
2.2.2. Graphics & Image Processing.....	11
2.2.3. Programming	11
2.2.4. Software Engineering	11
3. Technical Research	12
4. Design	13
4.1. Drone Communication System	14
4.2. YOLO Model Training System	15
4.3. Live Processing System.....	16
5. Development & Implementation.....	17
5.1. Building a custom Dataset	17
5.1.1. Labellmg & Image Downloader	17
5.1.2. Google Open Images Database & OIdv4 Tool Kit	18
5.2. Cloning and Compiling Darknet on the Cloud	19
5.3. Preparing Data for Training.....	19
5.4. Customising Detection Model Configuration	21
5.4.1. YOLOv3.cfg.....	21

5.4.2.	YOLOv3-tiny.cfg	22
5.4.3.	YOLOv4-tiny.cfg	22
5.5.	Training Detection Models.....	23
5.6.	Handling Detections with OpenCV	24
5.6.1.	Getting Live Drone Feed.....	24
5.6.2.	Loading Custom Detection Model	25
5.6.3	Finding Detections.....	25
5.6.4.	Drawing Bounding Boxes and Counting Objects	27
5.6.5.	Applying Detection Model to Drone Feed.....	27
5.6.7.	Getting FPS of Video Capture.....	28
5.6.8.	Saving Output Video Capture	29
6.	Testing & Results	29
6.1.	Testing.....	29
6.2.	YOLOv3 Results.....	30
6.3.	YOLOv3-tiny Results	33
6.4.	YOLOv4-tiny Results	36
6.5.	Model Comparison	39
7.	Conclusions & Future Work	40
8.	Bibliography.....	41
9.	Appendix.....	42
9.1.	Object Detection Code	42
9.2.	Split Dataset Code	45
9.3.	YOLO Model Training Notebook	46

Table of Figures

Figure 1: Drone Communication System	14
Figure 2: Model Training System	15
Figure 3: Live Processing System	16
Figure 4: OIdv4 Building Cow/Sheep Dataset	18
Figure 5: YOLOv3.cfg File.....	22
Figure 6: Images Used for Testing.....	30
Figure 7: YOLOv3 Loss & mAP Chart	31
Figure 8: Results of YOLOv3 on Test Images	32
Figure 9: Results of YOLOv3 in Real-Time.....	33
Figure 10: YOLOv3-tiny Loss & mAP Chart	34
Figure 11: Result of YOLOv3-tiny on Test Images	34
Figure 12: Result of YOLOv3-tiny in Real-Time	35
Figure 13: YOLOv3-tiny (large weights) Loss & mAP Chart	35
Figure 14: YOLOv4-tiny (cattle/sheep) Loss & mAP Chart	36
Figure 15: Result of YOLOv4-tiny on Test Images	37
Figure 16: Result of YOLOv4-tiny in Real-Time	37
Figure 17: YOLOv4-tiny (cattle) Loss & mAP Chart.....	38
Figure 18: Result of YOLOv4-tiny Cattle on Test Image	38
Figure 19: Result of YOLOv4-tiny Cattle in Real-Time	39

Table of Tables

Table 1: Object Detection Model Comparison	12
Table 2: Output NumPy Array from the Network Detection Layers	25
Table 3: YOLO Results	39

Abbreviations

UAV	Unmanned Aerial Vehicle
YOLO	You Only Look Once
FAA	Federal Aviation Administration
CNN	Convolutional Neural Network
RTMP	Real-Time Messaging Protocol
NMS	Non-Maximum Suppression
mAP	mean Average Precision
fps	frames per second
IDE	Integrated Development Environment

1. Introduction

In this chapter, I introduce the project, briefly discuss the main concepts behind it and put the work in its business context.

1.1. Project Abstract

My final year project is titled *“Livestock Management with OpenCV and a UAV (Drone)”*. The goal of this project was to analyse the first-person view from a UAV and process it in real time using OpenCV to create functionality such as detecting and counting objects in motion.

I decided to take this project goal and base it around the revolutionary use of drones in agriculture. During this project, I focused on the recognition and detection of livestock animals such as cattle and sheep. Combining techniques of machine learning, image processing and python programming, I was able to train my own object detection models to recognise livestock. With the aid of the OpenCV library I was able to apply these object detection models to the eyes of a drone while highlighting and counting each of the livestock animals in frame.

With the autonomous flight feature of a drone and an accurate object detection model, a project such as this could prove useful in livestock farming as it would simplify the process of, e.g., keeping count on various livestock animals and general monitoring of livestock.

1.2. Agricultural Drones

The agricultural world has welcomed the use of drone technology with open arms to make significant advancements in modern farming. An agricultural drone is one that can perform many useful jobs in a farmer’s daily life, even something as simple as flying a drone through fields to scout livestock would help by minimising footwork for farmers. Sensors and digital imaging capabilities can give farmers a far more detailed view of their fields and can help to increase efficiency with various farming processes such as crop monitoring, crop spraying, pest/fungal infestation monitoring, irrigation mapping and livestock management.

As the use of drones began to boom in the world of agriculture, in 2016, following incidents such as drones colliding with crop-dusters, the Federal Aviation Administration (FAA) brought in new regulations requiring commercial drone operators, such as farmers, to pass a knowledge exam, obtain a license and abide by restrictions that would allow drones to be used in a safer manner in the workspace.

Today, the most prominent use for drones in agriculture is crop management. With permission from the FAA, drones can be equipped with tanks to hold fertilizer and a nozzle to spray the crops.

Traditionally, this would be done manually or with an airplane, so the use of a drone in this farming process greatly improves crop spraying making it more efficient and less costly. The DJI Agras drone series was developed for the specific function of crop spraying. The newest model, DJI Agras T20, can hold up to 20 litres of fertilizer, spraying in a 7-meter radius and has a 3D flight planning function built into the controller. In a similar field, drone planting is a relatively newer technology that is not as widely used, but companies are experimenting with drone planting for rapid reforestation after disastrous forest fires. DroneSeed is a company which provides a rapid reforestation service, with the use of drone technology for landowners with 50+ acres of land. Their drones can spread up to 40 acres of seeds a day, making reforestation more scalable and mitigating climate change.

In a field more relevant to my project, drone technology for livestock management has not yet taken off quite as much as crop management. However, this is certain to change in the coming years as drone technology in agriculture continues to gain momentum. In fact, in recent years, researchers have been developing ways of incorporating drones into livestock production. In 2017, a study carried out by students of Chosun University lead to successfully herding animals into their pen by installing speakers on to multiple quadrotor UAVs.

In relation to counting cattle, a project such as mine would set the foundation for livestock management drone with counting functionality. There has not been much evidence of an efficient system for this particular application. However, one approach devised by Jayme Barbedo at Brazilian Agricultural Research in 2020 (Ref: [5]), combined techniques such as: deep learning to predict the rough location of an animal as it moves out of frame, colour and background manipulation to make the animals more prominent, mathematical morphology to isolate the animals whilst in clustered groups and image matching to take into account overlapping. This approach showed accurate results of up to 90% when counting cattle in a herd. Therefore, there is evidence that the world of agriculture may see drones developed specifically for livestock management in the near future.

1.3. Object Detection

Object detection is a computer vision technique which focuses on identifying and locating objects of interest of a certain class, such as animals or cars, in a video or image capture. Every object of a certain class has unique features that help to classify them. Using bounding boxes and class labels, an object detection model should be able to highlight and classify multiple objects in a given frame.

The most notable feature of an object detection model is its adaptation of deep learning with Convolutional Neural Networks (CNNs). CNNs are neural networks primarily used to classify images and to perform object recognition. When training an object detection model, CNNs process images as objects and tensors made up of multidimensional matrices containing numbers, rather than flat images measure by width and height only. As images are inputted to a CNN, they are perceived as a rectangular box measured not only by width and height, but also by depth. The depth of an image in a CNN is separated into three layers, also known as channels, one channel for each letter of RGB encoding. The architecture of a CNN is generally made up of a number of convolutional layers, pooling layers and fully-connected layers.

Convolutional layers have a specified number of filters defined by a width and height, these filters are essentially what detect patterns in images. For example, filters in the first few convolutional layers of a CNN may detect geometrical shapes patterns in an image. The deeper the CNN goes and as images

pass through more and more convolutional layers, the more sophisticated these filters become and will eventually be able to detect full objects. Convolutional layers also hold the number of input and output channels in the network as they convolve the input and pass the output to the next layer. After passing through a convolutional layer, an image is extracted to a feature map (aka. Activation map) with a structure: (number of inputs) * (activation map height) * (activation map width) * (activation map channels). Pooling layers in a CNN perform a downsampling operation whereby the spatial size of the input is progressively reduced which helps to avoid overfitting and minimize computational work. Fully-Connected layers then connect all inputs in one layer to the outputs of the previous convolutional layer.

There are many models/algorithms that adapt a variation of CNN learning for object detection such as Regional-CNN, Single Shot MultiBox Detector and Retina-Net. However, the model that I have implemented and will be discussing in detail in this project is YOLO (You Only Look Once), a family of real-time object detection algorithms.

2. Technical Review:

In this chapter, I review each of the tools and technologies used in my project. I also discuss what techniques and methodologies gained from my time studying the BCT course content influenced the development of my project.

2.1. Tools & Technologies Used

2.1.1. Labellmg & Image Downloader

As necessary for training a custom object detection model, I needed to create custom datasets containing a large number of labelled images to carry out this project. Image Downloader is a browser extension that can be downloaded and used as a plugin to Chrome. It allows users to download images found on a web page in bulk, for example a Google Images page containing pictures of sheep, while also filtering the images by a user specified image size (width & height). Labellmg is a graphical image annotation tool that allows users to define classes (e.g. dog / cat) and label them by manually drawing a bounding box over each class instance in the image and selecting a label for them. Users can then select the output type of Labellmg to suit the object detection model at hand. For example, selecting the YOLO formatting output type generates a text file for each image, containing the coordinates of the bounding boxes drawn around each class instance in the image. However, I personally found this method very tedious and time consuming as you are required to manually draw bounding boxes and label classes in every image. As large image datasets are needed to build accurate detection models, I found this method of dataset building very inefficient.

2.1.2. Google Open Images Dataset & OIv4 Tool Kit

Open Images is an image dataset put together by Google of around 1.9 million images annotated with class labels and object bounding boxes. It contains 16 million bounding boxes for 600 object classes to choose from. This dataset, combined with a tool named OIDv4 Tool Kit, allows users to create datasets fit for training an object detection model. OIDv4 Tool Kit can be forked from its github repository and allows users to take advantage of the Open Images dataset with minimal effort required. With this tool kit, users can download a specified number of images per class and save them to the same or different folder per class, while also saving the bounding box coordinates and labels for each image in different folders. In comparison to the combination of Labellmg and Image Downloader, Open Images and the OIDv4 Tool Kit is a much more efficient way of creating large datasets for a custom object detection model as the workload is minimised greatly and the bounding boxes are accurately drawn over objects. However, it is possible that the object you wish to create a detection model around is not included in the 600 classes that Open Images covers, in which case Labellmg would be the more flexible option here.

2.1.3. YOLO

YOLO is a collection of deep learning models designed for quick and efficient object detection in real-time. During this project, I experimented with multiple versions of YOLO such as YOLOv3, YOLOv3-tiny and YOLOv4-tiny.

YOLOv3 is a model that was developed on its predecessors, YOLOv1 and YOLOv2. It uses Darknet-53 as its backbone architecture which is used for feature extraction. Darknet-53 is a deep CNN framework with 53 convolutional layers. YOLOv3 has refined the design of previous models by introducing new concepts such as multi-scale prediction, as well as bounding box prediction using a logistic regression technique. The model uses downsampling to reduce the input images into 3 different scales to support multi-scale prediction which improves the detection performance on smaller or further away objects. Independent logistic regression classifiers are then used to give a class score prediction and a threshold is used to provide multi-label classification for objects detected in images. A Non-Maximum Suppression (NMS) threshold is included in the network to filter out duplicated or internal bounding boxes around a detected object. This YOLOv3 network essentially divides input images into regions and predicts bounding boxes and probabilities for each of these regions.

YOLOv4 is proven to be twice as fast as its predecessor, though greater GPU power is required to train and run this YOLO version. This model introduces a more optimised backbone architecture and new data augmentation techniques.

YOLOv3-tiny and YOLOv4-tiny are essentially compressed version of the regular YOLOv3 and YOLOv4 models. The main difference is that the number of convolutional layers is reduced and instead of making predictions on three different scales they instead use two different scales. There is a trade-off here between accuracy and fps. These 'tiny' models are much quicker to train as there are far less layers for the input to travel through and they generally give better fps results when running the detection model on real-time footage. However, these 'tiny' models do not give as accurate scores for the confidence of detections as the regular models do.

Training a custom YOLO model can be done via transfer learning whereby the configuration for a YOLO model can be adjusted to suit a custom dataset and with pre-trained weights, a custom model can be

trained on those weights rather than randomly initializing weights. This saves a lot of computational time when training a custom detection model. Darknet provides pretrained weights and a sample configuration file for all versions of YOLO.

2.1.4. Google Colab GPU

Due to the fact that my personal computer had limited GPU specifications and this project was quite computer-vision based, it was necessary for me to seek extra GPU resources for training object detection models as training them on CPU would take much longer. Google offer free GPU via Colab notebook whereby the runtime type settings can be changed to enable GPU as the hardware accelerator. Google Colab provides a 12GB NVIDIA Tesla K80 GPU that can be enabled and used for up to 12 hours continuously, which proves very useful when carrying out GPU intensive tasks such as training an object detection model.

2.1.5. Python

I saw this project as a great opportunity to learn and use the Python programming language for the first time. I chose to use Python for this project not only because it would be greatly beneficial to learn for future endeavours, being one of the most widely used languages today, but also because Computer Vision experts recommend it as the ideal language to use for such computer vision. This is because it is quick and easy to learn, and it provides an extensive range of resources and libraries suitable for any computer vision applications, such as OpenCV, TensorFlow and NumPy to deal with the matrix-heavy output of an object detection model.

As I had plenty of previous experience in programming in different languages such as Java, JavaScript and C, I found learning Python and adapting to syntactic differences did not prove to be too much of a challenge after a number of weeks practicing the language. Python was used in this project to write functions that implement object detection models and run them on a variety of input files such as .MP4, .JPG and drone feed, as well as outputting the fps of the running detection model and saving the resulting video to a destination. It was also used to write a script that splits large image datasets by a particular percentage, into training and validation files used for training an object detection model. The IDE that was used to write all of the scripts for this project was PyCharm Professional.

2.1.6. OpenCV & Other Python Libraries

OpenCV is a library that can be imported into a Python project. It provides programming functionality for Machine Learning and real-time Computer Vision applications. It primarily focuses on image processing, video capture and object detection. Features of the OpenCV library that I benefited from the most during this project were:

- The Deep Neural Network (DNN) OpenCV module, to read in and implement the object detection model, i.e. `cv.dnn.readNetFromDarknet(yolo_weights, yolo_config)`.

- Reading and writing to images and video capture to draw bounding boxes around objects detected and displaying the confidence score for each object.
- Accessing the live drone feed via OpenCV's video capture module and an RTMP server.

Another useful library used in this project was NumPy, which adds support for dealing with large multi-dimensional matrices and arrays. This was a necessary library to include in my Python project as YOLO detection layers output a list of NumPy arrays. These arrays include the position, class probability and confidence of a detection. Therefore, this library was useful for tasks such as extracting the maximum probability and confidence scores from a list of NumPy arrays produced by the model.

Other libraries used included the Time module which was useful when calculating the FPS of a video capture. The combination of the Glob and OS python modules were also useful inclusions in this project and were used to get the directory path of the image dataset and write a certain percentage of the images to a different directory path. The Glob module is useful for finding all pathnames that match a specific pattern (i.e. ending with ".jpg") and the OS module was then used to specify a path to a folder to which to write a percentage of the images.

2.1.7. NginX RTMP & DJI GO 4

As this project focused on livestock management using a UAV, I needed means to run the custom object detection model on the live camera feed of a drone. A way of achieving this was to use OpenCV's video capture module, which allows the input of a Real-Time Messaging Protocol (RTMP) streaming server address and then connect my DJI Mavic Air drone to that RTMP server set up on my PC with the DJI GO 4 mobile app.

NginX provide a downloadable application that runs an RTMP server on your local network. The DJI GO 4 app is used to control the drone and it has the option of live streaming the camera feed to an IP address. Connecting the mobile device with the DJI GO 4 app on it to the PC with the RTMP server running, allows the drone to make a direct connection with the PC and can therefore forward the live camera stream to that PC. Using the IP address of the phone connected to a hotspot created on the PC as the input for drone's livestream address and OpenCV's videoCapture module then allows means of running the object detection model on the live feed of the drone-mounted camera.

2.2. Relevance of BCT Course Content to Project

In this subsection, I discuss how elements of my project relate to some of the content studied in various modules of the BCT course and how these modules helped me during the development of my project.

2.2.1. Machine Learning

During my time studying the Machine Learning module in the first semester of my 4th year, I learned about many concepts that directly relate to my project. Topics that were covered in this module included Classification and Regression models, as well as identifying and solving the issue of Overfitting and Underfitting a model. These concepts relate to my project as the object detection model that I chose to implement, adapts a variation of an Artificial Neural Network which was studied as a classification algorithm in this module. Also, YOLO makes its predictions using Logistic Regression which was a concept that was studied in this module that interestingly can be used for both classification and regression models. This module also taught me that overfitting of data occurs if the model performs well on the training data but poorly on unseen data, and underfitting occurs if the model performs poorly on both training and unseen data. I took these concepts into account when training the object detection models.

2.2.2. Graphics & Image Processing

Some of the concepts taught in the CT404 Graphics & Image Processing module also tied in with some of the concepts seen when researching and implementing my custom object detection models. The idea of Non-Maximum Suppression (NMS) ties in with this module as it was a technique discussed during the topic of edge detection. In edge detection NMS is used to only select pixels with intensity over a certain threshold (NMS-threshold) and discard the other pixels effectively providing a more accurate representation of real edges in an image. This relates back to my project as the YOLO models have a built in NMS feature to select the best bounding boxes for a detected object and discard all other bounding boxes under the NMS-threshold.

2.2.3. Programming

While the Python language was not studied in any module of the BCT course, the various programming modules using Java, JavaScript and C throughout the years certainly helped with quickly learning the Python language, writing scripts effectively and safely using programming concepts such as for loops, while loops and if statements learned in these modules.

2.2.4. Software Engineering

The CT5106 Software Engineering module presented useful techniques of project planning and scheduling. In this module, we discussed work breakdown structures and project planning structures that should be considered when starting a project. I took these structures into account and created a Gantt chart using an open-source project management tool, ClickUp, which I included in my Project Definition Document. I populated this Gantt chart with tasks that I devised from a work breakdown structure and gave each task a deadline. The ClickUp tool would send me regular emails when a task deadline is due which would keep me on top of my project plan.

3. Technical Research

I devised a series of research objectives and proposed the following questions to be considered throughout the development and implementation of my project:

1. How long should an object detection model train for?
2. How should an image dataset be split for training/validation purposes and what is an optimal size for such a dataset?
3. How do overlapping and tightly clustered objects impact the performance of the object detection model?
4. How does distance from the object impact the performance of the object detection model?
5. How does the size of a model's architecture impact the FPS when implemented in real-time?

The research of technology for this project involved choosing a suitable object detection algorithm, methods of creating large datasets to train a custom object detection model, methods of implementing such detection models with Python programming and means of running an object detection model on live drone feed.

When researching suitable object detection techniques, I found that the most commonly used techniques were based on convolutional neural networks (CNNs), which I had encountered previously in the Machine Learning module from semester 1. Therefore, I felt it was best to choose this approach to train a custom object detection model. When researching this approach further, I found that there were many different object detection algorithms to choose from, each having their own unique structure with a variety of advantages and disadvantages. *Table 1* contrasts between some of the more popular algorithms used for object detection.

Table 1: Object Detection Model Comparison

Detection Model	Description	Advantages	Disadvantages
R-CNN	<ul style="list-style-type: none">• Region proposal based framework.• Uses Selective search method to extract regions of interest (ROIs) from the input images.• SVM (Support-Vector Machine) classifiers used to determine types of objects detected.	<ul style="list-style-type: none">• Bypasses problem of searching huge number of regions.	<ul style="list-style-type: none">• Multi-stage training process.• Cannot be implemented in real-time.• Slow training time.
SSD (Single Shot Detector)	<ul style="list-style-type: none">• Regression based framework.• Makes class predictions from multiple feature maps with different resolutions.	<ul style="list-style-type: none">• Fast training speed and highly accurate object detection.• Can be implemented in real-time.	<ul style="list-style-type: none">• Poor at predicting small objects.• Needs very large dataset for training.

SPP-Net (Spatial Pyramid Pooling)	<ul style="list-style-type: none"> • Region proposal based framework. • Uses a pooling strategy to eliminate the requirement for the input of fixed-length images. 	<ul style="list-style-type: none"> • Can generate a fixed-length representation regardless of input image size. • Improved accuracy on R-CNN. 	<ul style="list-style-type: none"> • Multi-stage training process. • Slow training time. • Cannot be implemented in real-time.
YOLO	<ul style="list-style-type: none"> • Regression/Classification based framework. • Divides images into SxS grid and each cell predicts presence of a class and corresponding class probabilities. 	<ul style="list-style-type: none"> • Only one step training process. • Fast training time. • Can be implemented in real-time. • A variety of versions. • Most widely used detection algorithm. 	<ul style="list-style-type: none"> • Less accurate than R-CNN. • Difficulty in detecting smaller objects in a group.

After researching a variety of object detection models and comparing their key attributes, I chose YOLO because it is best for real-time object detection which was a necessity for this project. Additionally, as this project was my first experience working with object detection, YOLO was the best choice for me as it had plenty of online learning material, such as tutorial videos and documentation that would prove useful when training and implementing the object detection model. Such documentation would also recommend a number of methods and tools for building suitable image datasets to be used for the YOLO model training such as OIDv4 Tool-Kit and Labellmg.

I also researched methods for connecting my personal drone to my PC in such a way that an object detection model could be ran on the drone-mounted camera. Here I found tutorials for setting up and RTMP server that can be used as a connection from the drone to the PC. Further research into this subject then revealed that the RTMP server address can then be used with OpenCV to stream the live drone feed in the python IDE and apply an object detection model on that stream.

4. Design

This chapter outlines a high-level design overview of the overall livestock management system. The system can be broken into three separate components, the drone communication system, the object detection model (YOLO) training system and the live processing system. Using draw.io software, I designed diagrams which outline a high-level overview of these three components.

4.1. Drone Communication System

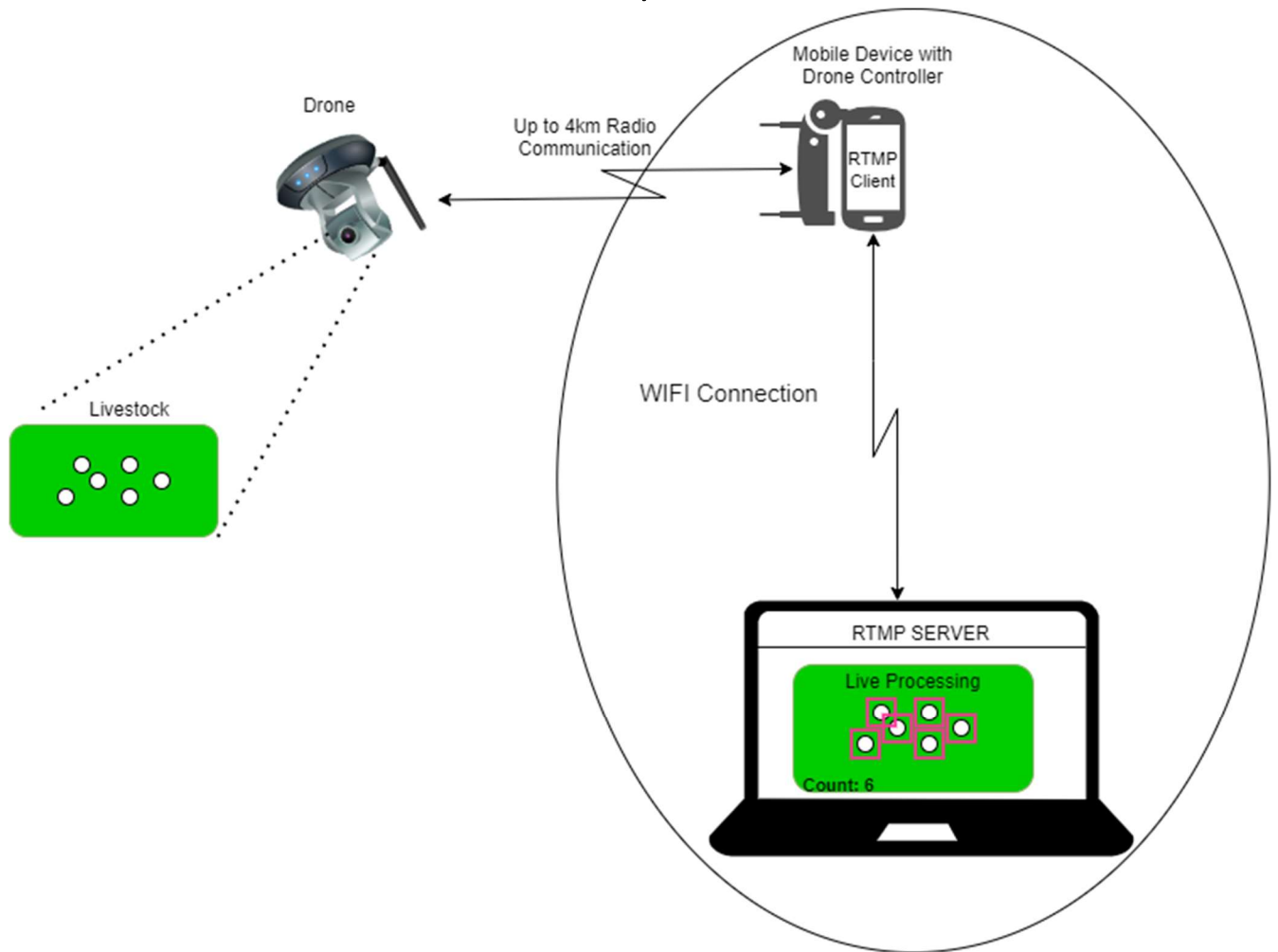


Figure 1: Drone Communication System

Figure 1 outlines how the drone communication system works within this project. The PC and mobile device used to control the drone are connected via WIFI. The PC has an RTMP server running which allows the mobile device to transmit the camera stream of the drone to that PC via an RTMP. While the PC and mobile device with the drone controller attached must be within a certain close range of each other to establish a WIFI connection, the Mavic Air drone that I was using for this project could then fly up to 4km away from the controller while streaming it's live footage to the PC. Live processing can then be done on the drone feed to apply an object detection model and other functionality.

4.2. YOLO Model Training System

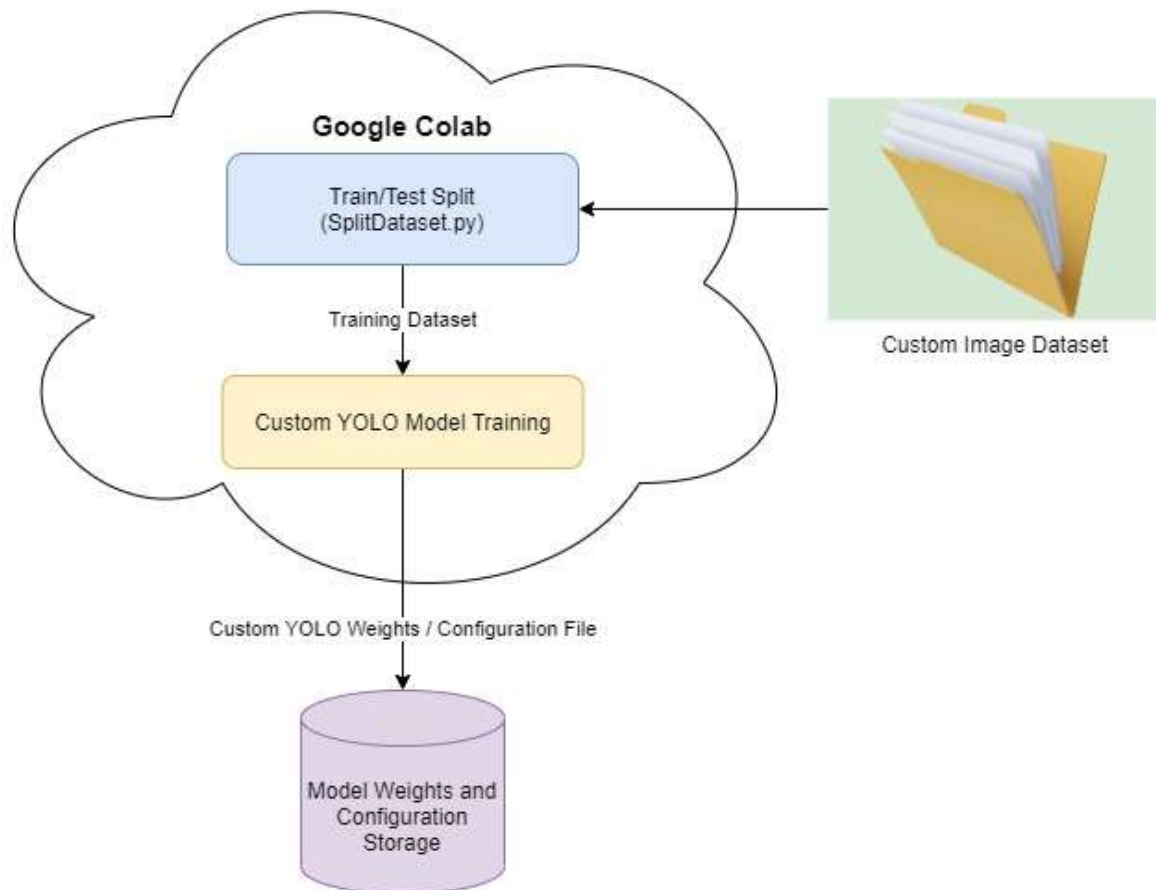


Figure 2: Model Training System

Figure 2 outlines the model training system that was used to train YOLOv3, YOLOv3-tiny and YOLOv4-tiny object detection models. A custom image dataset is created with the OIv4 Tool Kit and is inputted in its entirety to a python function, which can be ran on the Google Colab cloud service. This python script splits the dataset and writes a percentage of the images to two files. 90% of images are written to a training file which YOLO uses to model the data, while the remaining 10% are written to a validation file that YOLO uses to validate the model. The YOLO object detection models are trained on the Google Colab cloud service to utilize free GPU resources. During and after training the model, weights at every 1000 iterations of the CNN and final weights are saved to my Google Drive storage, as well as the configuration file used when running the training model. These files are then accessible to my local machine and ready for use in the live processing stage to run the custom detection model on live drone feed.

4.3. Live Processing System

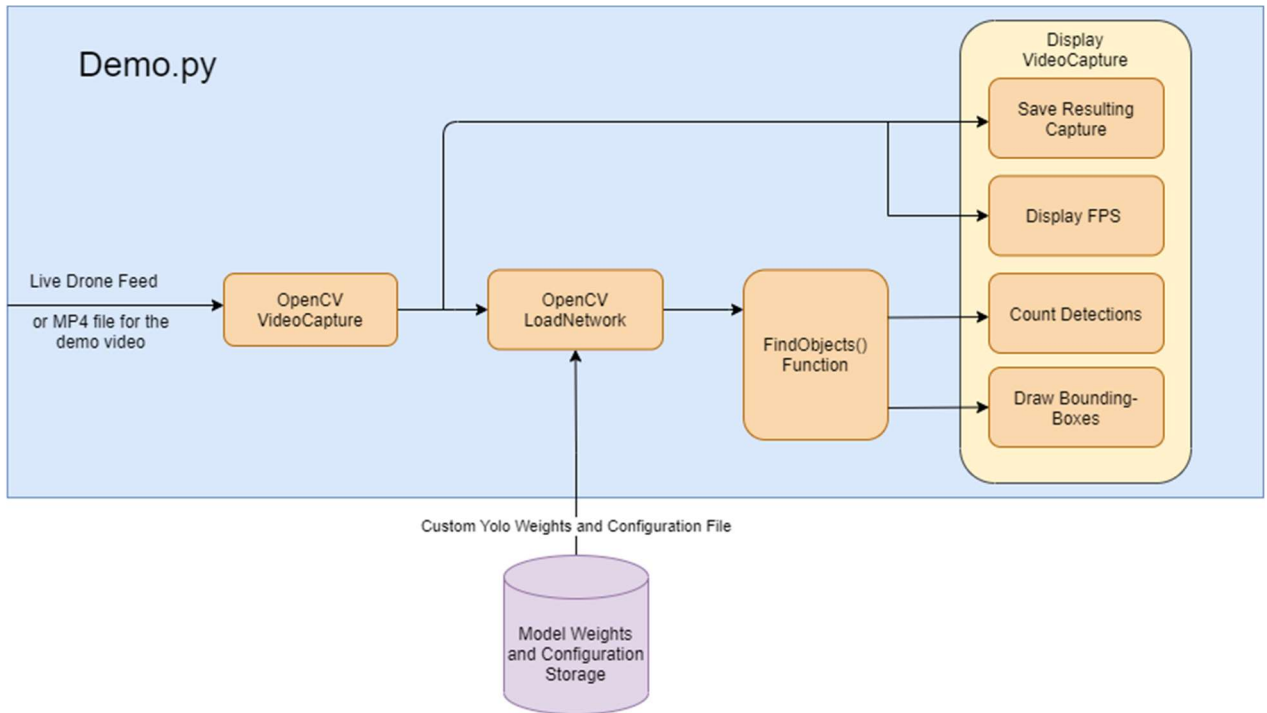


Figure 3: Live Processing System

Figure 3 outlines the main python program (Demo.py) used to implement the output from each of the previous two systems seen in Figure 1 and Figure 2. The IP address of the RTMP connection can be used as an input in this program to display the live drone feed via OpenCV's `videoCapture()` module. The custom YOLO model weights and configuration can be used as an input into this program to load the object detection model with OpenCV's `deep neural network` module. The `findObjects()` function is used in this design to extract the detections in the video capture that are over a certain confidence threshold and to apply NMS on those detections. This function also draws the bounding boxes around each detection and displays the detection count on screen. With the `videoCapture()` module, the frames per second of the live drone feed with the object detection model running on it is calculated and the resulting capture is saved to a .avi file for future analysis.

5. Development & Implementation

In this chapter, I describe in detail, each of the steps taken to develop and implement this livestock management system. These steps included building image datasets, training the detection models and handling detections with OpenCV Python on live drone feed.

5.1. Building a custom Dataset

For training a YOLO object detection model, I first needed an image dataset containing the objects that the model would learn to detect. The YOLO model family read in images to the network as: .jpg image files with an accompanied text file for each image file. Each accompanied text file must contain the following information:

- The object class in the image, from 0 to (classes-1). For example, 0 would represent a dog ,1 would represent a person, 2 would represent a car etc.
- X and Y (float values) coordinates representing the centre of the bounding box placed around an object.
- Width and Height (float values) of the bounding box placed around an object.
- I.e., for each object in the image: <object-class> <x> <y> <width> <height>.
- Example: 0 0.18457 0.6870 0.2760 0.8203

I experimented with two different software tools that can be used to build bounding boxes around objects and generate these text files for images in a dataset. These tools were: Labellmg and OIDv4.

5.1.1. Labellmg & Image Downloader

Initially, I chose Labellmg & Image Downloader software to build a custom image dataset consisting of different poultry birds to be used for training an object detection model. I used the Image Downloader chrome extension to download ~70 images, each, of ducks, turkeys and hens, with a total of 207 images in the dataset. Labellmg, a graphical image notation tool, then allowed me to define classes in this image dataset and manually draw bounding boxes over every poultry animal in every image, while labelling them as their respective classes. Drawing bounding boxes around objects in an image and labelling would automatically generate a text file for that image, populated with the YOLO bounding box format as outlined in the intro to this subchapter, 5.1. After manually drawing bounding boxes and labelling every class in each of the images, the data was then ready for insertion into a YOLOv3 training network.

However, after training the YOLOv3 model on this dataset of 207 images of poultry, I quickly discovered that the size of the dataset was far too small to produce results. As there were only ~70 images per class, when testing the model, it was unable to classify objects or distinguish between objects. After consulting my project supervisor on the issue and researching optimal dataset sizes for object detection, I could confirm that I needed a far bigger dataset with at least 1000 images per class to train a useable detection model.

With these issues in consideration, I decided to change my method of building a dataset and research different techniques because manually labelling 1000s of images with Labellmg would be an extremely tedious and time-consuming task.

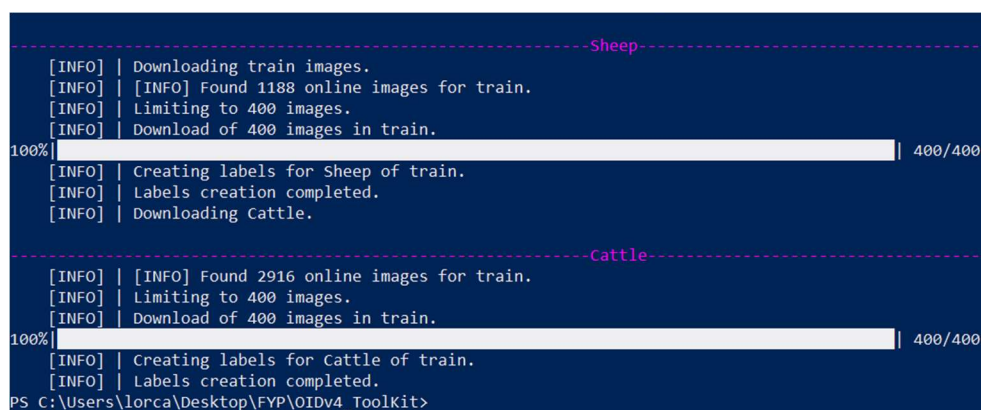
5.1.2. Google Open Images Database & OIDv4 Tool Kit

A more efficient method of building a dataset suitable for training a YOLO detection model was using Google's Open Images database of pre-labelled classes. I also decided to change my target of detecting different poultry types to detecting more profitable livestock animals like cattle and sheep. Using OIDv4, a tool kit for downloading images in bulk from Google's Open Image database, I could effortlessly download a specified amount of images for any class that was available in the database. With the OIDv4 tool downloaded, I could create a dataset consisting of 1500 images of cattle, 1500 images of sheep and YOLO label format text files for each image. This was possible using the following command in Windows Powershell:

```
PS C:\Users\lorca\Desktop\FYP\OIDv4_ToolKit> python main.py downloader --classes Sheep Cattle --type_csv train --limit 1500 --multiclass 1
```

This command runs the main python program of the tool-kit and it takes in arguments such as; the class names that it searches Google Open Images for, the image download limit (1500 per image in this case) and a multi-classes option which specifies whether or not the images for each class should be saved to the same folder.

Figure 4 shows an example of the output that OIDv4 produces after the above command is ran, but instead of --limit 1500, it was --limit 400 in this occasion. It outlines how many images were found in the Open Images database, sets the limit of images to download, downloads the images and creates text files for each image containing the label/bounding box coordinates in YOLO format for each image. The images and corresponding label text files are saved to a data folder in the OIDv4 source folder on the local machine and this data folder can then be used to train YOLO detection models.



```
[INFO] | Downloading train images.
[INFO] | [INFO] Found 1188 online images for train.
[INFO] | Limiting to 400 images.
[INFO] | Download of 400 images in train.
100%|-----| 400/400
[INFO] | Creating labels for Sheep of train.
[INFO] | Labels creation completed.
[INFO] | Downloading Cattle.

-----Cattle-----
[INFO] | [INFO] Found 2916 online images for train.
[INFO] | Limiting to 400 images.
[INFO] | Download of 400 images in train.
100%|-----| 400/400
[INFO] | Creating labels for Cattle of train.
[INFO] | Labels creation completed.
PS C:\Users\lorca\Desktop\FYP\OIDv4_ToolKit>
```

Figure 4: OIDv4 Building Cow/Sheep Dataset

When attempting to use this tool to download 1500 images of cattle and 1500 images of sheep, it succeeded in downloading 1500 images of cattle, however, only 1188 images of sheep were available from the Open Images database. Therefore, for the models trained on this dataset, I took into account when analysing the results that there was a class imbalance in the cattle/sheep dataset.

Additionally, using the same methodology as before, I built another dataset of 2000 labelled images of cattle alone as I wanted to train a model that was optimised for just detecting cattle to compare it against the cattle/sheep model.

5.2. Cloning and Compiling Darknet on the Cloud

In a Google Colab notebook, I cloned the darknet github repository. This repository includes the darknet CNN structure to use as the backbone architecture when training a YOLO model. It also contains a folder of configuration file for all version of YOLO. These configuration files can be edited to suit a custom dataset and used for training the model.

```
!git clone https://github.com/AlexeyAB/darknet #cloning darknet
!make #building darknet
```

In the Colab notebook, I also downloaded pre-trained convolutional layer weights from the github repositories of the creators of YOLO/darknet: *darknet53.conv.74*, which are the optimal layer weights for training YOLOv3 models, and *yolov4-tiny.conv.29* which are the optimal layer weights for training the YOLOv4-tiny model. When training a custom YOLO model with these pre-trained weights, the idea of transfer learning is being used whereby the network is not being trained from absolute scratch, it is utilizing the optimal layer weights of a pre-trained model and applying them to a new custom model configuration file and dataset. The commands below were ran on the Colab notebook to retrieve the pre-trained weights from their respective repositories using the `!wget` command which is a program that retrieves content from a specified URL.

```
!wget http://pjreddie.com/media/files/darknet53.conv.74
```

```
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_
yolo_v4_pre/yolov4-tiny.conv.29
```

5.3. Preparing Data for Training

To prepare my two datasets (cattle/sheep & cattle) for training, I wanted to employ a train/validation split on the data. Separating the dataset into training and validation subsets for training an object detection model is important as it provides an unbiased model evaluation during training and can detect if the model is suffering from underfitting/overfitting. When training the YOLO models, the training subset is the sample of data used to fit the model, whereas the validation subset is the sample of data used to provide an unbiased evaluation of a model fit on the training data. I chose to allocate 10% of the images in a dataset to the validation subset and the remaining 90% to the training subset.

With reference to the github repository of Techzizou (Ref: [12]), I wrote a python script named SplitDataset.py, which is designed to be ran on the Google Colab notebook where training will take place. This script gets the directory to the folder on the Colab notebook that contains the dataset.

```
import glob, os

# Current directory
currentDir = os.path.dirname(os.path.abspath(__file__))
# change directory to folder containing entire dataset
currentDir = 'data/obj'
```

It then defines a percentage to be used to split the dataset into training/validation (i.e. 10%) and writes two text files, train.txt and test.txt.

```
# Percentage of images to be used for validating model (test split)
testSplit = 10; # 10% test, 90% train

# using the open() functions write ('w') feature to create the train and
test files
trainFile = open('data/train.txt', 'w')
testFile = open('data/test.txt', 'w')

# Populating train.txt and test.txt files
counter = 1
# round the test split up to nearest full percentage
indexTest = round(100 / testSplit)
```

It then writes 10% of the jpeg files from the dataset into test.txt and the remaining 90% into train.txt.

```
# if counter in loop equals the test split percentage
if counter == indexTest:
    counter = 1
    # write the jpg files to test.txt
    testFile.write("data/obj" + "/" + title + '.jpg' + "\n")
else:
    #else write the remaining percentage to train.txt
    trainFile.write("data/obj" + "/" + title + '.jpg' + "\n")
    counter = counter + 1
```

This script can then be ran on the Colab notebook to generate the train/validation splits with the following command:

```
!python SplitDataset.py
```

The YOLO network reads in two files before training the model: obj.names and obj.data. Obj.names is a file that contains just the names of the classes being trained, e.g. Cattle and Sheep. The obj.data file contains the path on the colab notebook to the train.txt file and to the test.txt file that were generated with the SplitDataset.py script. Obj.data also contains the number of classes being trained, the path to the obj.names file with the class names in it and the path to a folder on my Google drive to automatically save weights after every 1000 iterations of the training network and also saves the best/final weights of the network. The following screenshot is taken from the obj.data file for the YOLOv4 model which was trained for detecting cattle and sheep:

```
1 classes = 2
2 train = data/train.txt
3 valid = data/test.txt
4 names = data/obj.names
5 backup = /mydrive/Livestock_YOLOv4-tiny/training/
```

5.4. Customising Detection Model Configuration

When training a YOLO object detection model, a configuration file containing the training parameters for the network and the parameters for every layer in the network is read. As I am using transfer learning to build an object detection model, it was necessary to change only the training parameters and a small subset of layer parameters in configuration files from pre-trained YOLO models to suit my custom dataset.

5.4.1. YOLOv3.cfg

YOLOv3 was the largest object detection model that I experimented with in this project. The contents of the configuration file for this model included parameters for 75 convolutional layers, 23 shortcut layers, 4 routing layers and 3 YOLO (detection) layers.

In a pre-trained YOLOv3 model configuration file, I made the following changes to suit my custom dataset of two classes, cattle and sheep in preparation for training a model on that dataset:

- Commented out testing parameters, uncommented training parameters.
- Changed the maximum batch parameter considering the following rule of thumb, `max_batches`: $2000 * \text{<number of classes>} = 4000$.
- Changed the steps parameter to 80% of the `max_batches` and 90% of the `max_batches` = 3200,3600.
- In each of the 3 YOLO (detection) layers, I changed the `classes` parameter to `<number of classes> = 2`.
- In the convolutional layer above each of those 3 YOLO layers, I changed the `filters` parameter considering the following formula for an optimal filter number: $(\text{<number of classes>} + 5) * 3 = 21$.

```

1 [net]
2 # Testing
3 #batch=1
4 #subdivisions=1
5 # Training
6 batch=64
7 subdivisions=16
8 width=416
9 height=416
10 channels=3
11 momentum=0.9
12 decay=0.0005
13 angle=0
14 saturation = 1.5
15 exposure = 1.5
16 hue=.1
17
18 learning_rate=0.001
19 burn_in=1000
20 max_batches = 4000
21 policy=steps
22 steps=3200,3600
23 scales=.1,.1

```

<pre> [convolutional] size=1 stride=1 pad=1 filters=21 activation=linear </pre>	<pre> [convolutional] size=1 stride=1 pad=1 filters=21 activation=linear </pre>	<pre> [convolutional] size=1 stride=1 pad=1 filters=21 activation=linear </pre>
<pre> [yolo] mask = 0,1,2 anchors = 10,13, classes=2 </pre>	<pre> [yolo] mask = 3,4,5 anchors = 10,13, classes=2 </pre>	<pre> [yolo] mask = 6,7,8 anchors = 10,13, classes=2 </pre>

Figure 5: YOLOv3.cfg File

I used the same steps to edit the configuration files of the other YOLO models that I trained, but since they each have their own unique backbone architecture, the structure of their configuration files differed.

5.4.2. YOLOv3-tiny.cfg

YOLOv3-tiny is a compressed version of YOLOv3. The configuration file contains 13 convolutional layers, 6 maxpooling layers, 2 routing layers and 2 YOLO layers.

I edited a pre-trained YOLOv3-tiny configuration file to suit my custom dataset using the same steps as outlined previously for the YOLOv3 configuration file. However, since this is a more compressed version, there were just 2 YOLO layers rather than 3 to edit, alongside the convolutional layers above those YOLO layers.

5.4.3. YOLOv4-tiny.cfg

The YOLOv4-tiny model is more optimized than the YOLOv3-tiny model and is capable of producing more accurate results. This model has a different backbone architecture than the YOLOv3 and YOLOv3-tiny models and it is a compressed version of the YOLOv4 model. The configuration file for this model contains the parameters for 21 convolutional layers, 3 maxpooling layers, 11 routing layers and 2 yolo layers.

I edited a pre-trained YOLOv4-tiny configuration file to suit my custom dataset of cattle and sheep using the same steps as before with the YOLOv3 and YOLOv3-tiny configuration files.

During the testing phase of the project, I could observe that the YOLOv4-tiny model was producing the best overall performance out of the three YOLO versions. Therefore, I decided to train another custom YOLOv4-tiny model on a different dataset containing 2000 images of cattle alone. When constructing a configuration file suited to this dataset, I edited the configuration file of the previous YOLOv4-tiny model to suit the single-class dataset as follows:

- Changed the maximum batch parameter considering the following rule of thumb: $\text{max_batches} = 2000 * \text{<number of classes>} = 2000$.
- Changed the steps parameters to 80% of the max_batches and 90% if the max_batches = 1600,1800.
- In each of the 2 YOLO layers, I changed the classes parameter to $\text{<number of classes>} = 1$.
- In the convolutional layer above each of those 2 YOLO layers, I changed the filters parameter considering the following formula for an optimal filter number: $(\text{<number of classes>} + 5) * 3 = 18$.

5.5. Training Detection Models

With the pre-trained model weights downloaded, the dataset split into training and validation subsets and the pre-trained model configuration file customised to suit the datasets, the custom detection models were ready to train on the Google Colab notebook with GPU set as the hardware accelerator.

Training a model can be done in a single command line in a cell on Google Colab by calling darknet.exe followed by a series of arguments including the obj.data file containing the number of classes and train/validation files, the custom YOLO configuration file, the pre-trained YOLO model weights and an optional set of flags. The -map flag calculates the mean Average Precision (mAP) of the model at various iterations when training, this calculation is a popular method of evaluating the accuracy of a model.

The following commands were ran on separate Colab notebooks to train the different YOLO object detection model versions:

```
# training yolov3 custom detector
!./darknet detector train data/obj.data cfg/yolov3_custom.cfg darknet53.conv.74 -dont_show -map
```

The above command was used to train a custom YOLOv3 model on the cattle/sheep dataset using a custom configuration file and the weights of a pre-trained YOLOv3 model, named darknet53.conv.74.

```
# training yolov3-tiny detector
!./darknet detector train data/obj.data cfg/yolov3tiny_custom.cfg yolov3-tiny.conv.15 -dont_show -map
```

The above command was used to train a custom YOLOv3-tiny model on the cattle/sheep dataset using a custom configuration file and the weights of a pre-trained YOLOv3-tiny model, named yolov3-tiny.conv.15.

```
# training yolov4-tiny detector
!./darknet detector train data/obj.data cfg/yolov4-tiny-custom.cfg
yolov4-tiny.conv.29 -dont_show -map
```

The above command was used on two different occasions, to train a YOLOv4-tiny detection model on a cattle/sheep dataset, and to train a YOLOv4-tiny detection model on a single-class (cattle) dataset.

After the training has reached its maximum iteration number, the final weights for each model are saved to a backup folder that was specified in the obj.data file. These weights along with their corresponding configuration files can then be inputted to OpenCV's deep neural network module to test the detection model on a variety of video capture types such as MP4 or live drone feed.

5.6. Handling Detections with OpenCV

After the custom YOLO weights and configuration were generated from the training phase, I wrote a Python script, with reference to CV ZONE's Object Detection YOLO course (Ref: [15]), to implement the model and test it on various inputs including live drone feed.

5.6.1. Getting Live Drone Feed

Running the NginX RTMP server on my local PC allowed a connection to be made between the PC and DJI Mavic Air drone. The DJI GO 4 controller app could then live stream the drone-mounted camera feed over the RTMP address. Using this address as an input for OpenCV's *VideoCapture* module, the drone camera feed could be displayed and processed live in the PyCharm environment. For testing and demonstration purposes, I have also included in the code block below, other video capture types that were used including MP4 files, JPG files and webcam feed for more accessible real-time testing.

```
cap = cv2.VideoCapture('rtmp://192.168.137.1/live/drone') # Live drone feed
#cap = cv2.VideoCapture(0) # Webcam feed
#cap = cv2.VideoCapture('Data/spacedCows.jpg') # jpg file
#cap = cv2.VideoCapture('Data/grazing-cows.mp4') # .mp4 file
```

I refer to the following while loop as the script's main while loop. When running the script, this while loop reads the video capture. The function for detecting objects is called in this while loop as well as the method for displaying the fps and saving the resulting video, which I will discuss further in later subchapters.

```
#main while loop that gets the frames of our cam
while True:
    success, img = cap.read()
    if success == True:
```

OpenCV's *imshow()* method is called in the main while loop, which displays the video capture in a window named 'Drone Footage'. While the window is open, pressing the 'e' key will close the window, otherwise, stopping the script will close the window.


```

cv2.imshow('Drone Footage', img)
    #delay cam for 1 milisec, and e to exit
    if cv2.waitKey(1) & 0xFF == ord('e'):
        break
    else:
        break
cap.release()
cv2.destroyAllWindows()

```

5.6.2. Loading Custom Detection Model

The weights and configuration files for the custom YOLO detection models are used in the script here. Using OpenCV's deep neural network module, `dnn.readNetFromDarknet()`, I could load the detection network, inputting the weights and configuration files generated from training the custom models.

```

# yolo model configuration file path
yolo_config = 'YOLOv4-models/COWyolov4-tiny-custom.cfg'
# yolo model trained weights file path
yolo_weights = 'YOLOv4-models/COWyolov4-tiny-custom_best.weights'
# using opencv's deep neural network (dnn) module to create the darknet
network
net = cv2.dnn.readNetFromDarknet(yolo_config, yolo_weights)

```

Rather than inputting the plain image from the drone footage or MP4 file to the network, this image must be converted to a particular format. The network only accepts blob formatting. In the main while loop, I convert the video capture image to blob format. OpenCV's `dnn.blobFromImage()` method takes in the image and its width/height, I left the other parameters as default in this method and converts the image to blob format.

```

blob = cv2.dnn.blobFromImage(img, 1/255, (whT, whT), [0,0,0], 1, crop=False)
net.setInput(blob)

```

5.6.3 Finding Detections

The function, `findObjects(outputs, img)`, takes in the output arrays from the YOLO model detection layers and the video capture that the model is running on. To better understand this function, *table 2* outlines an example of the output NumPy arrays that are produced when the detection model is running on a video capture.

Table 2: Output NumPy Array from the Network Detection Layers

	index[0]	index[1]	index[2]	index[3]	index[4]	index[5]	index[6]
BBox Number (Bounding box number)	Cx (Centre x point of bbox)	Cy (Centre y point of bbox)	w (Width of bbox)	h (Height of bbox)	Confidence (Confidence score for object being inside the bbox)	ClassID: 0 (Sheep) Probability (Probability that the object is a sheep)	ClassID:1 (Cow) Probability (Probability that the object is a cow)
0	0.252	0.452	0.504	0.904	0.88	0.04	0.92
...							
n	0.24	0.401	0.502	0.899	0.80	0	0.90

In the function, variables are defined that represent the shape of the image as well as lists to store bounding box values (indices [0] to [3] in output array), confidence scores (index [4] in array) and class ID probabilities (indices [5], [6]).

```
def findObjects(outputs,img):
    # getting height, width and channels of the image
    hT, wT, cT, = img.shape
    # whenever a good detection is found, the values will be stored in
    these lists
    # list to contain (bounding box) values for cx, cy, w, h
    bbox = []
    # list to contain all class id probabilities
    classIds = []
    # list to contain confidence values
    confs = []
```

The next step was to loop through the output layers of the network and find the class IDs with the largest probability value and set their corresponding confidence scores. The *scores* variable here defined all indices after the 5th index in the output arrays. These indices represent the class ID probabilities. Using the NumPy method *argmax()*, the class with the largest probability value is defined as *classID*. The confidence score can then be defined as this *classID* probability.

```
for output in outputs:
    # for loop on detections in output (outputs contain the array of values
    for each detection)
    for det in output:
        # moving past the first 5 (cx, cy, w, h, conf) elements in NumPy
        arrays to access the class id probabilities
        scores = det[5:]
        # define classID as the class with the largest probability
        classId = np.argmax(scores)
        # confidence is the class with the max probability
        confidence = scores[classId]
```

After finding the class with the highest probability score and setting it as the confidence score for that detection, the detections had to then be filtered with a confidence threshold to only store the best detections. This confidence threshold was defined outside of the function and when testing the detection models, I would experiment with different values for this threshold in order to find the optimal value for finding every detection in the video capture. This tended to be around 0.4. If the confidence value of a detection was above the threshold, the bounding box information of that detection (i.e. indices [0] to [3] in output array) would be stored in the bounding box list to be used later for drawing the boxes. The *classID* of the detection would then be stored in the *classIDs* list and the confidence score of the detection would be stored in the *confs* list.

```
if confidence > confidence_threshold:
    #save the w and h values (3rd and 4th element in the NumPy arrays),
    multiply by actual w and h of the image to get (int)pixel values
    w,h = int(det[2]*wT) , int(det[3]*hT)
    # save the cx and cy values (1st and 2nd element in the NumPy arrays),
    divide w and h by 2 and subtract from cx,cy to get centre point of image,
    (int) pixel values
    cx,cy = int((det[0]*wT) - w/2), int((det[1]*hT) - h/2)
    # add the good detections to the bbox list
    bbox.append([cx,cy,w,h])
    # add the classID of the detection to the classIDs list
    classIds.append(classId)
```

```
# add the confidence scores to the confs list
confs.append(float(confidence))
```

5.6.4. Drawing Bounding Boxes and Counting Objects

OpenCV's deep neural network module has a method that can perform Non-Maximum Suppression (NMS) on a set of bounding boxes. The method takes in the parameters; list of bounding boxes, list of corresponding confidences, a confidence threshold and a NMS threshold. Using the inputted parameters, this method eliminates overlapping bounding boxes. It picks the bounding box with the maximum confidence score and suppresses all the non-maximum boxes. The NMS threshold, also known as the intersection over union threshold, is the value that measures the overlap of a predicted bounding box for a detection versus the actual bounding box for a detection.

```
## which of the bounding boxes we want to keep
indices = cv2.dnn.NMSBoxes(bbox, confs, confidence_threshold, nms_threshold)
```

For every bounding box with NMS applied, a rectangle is drawn using OpenCV's *rectangle()* method which takes in the video capture, the bounding box centre points, the bounding box width and heights, the colour for the rectangles and the thickness of the rectangle border.

```
for i in indices:
    # take first element, i.e. remove square brackets
    i = i[0]
    box = bbox[i]
    cx, cy, w, h = box[0], box[1], box[2], box[3]
    # drawing rectangles for highlighting detections in the image
    rectangle = cv2.rectangle(img, (cx, cy), (cx+w, cy+h), (255, 0, 255), 2)
```

Using OpenCV's *putText()* method, the confidence score is then placed on top of the rectangles drawn for each bounding box. Furthermore, I defined a new list named *rectangles*, and appended the rectangles drawn for every bounding box to that list. Using another *putText()* method, I displayed the length of this rectangles list as a way of counting the objects detected in the video capture at every frame.

```
# writing text: image window, output: class name and confidence score *100,
position (above box), font, font-scale, color, thickness
cv2.putText(img, f'{classNames[classIds[i]].upper() }
{int(confs[i]*100)}%', (cx+50, cy-
10), cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 0, 255), 1)

# storing rectangles drawn in an list for counting
rectangles.append(rectangle)
cv2.putText(img, 'Livestock Count: ' + str(len(rectangles)), (7, 70),
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (100, 255, 0), 2)
```

5.6.5. Applying Detection Model to Drone Feed

With the *findObjects()* function written, I could then produce the output NumPy arrays of the detection model in the main while loop and feed them to the *findObjects()* function.

As the YOLO object detection networks are made up of many different layers, I had to extract only the YOLO (detection) layers from the networks to use as the output vectors. OpenCV's *getLayerNames()* method gets the name of all layers in the network and the *getUnconnectedOutLayers()* method gets

the index of the output YOLO layers. Something that I noted with the *getUnconnectedOutLayers()* method is that the indices outputted do not take the 0 index into account, so the indices for the output layers are incorrect by one value. Therefore, when getting the output layer indices via a for loop, I subtracted one value from the index in the loop. By printing the values from the loop I could verify that the indices produced did indeed represent the YOLO (detection) layers.

```
layerNames = net.getLayerNames()
## Getting index of the yolo layers. getUnconnectedOutLayers
retrieves indexes of all layers with unconnected output
# print(net.getUnconnectedOutLayers())
outputNames = [layerNames[i[0]-1] for i in
net.getUnconnectedOutLayers()]
#print(outputNames)
```

OpenCV's *forward()* method can then take in the indices of the output layers, creating a forward pass to the network and returning the output NumPy arrays from the output layer indices. These output arrays are then passed to the *findObjects()* function in the main while loop, along with the video capture. This effectively applies the YOLO object detection model on the live drone stream which has been passed to the video capture method via an RTMP address.

```
# the output layers are lists of NumPy arrays (matrices)
outputs = net.forward(outputNames)
findObjects(outputs,img)
```

5.6.7. Getting FPS of Video Capture

I defined two variables at the start of the script that were used to get the fps of the video capture.

```
## FPS Variables
# used to record the time when we processed last frame
prev_frame_time = 0
# used to record the time at which we processed current frame
new_frame_time = 0
```

In the main while loop, while the object detection model is applied to the video capture, the fps can be calculated. The *new_frame_time* variable represents the time this frame finished processing. The *fps* variable then represents the number of frames processed within a given time frame. The *fps* variable must then be converted to a string to use with the *putText()* method, which displays the fps positioned in the top right corner of the video capture window.

```
# time the frame finished processing
new_frame_time = time.time()
# fps is number of frame processed in given time
fps = 1 / (new_frame_time - prev_frame_time)
prev_frame_time = new_frame_time
# fps into integer
fps = int(fps)
# fps to string for putText()
fps = str(fps)
# configuring putText function: window, output, pos, font, font-scale,
color, thickness
```

```
cv2.putText(img, 'FPS: '+fps, (7, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (100, 255, 0), 2)
```

5.6.8. Saving Output Video Capture

After applying the detection model and FPS to the video capture, I wanted means to save the resulting video to my project folder to use for analysis and comparison with other detection models. The result variable represents OpenCV's *VideoWriter()* method which takes in the parameters: name of video output file, the video codec type (MJPG results in a higher resolution video), the number of fps the video is saved in and the size of the window frame.

```
size = (frame_width, frame_height)
result = cv2.VideoWriter('resultingVideo.avi',
cv2.VideoWriter_fourcc(*'MJPG'),10, size)
```

Then in the main while loop, the capture is written to the video writer. When the while loops breaks or is exited via the 'e' key, the resulting capture is saved as a .avi file to the project folder.

```
result.write(img)
    cv2.imshow('video', img)
    #delay cam for 1 milisec, and e to exit
    if cv2.waitKey(1) & 0xFF == ord('e'):
        break
    else:
        break
cap.release()
result.release()
# Closes all the frames
cv2.destroyAllWindows()
print("The video was successfully saved")
```

6. Testing & Results

This chapter outlines how testing was done for each of the models trained and the results of those tests in terms of accuracy and FPS. Additionally, the results of the training phase for each model will be discussed here in terms of the mean Average Precision (mAP) and average loss of the models after training. Mean Average Precision is a popular metric used in machine learning to calculate the accuracy of a model by taking the mean Average Precision over all classes being trained. A loss function is used to determine how well the YOLO algorithm models the given dataset. YOLO calculates the average loss of a model during every iteration of the network. At every iteration, the loss function gradually learns to reduce error in predictions. A large average loss value (>2) after training would indicate that the predicted classifications deviate too much from actual classifications.

6.1. Testing

To test this livestock management system with a UAV project, each custom object detection model was inputted to the Demo.py script and ran on images and real-time drone footage. Before testing the custom object detection models in real-time, I first tested each of them on the same collection of images to observe the classification accuracy. When testing the models on images, I ensured to select images that contain the objects in both the foreground and background to observe how distance impacts the classification and confidence score of an object. Another observation that I wished to make when testing the models on images was how well they could classify overlapping objects. I could then experiment with different values for both the confidence threshold and the Non-Maximum suppression threshold in the Demo.py script to approximate the best values for both thresholds before applying the models to real-time footage. *Figure 5* shows the images that were used for testing each model.



Figure 6: Images Used for Testing

6.2. YOLOv3 Results

The YOLOv3 network had the most complex architecture out of all the YOLO versions that were trained in this project. Therefore, it took far longer to train this model than the other ‘tiny’ YOLO models. The training process of the YOLOv3 on the cattle/sheep dataset spanned across two days. Unfortunately, Google Colab disconnects users from a notebook runtime and wipes all data in that notebook when inactive for more than 90 minutes. Subsequently, I could not leave the model training overnight. Instead, I split the training process in half, letting the model train to 2000 iterations one day, then using the weights trained at 2000 iterations to train the rest of the model the next day, as a result the loss and mean Average Precision chart is in two halves in *Figure 6*. Another issue that I faced with

Google's Colab cloud service was the GPU usage limit. Users can utilize the free GPU provided by Google for up to 12 hours continuously, when that limit is reached, the cooldown period before GPU is accessible again can extend anywhere from 6 hours to 5 days. Therefore, I shared the Colab notebook used to train the model with a backup Google account so when the GPU usage limit was reached, I simply switched to the backup account to finish the training process.

The YOLOv3 model was trained on the cattle/sheep dataset for over 10 hours before the maximum iteration number was reached. *Figure 6* displays the chart of the mean average precision of the model and the average loss of the model at every iteration of the network. The mean average precision reached 60% and the average loss value was 0.8725 at the end of training.

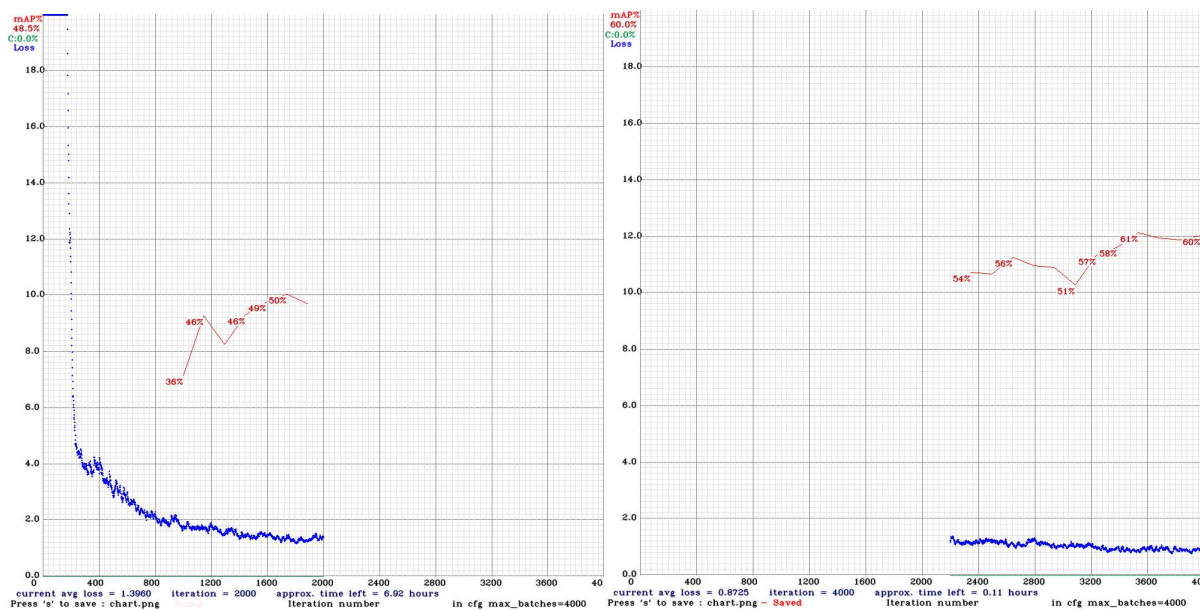
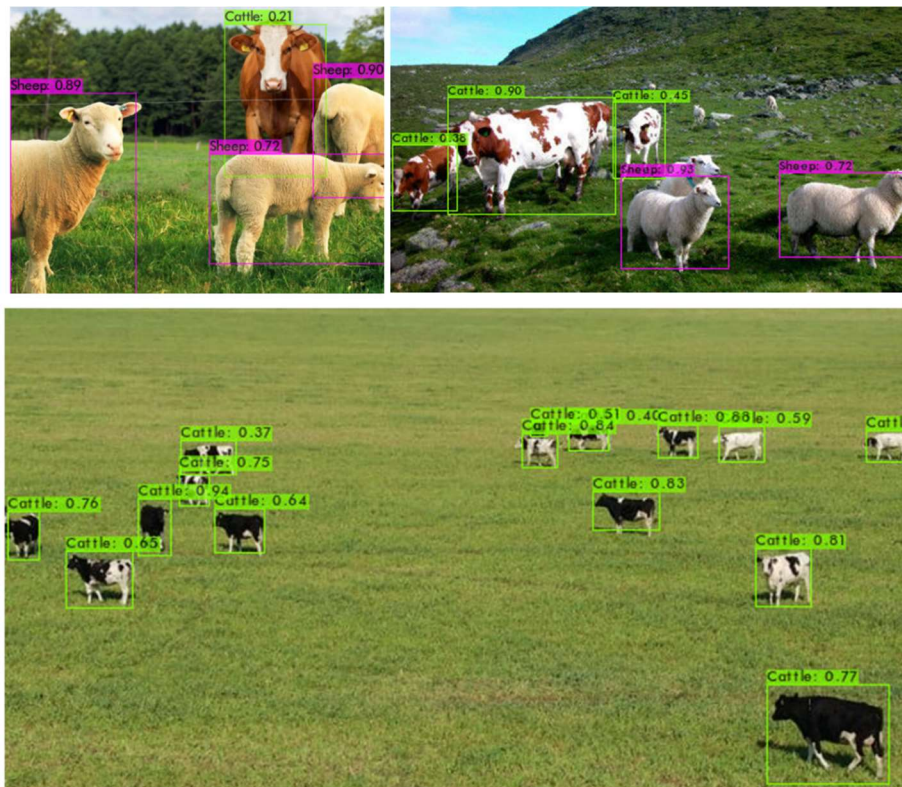


Figure 7: YOLOv3 Loss & mAP Chart

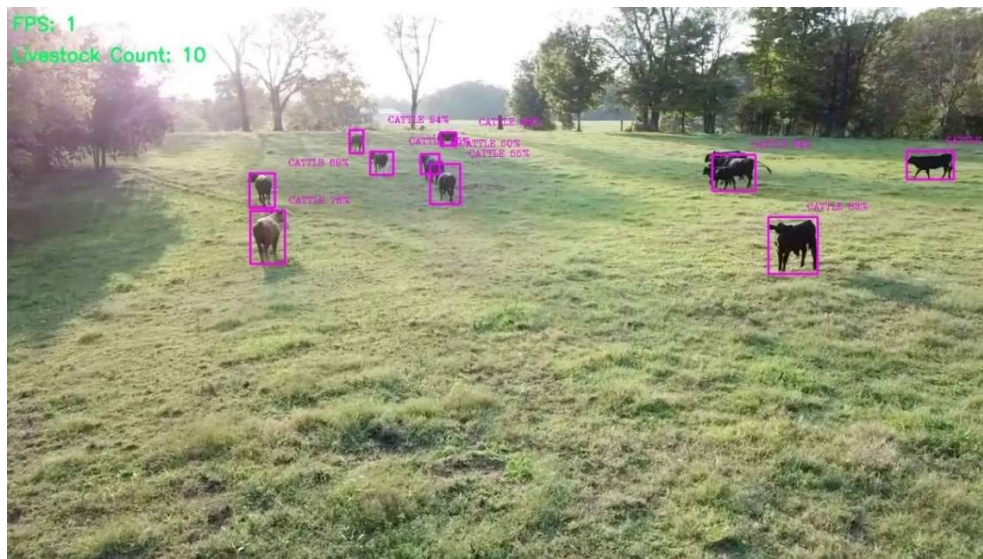
Figure 7 shows the results of the YOLOv3 model when ran on the collection of images. The model performed quite well when predicting sheep in the foreground of the images, however it failed to classify some sheep in the background and also failed to detect the overlapped sheep. The likely reason for this is the imbalance in the dataset as it contained 1500 images of cows and 1188 images of sheep. When classifying cattle in the images, the model performed quite consistently in terms of detecting every cattle in the image regardless of the distance from the camera. However, the confidence score of these detections varied. From these results, it can be observed that the cattle facing the camera are given lower confidence values. Again, this is likely because most of the images of cattle in the dataset were taken from a side view of the cattle.

Figure 8: Results of YOLOv3 on Test Images



When applying the YOLOv3 model to the real-time drone camera, I recorded mixed results. Since the goal of the project was to detect the livestock in real-time with a UAV, the frames per second in which the model was running on the drone camera stream was a key result to record when testing the model. *Figure 8* is a screenshot taken from the YOLOv3 model running on real-time drone footage. The model had trouble detecting some overlapping cattle, as tightly clustered cattle would often be classified as one object instance. However, cattle that were clustered more loosely were correctly classified as individual objects. The model performed very well when classifying the cattle at a greater distance from the drone's camera, giving them high confidence scores. Additionally, the model would rarely misclassify objects, whereby cattle seen in the drone feed would be classified as sheep and vice versa. While I was quite satisfied with the accuracy of the YOLOv3 model, the fps of the drone camera stream dropped significantly to 1 fps after applying this model to it, which made testing the model on real-time drone feed an unpleasantly long task. This is because CPU was used rather than GPU to run the detection model in the PyCharm environment, as the PC that was used for the project did not have a GPU.

Figure 9: Results of YOLOv3 in Real-Time



6.3. YOLOv3-tiny Results

As the YOLOv3 model performed very slowly in terms of FPS when ran in real-time, I decided to train a YOLOv3-tiny model which is a compressed version of YOLOv3 suitable for CPU environments. This compressed version produces a much faster model but sacrifices accuracy for better FPS. The YOLOv3-tiny model took far less time to train than the YOLOv3 model. The YOLOv3-tiny model reached its maximum of 4000 iterations after 1 hour and 28 minutes, whereas the YOLOv3 model took 10 hours and 5 minutes to train to the maximum of 4000 iterations. However, the mAP of the YOLOv3-tiny model was 41.7%, while the YOLOv3 model reached a mAP precision of 60%.

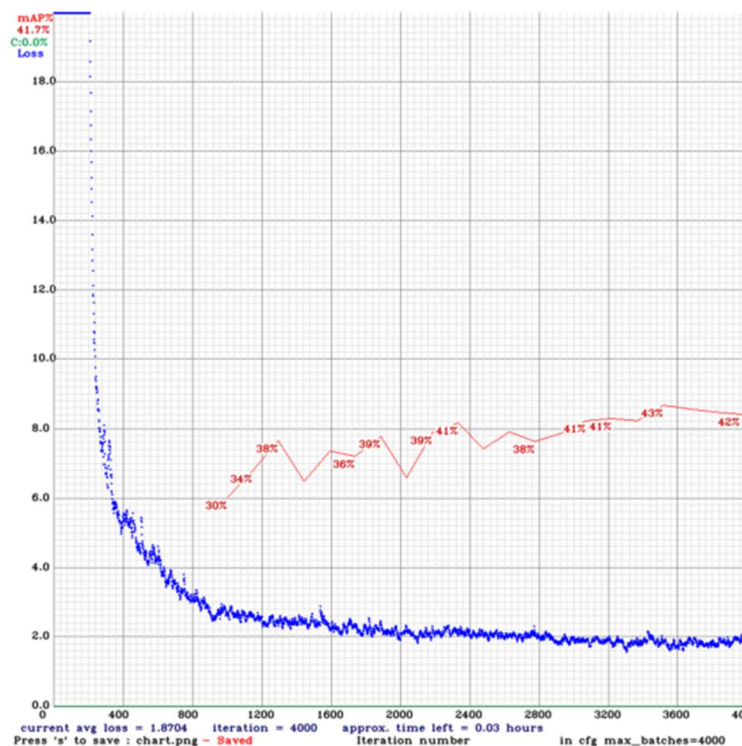
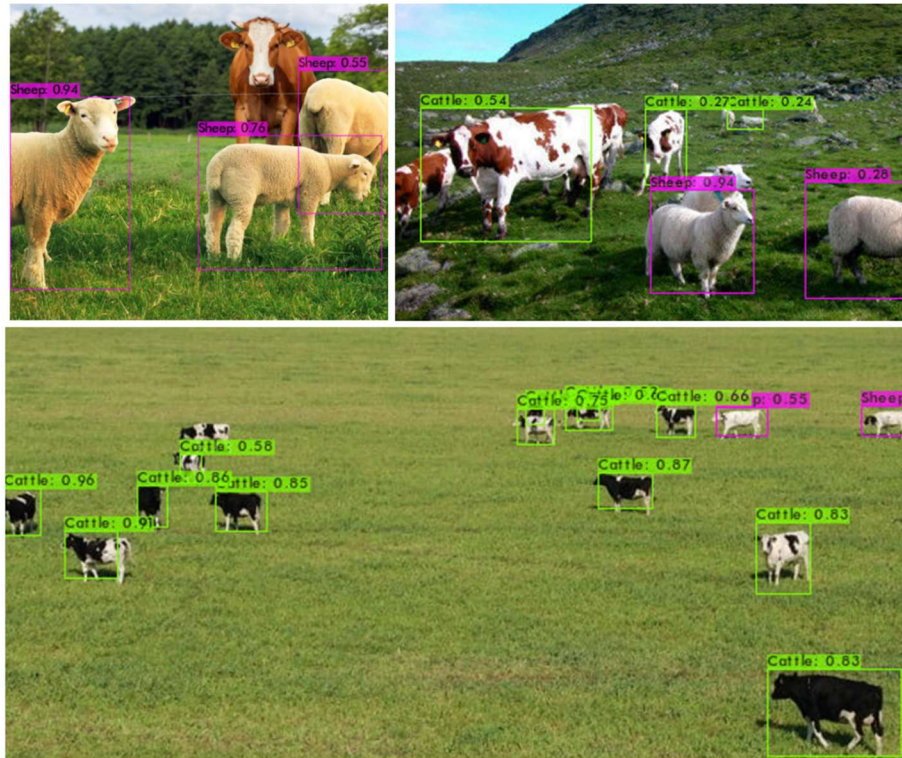


Figure 10: YOLOv3-tiny Loss & mAP Chart

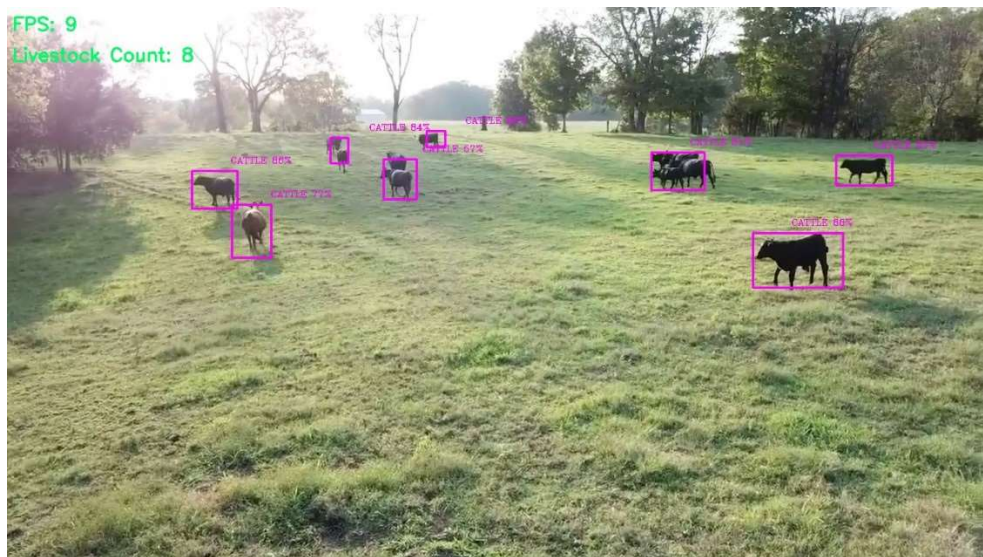
In terms of the accuracy, the YOLOv3-tiny model did not perform as well as the YOLOv3 model, as expected. When ran on the set of test images, as seen in *Figure 10*, misclassification happened more often than with the YOLOv3 model, as some cows were classified as sheep and vice versa. The model would also fail to detect some objects present in the images. The objects that were misclassified or undetected in the images were either located in the background of the image or partially overlapped by another object.

Figure 11: Result of YOLOv3-tiny on Test Images



When running the YOLOv3-tiny model in real-time, the model struggled mainly with overlapping objects. As seen in *Figure 11*, objects in the drone camera stream that were even slightly overlapping were often classified as one object, especially when located further away from the drone. Furthermore, objects that were correctly classified were generally given a smaller confidence score using this model in comparison with the YOLOv3 model. A positive result that was recorded from training this YOLOv3-tiny model was that the FPS of the real-time drone feed with the model applied was greatly improved from the YOLOv3 model. The fps ranged from 8 fps to 11 fps, which made it a lot easier to test on real-time drone feed.

Figure 12: Result of YOLOv3-tiny in Real-Time



Additionally, in an initial attempt at training a YOLOv3-tiny model, the pre-trained weights of the larger YOLOv3 model were mistakenly used instead of pre-trained YOLOv3-tiny weights. Training a model using a YOLOv3-tiny configuration file combined with the pre-trained weights of the larger YOLOv3 resulted in a very inaccurate model. Figure 12 displays the high loss curve and low mean average precision of this erroneous model. This is an example of an underfit model where the loss function produces a much noisier curve of relatively high loss, indicating that the model is struggling to learn the training dataset.

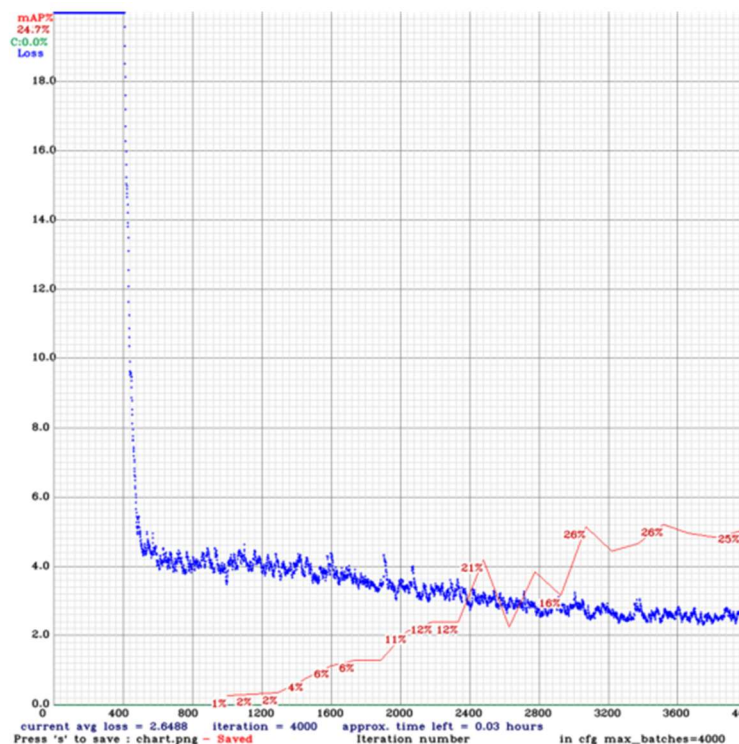


Figure 13: YOLOv3-tiny (large weights) Loss & mAP Chart

6.4. YOLOv4-tiny Results

My goal when training the YOLOv4-tiny model was to find a balance between the performance of previous models, whereby the accuracy improved upon the YOLOv3-tiny model and the real-time fps performance improved upon the YOLOv3 model. This is essentially what I got when training the YOLOv4-tiny model on the cattle/sheep dataset. *Figure 13* shows that the mAP of the model reached a 56.7% which was an improvement on the mAP of the YOLOv3-tiny model at 41.7%. Also, the average loss of the YOLOv4-tiny after 4000 iterations was 0.7906, a big improvement on the average loss of the YOLOv3-tiny model after 4000 iterations, which was 1.8704. As the network architecture for the YOLOv4-tiny model is slightly bigger than the YOLOv3-tiny model, it took a bit longer for the model to train to its maximum iterations, which was 1 hour and 40 minutes, 12 minutes faster than YOLOv3-tiny.

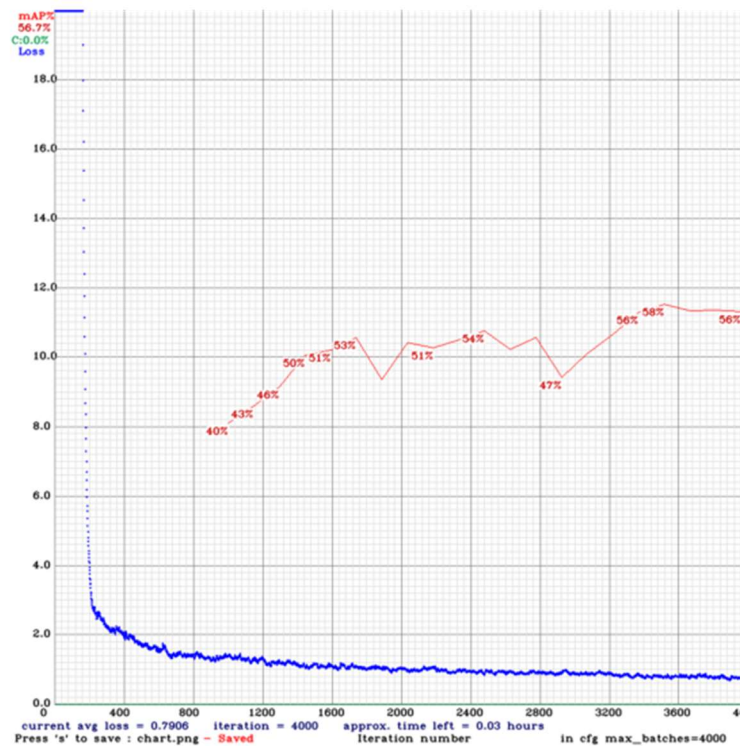
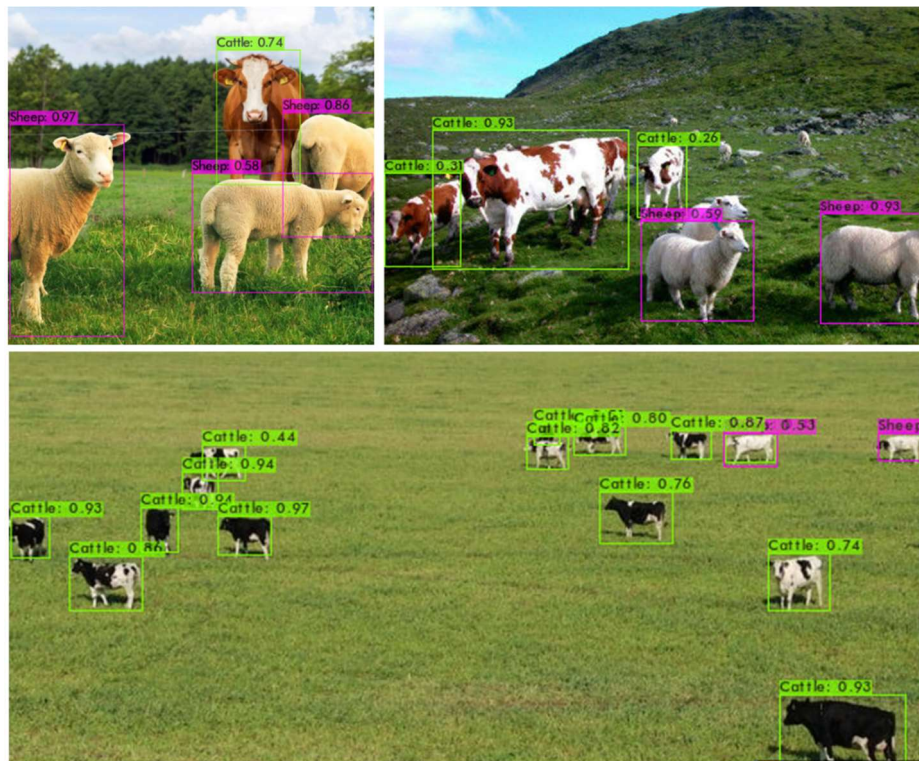


Figure 14: YOLOv4-tiny (cattle/sheep) Loss & mAP Chart

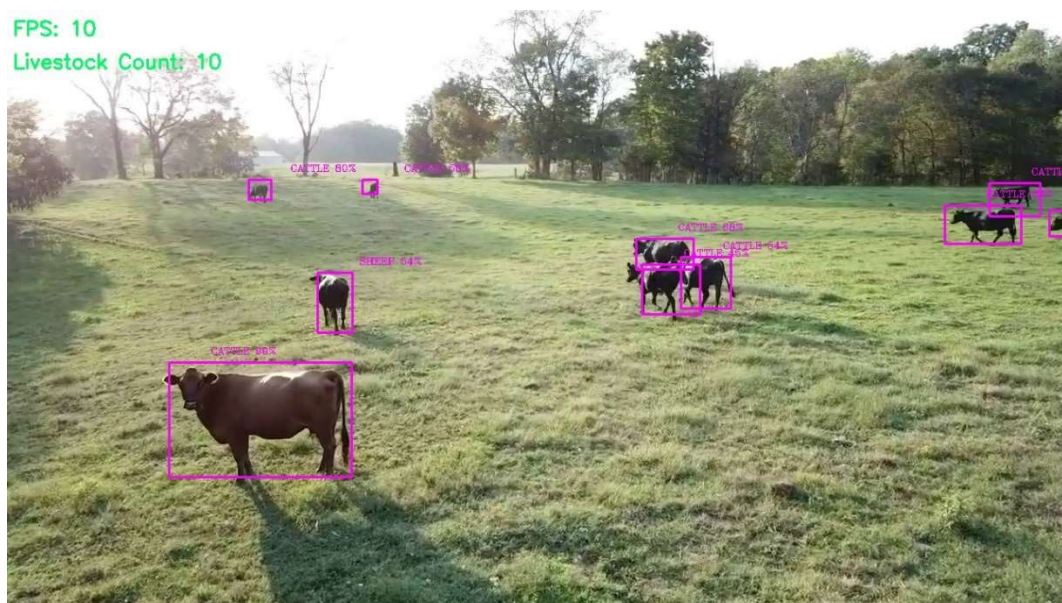
When running the YOLOv4-tiny model on the collection of test images, improvements could be observed as some background objects that went previously undetected were now classified by the model. As can be seen in *Figure 14*, it was evident that this model performed better on overlapping objects as well as further away objects, than the YOLOv3-tiny model did, giving higher confidence scores to objects that were positioned far away from the camera and objects that were being overlapped by others.

Figure 15: Result of YOLOv4-tiny on Test Images



The YOLOv4-tiny model had the best overall performance out of all models when ran in real-time. *Figure 15* displays a capture taken from the model running on drone footage. Unlike previous models, YOLOv4-tiny produced accurate classification of tightly clustered objects. The fps of the drone feed with the model running on it ranged from 7 fps to 11 fps, a frame lesser than the YOLOv3-tiny model but much greater than the YOLOv3 model. However, a downside of this model is that it produced slightly more regular misclassifications than previous models.

Figure 16: Result of YOLOv4-tiny in Real-Time



As the results of the YOLOv4-tiny model on the cattle/sheep dataset were satisfactory apart from the odd misclassification, I decided to train another YOLOv4-tiny model on a different dataset consisting of 2000 images of cows alone. This model was quicker to train than the previous YOLOv4-tiny model and the YOLOv3-tiny model as the dataset used for training was 688 images smaller. There was also just one class to train for rather than two previously, which had an impact on the mAP of the model. After training to the max iterations, which was 2000 in this case, the model reached a mAP of 65.2% which makes it the most accurate of the models trained in this project.

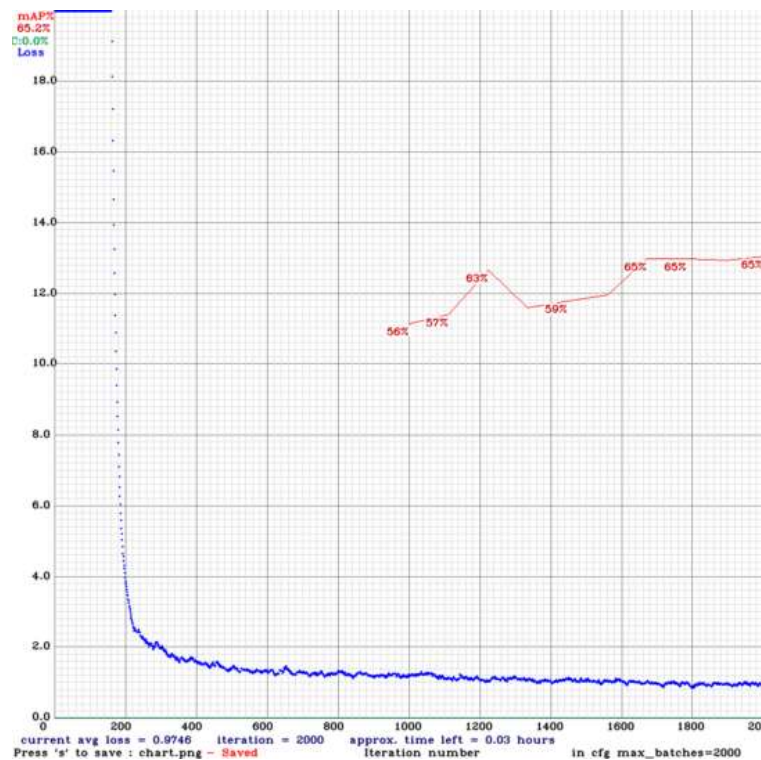


Figure 17: YOLOv4-tiny (cattle) Loss & mAP Chart

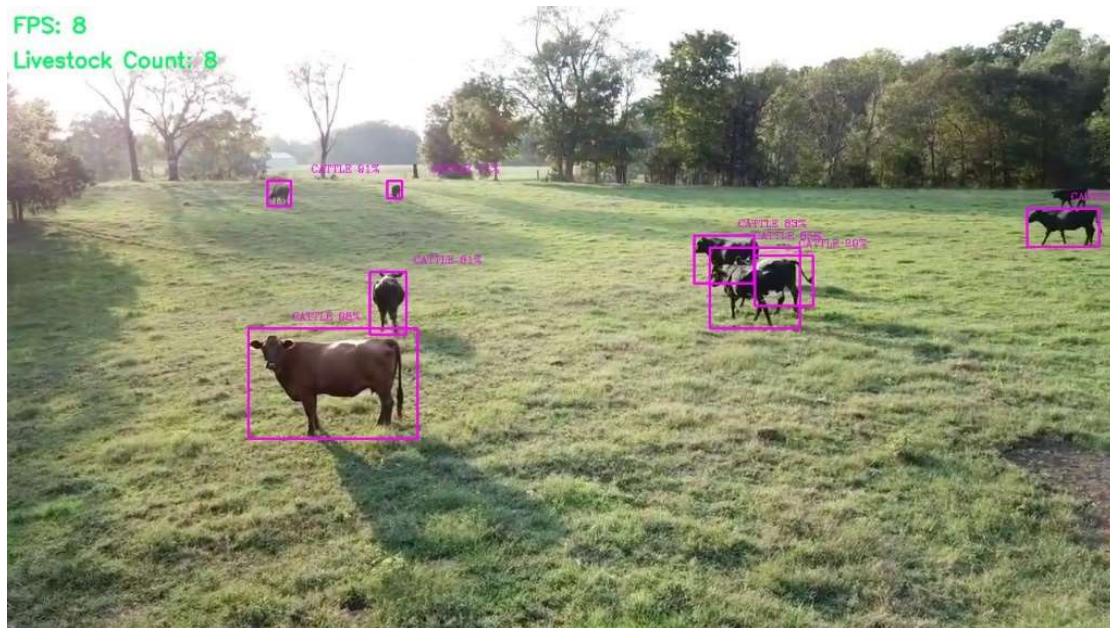
When applying the YOLOv4-tiny cattle detection model to an image of cows grazing in a field, it successfully detected every cow in frame giving relatively high confidence scores for the majority of the detections.

Figure 18: Result of YOLOv4-tiny Cattle on Test Image



The following screenshot is taken from the YOLOv4-tiny cattle model running on real-time drone footage of cows grazing in a field. In the top right, a cow has not been detected, this is likely because it is blending in with the trees in the background and the model cannot detect its features. The cows being overlapped by another in the foreground of the screenshot is given quite a low confidence score, 29%. The model does well with detecting cattle that are far away, as seen in the screenshot below, the cows in the background are detected and are given a relatively high confidence score. In this real-time demonstration, the fps ranged from 7 fps to 11 fps.

Figure 19: Result of YOLOv4-tiny Cattle in Real-Time



6.5. Model Comparison

Table 3 summarises the data gathered from training and testing different custom YOLO detection models.

Table 3: YOLO Results

Model	Mean Average Precision	Average Loss	Training Time	Real-Time FPS
YOLOv3	60.0%	0.8725	10h 05min	1-2
YOLOv3-tiny	41.7%	1.8704	01h 28min	8-11
YOLOv3-tiny (with large pre-trained model weights)	24.7%	2.6888	01h 34min	8-11
YOLOv4-tiny	56.7%	0.7906	01h 40min	7-11
YOLOv4-tiny (1 class: Cattle)	65.2%	0.9746	01h 24min	7-11

7. Conclusions & Future Work

As the agricultural world progresses, drone technology is becoming more prominent to help optimize operations. In this project, I explored how an object detection model applied to a drone-mounted camera could provide the foundations for a livestock management system using a UAV. As part of the project, the DJI Mavic Air drone was used to capture aerial footage of cattle grazing in a field. I explored different object detection algorithms and chose the YOLO family of algorithms as it was the best choice for real-time object detection. I trained different versions of this YOLO model on datasets of livestock images and found that the YOLOv4-tiny model produced the optimal results in terms of classification accuracy and fps. Applying this model to drone footage produced successful results as it could accurately count and detect livestock at various distances, at ~11 fps.

Future enhancements may be implemented to further optimize the detection and counting of livestock, specifically when they overlap or temporarily move out of frame. For example, the DeepSORT tracking algorithm could be used in future work for this purpose. Furthermore, research carried out by others in the field (Ref: [5]) suggests that potential valuable enhancements for overlapping animals could include the following techniques:

- Applying a tracking algorithm, giving each classification a unique identification value as they move across the field of view.
- Colour Manipulation of the field of view to increase contrast between the animal and background.
- Mathematical Morphology to isolate individual animals in tightly clustered groups.
- Image Matching to take into account image overlapping.

During this project I developed a more in depth understanding of Machine Learning concepts such as convolutional neural networks, classification, overfitting and underfitting, studied in the BCT course. Applying these machine learning concepts to a real-life application firmly solidified the concepts for me and engendered a fascination with this field of study as I look forward to my career in the world of computer science & IT.

8. Bibliography

- [1] Drones for Livestock Management, by Beth Jackson. Accessed Nov 2020. Available at: <https://coptrz.com/drones-for-livestock-management/>
- [2] Agricultural Drone, Wikipedia. Accessed Nov 2020. Available at: https://en.wikipedia.org/wiki/Agricultural_drone#:~:text=An%20agricultural%20drone%20is%20an,richer%20picture%20of%20their%20fields.&text=Thus%2C%20these%20views%20can%20assist%20i n%20assessing%20crop%20growth%20and%20production.
- [3] Dji Agras T20, by DJI. Accessed Dec 2020. Available at: <https://www.dji.com/ie/t20>
- [4] Drone Reforestation, by DroneSeed. Accessed April 2021. Available at: <https://www.droneSeed.com/>
- [5] Drones and dairy cows: Managing Livestock, by Priya Saini and Jason P. de Koff. Accessed March 2021. Available at: <https://www.dairyglobal.net/Smart-farming/Articles/2020/7/Drones-and-the-dairy-cows-Managing-livestock-618713E/>
- [6] A Beginner's Guide to CNNs, by Chris Nicholson. Accessed March 2021. Available at: <https://wiki.pathmind.com/convolutional-network>
- [7] LabelImg (2015), by Tzutalin. Accessed Jan 2021. Available at: <https://github.com/tzutalin/labelImg/>
- [8] OIv4_Toolkit, by TheAIGuysCode. Accessed March 2021. Available at: https://github.com/theAIGuysCode/OIv4_ToolKit
- [9] OpenCV Tutorial, by tutorialspoint. Accessed Feb 2021. Available at: <https://www.tutorialspoint.com/opencv/index.htm>
- [10] All you need to know about YOLOv3 (2021), by Afroz Chakure. Accessed Feb 2021. Available at: <https://dev.to/afrozchakure/all-you-need-to-know-about-yolo-v3-you-only-look-once-e4m#:~:text=The%20output%20of%20the%20model,is%20represented%20by%2085%20numbers.>
- [11] Canny Edge Detector, by Wikipedia. Accessed May 2021. Available at: https://en.wikipedia.org/wiki/Canny_edge_detector
- [12] yolov4-tiny project, by Techzizou. Accessed March 2021. Available at: https://github.com/techzizou/yolov4-tiny-custom_Training/tree/main/yolov4-tiny
- [13] Displaying real time fps in OpenCV Python, by Ayush Malik. Accessed April 2021. Available at: <https://www.geeksforgeeks.org/python-displaying-real-time-fps-at-which-webcam-video-file-is-processed-using-opencv/>
- [14] Saving a video using OpenCV, by @pankajpatra1998. Accessed April 2021. Available at: <https://www.geeksforgeeks.org/saving-a-video-using-opencv/>
- [15] Object Detection Yolo Course, by CV ZONE, fka Murtaza's Workshop. Accessed Feb 2021. Available at: <https://www.computervision.zone/courses/object-detection-yolo/>
- [16] Difference Between Test and Validation Datasets, by Jason Brownlee. Accessed Apr 2021. Available at: <https://machinelearningmastery.com/difference-test-validation-datasets/>

[17] Darknet Repository, by AlexeyAB. Accessed Feb 2021. Available at:

<https://github.com/AlexeyAB/darknet>

[18] Evaluating Models, by Google. Accessed May 2021. Available at:

<https://cloud.google.com/vision/automl/object-detection/docs/evaluate#:~:text=IoU%20threshold%20%3A%20Intersection%20over%20Union,the%20greater%20the%20IoU%20value.>

[19] Common loss functions in Machine Learning (2018), by Ravindra Parmar. Available at:

<https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23>

9. Appendix

9.1. Object Detection Code

Demo.py:

```
import cv2
import NumPy as np
import time

#cap = cv2.VideoCapture('rtmp://192.168.137.1/live/drone') # Live drone
#feed
#cap = cv2.VideoCapture(0) # Webcam feed
#cap = cv2.VideoCapture('Data/spacedCows.jpg') # jpg file
cap = cv2.VideoCapture('Data/grazing-cows.mp4') # .mp4 file# .mp4 file

frame_width = int(cap.get(3))
frame_height = int(cap.get(4))
size = (frame_width, frame_height)
result = cv2.VideoWriter('resultingVideo.avi',
cv2.VideoWriter_fourcc(*'MJPG'),10, size)

## FPS Variables
# used to record the time when we processed last frame
prev_frame_time = 0
# used to record the time at which we processed current frame
new_frame_time = 0

# Detection Variables
# defining width height of input image
whT = 416
# confidence threshold
confidence_threshold = 0.2
# non maximum suppression threshold
nms_threshold = 0.4

# Defining classes for detection
# file containing all classes the yolo model was trained on
classesFile = 'Object-names/obj.names'
# array to store all classes in classesFile
classNames = []
# opening the names file and extracting the classes
# pythons open() function and the 'rt' mode opens and reads a text file
```

```

with open(classesFile, 'rt') as f:
    # splitting text file by line and storing in classNames
    classNames = f.read().split('\n')

# yolo model configuration file path
yolo_config = 'YOLOv4-models/COWyolov4-tiny-custom.cfg'
# yolo model trained weights file path
yolo_weights = 'YOLOv4-models/COWyolov4-tiny-custom_best.weights'

# using opencv's deep neural network (dnn) module to create the darknet
network
net = cv2.dnn.readNetFromDarknet(yolo_config, yolo_weights)
# setting OpenCV as the backend
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
# using CPU as my pc does not have GPU
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

##function for finding objects in img
def findObjects(outputs, img):
    # getting height, width and channels of the image
    hT, wT, cT, = img.shape
    # whenever a good detection is found, the values will be stored in
    these lists
    # list to contain (bounding box) values for cx, cy, w, h
    bbox = []
    # list to contain all class id probabilities
    classIds = []
    # list to contain confidence values
    confs = []
    rectangles = []

    # we have 2 output layers for YOLOv4-tiny and 3 for YOLOv3
    for output in outputs:
        # for loop on detections in output (outputs contain the array of
        values for each detection)
        for det in output:
            # removing the first 5 (cx, cy, w, h, conf) elements in NumPy
            arrays to access the class id probabilities
            scores = det[5:]
            # define classID as the class with the largest probability that
            its being detected
            classId = np.argmax(scores)
            # confidence is the class with the max probability
            confidence = scores[classId]
            # filtering the classes
            # if confidence is greater than the threshold save the
            detection
            if confidence > confidence_threshold:
                #save the w and h values (3rd and 4th element in the NumPy
                arrays), multiply by actual w and h of the image to get (int)pixel values
                w,h = int(det[2]*wT) , int(det[3]*hT)
                # save the cx and cy values (1st and 2nd element in the
                NumPy arrays), divide w and h by 2 and subtract from cx,cy to get centre
                point of image, (int) pixel values
                cx,cy = int((det[0]*wT) - w/2), int((det[1]*hT) - h/2)
                # add the good detections to the bbox list
                bbox.append([cx,cy,w,h])
                # add the classID of the detection to the classIds list
                classIds.append(classId)
                # add the confidence scores to the confs list
                confs.append(float(confidence))

```

```

print(len(bbox))
## which of the bounding boxes we want to keep
## applying non-maximum suppression to the bounding boxes
## dnn.NMSBoxes take in the array of bounding boxes, the array of
confidence scores, the confidence threshold and the NMS threshold
indices =
cv2.dnn.NMSBoxes(bbox,confs,confidence_threshold,nms_threshold)

for i in indices:
    # take first element, i.e. remove square brackets
    i = i[0]
    box = bbox[i]
    cx,cy,w,h = box[0],box[1],box[2],box[3]
    # drawing rectangles for highlighting detections in the image
    rectangle = cv2.rectangle(img, (cx,cy), (cx+w,cy+h), (255,0,255),2)
    # storing rectangles drawn in an array for counting
    rectangles.append(rectangle)
    #num_of_rectangles = len(rectangles)
    # writing text: image window, output: class name and confidence
score *100, position (above box), font, font-scale, color, thickness
    cv2.putText(img,f'{classNames[classIds[i]].upper()}
{int(confs[i]*100)}%', (cx+50,cy-
10),cv2.FONT_HERSHEY_COMPLEX,0.4, (255,0,255),1)
    cv2.putText(img, 'Livestock Count: ' + str(len(rectangles)), (7, 70),
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (100, 255, 0), 2)

#while loop that gets the frames of our cam
while True:
    success, img = cap.read()
    if success == True:
        # Displaying Detections

        # the dnn only supports blob format so we convert the input image
to blob
        blob =
cv2.dnn.blobFromImage(img,1/255, (whT,whT), [0,0,0],1,crop=False)
        net.setInput(blob)

        ## getting names of all layers in the network
        layerNames = net.getLayerNames()
        #print(layerNames)
        ## printing the index of the yolo layers. getUnconnectedOutLayers
retrieves indexes of all layers with unconnected output
        # print(net.getUnconnectedOutLayers())
        outputNames = [layerNames[i[0]-1] for i in
net.getUnconnectedOutLayers()]
        # print the yolo layer names, note: yolo-tiny has only 2 yolo
layers, while normal yolo has 3
        #print(outputNames)
        #forward pass to compute net output
        outputs = net.forward(outputNames)
        # the output layers are lists of NumPy arrays (matrices)
        #print(outputs[0].shape) ## (507, 6) matrices of 507 rows and 6
columns, the 6 values is num of classes + 5.. the 5 extra elements are cx,
cy, w, h, conf
        #print(outputs[1].shape) ## (2028, 6) 507 & 2028 are the bounding
boxes
        #calling the findobjects function, inputting the output NumPy
arrays and video capture

```

```

        findObjects(outputs,img)

        # Calculating the fps

        # time the frame finished processing
        new_frame_time = time.time()
        # fps is number of frame processed in given time
        fps = 1 / (new_frame_time - prev_frame_time)
        prev_frame_time = new_frame_time
        # fps into integer
        fps = int(fps)
        # fps to string for putText()
        fps = str(fps)
        # configuring putText function: window, output, pos, font, font-
scale, color, thickness
        cv2.putText(img, 'FPS: '+fps, (7, 30), cv2.FONT_HERSHEY_SIMPLEX,
0.8, (100, 255, 0), 2)

        result.write(img)
        cv2.imshow('Drone Footage', img)
        #delay cam for 1 milisec, and e to exit
        if cv2.waitKey(1) & 0xFF == ord('e'):
            break

    else:
        break
cap.release()
result.release()

# Closes all the frames
cv2.destroyAllWindows()

print("The video was successfully saved")

```

9.2. Split Dataset Code

SplitDataset.py:

```

import glob, os

# Current directory
currentDir = os.path.dirname(os.path.abspath(__file__))

print(currentDir)

# change directory to folder containing entire dataset
currentDir = 'data/obj'

# Percentage of images to be used for validating model (test split)
testSplit = 10; # 10% test, 90% train

# using the open() functions write ('w') feature to create the train and
test files
trainFile = open('data/train.txt', 'w')
testFile = open('data/test.txt', 'w')

# Populating train.txt and test.txt files
counter = 1
# round the test split up to nearest full percentage

```

```

indexTest = round(100 / testSplit)
# glob used to retrieve filenames based on a specific pattern, (ending with
.jpg)
for pathAndFilename in glob.iglob(os.path.join(currentDir, "*.jpg")):
    # get the name of every jpg file in the folder (i.e. split the path by
the basename)
    title, ext = os.path.splitext(os.path.basename(pathAndFilename))

    # if counter in loop equals the test split percentage
    if counter == indexTest:
        counter = 1
        # write the jpg files to test.txt
        testFile.write("data/obj" + "/" + title + '.jpg' + "\n")
    else:
        #else write the remaining percentage to train.txt
        trainFile.write("data/obj" + "/" + title + '.jpg' + "\n")
        counter = counter + 1

```

9.3. YOLO Model Training Notebook

This part of the appendix contains the commands in each cell of the Google colab notebook used to train the YOLOv4-tiny model. For the other models, the same commands were used with the exception of some path files and getting the pre-trained configuration and weight files.

```

# Clone darknet repository into cloud
!git clone https://github.com/AlexeyAB/darknet

# mounting my drive
%cd ..
from google.colab import drive
drive.mount('/content/gdrive')

#creating symbolic link "/mydrive" for easier access
!ln -s /content/gdrive/My\ Drive/ /mydrive

# change daknet's makefile to enable dependencies for training with GPU, Op
enCV and CUDA
%cd content/darknet/
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
!sed -i 's/LIBSO=0/LIBSO=1/' Makefile

# building darknet and all it's dependencies
!make

# cleaning darknet's data folder before repopulating
%cd data/
!find -maxdepth 1 -type f -exec rm -rf {} \;
%cd ..
# cleaning darknet's config folder before repopulating
%rm -rf cfg/
%mkdir cfg

```

```

# copy zipped image dataset to cloud
!cp /mydrive/Cattle_YOLOv4-tiny/obj.zip ../

# unzip dataset to darknet's data folder
!unzip ../obj.zip -d data/

# copy custom yolo class names and data files from drive to cloud
!cp /mydrive/Cattle_YOLOv4-tiny/obj.names ./data
!cp /mydrive/Cattle_YOLOv4-tiny/obj.data ./data

# copy SplitDataset.py to cloud
!cp /mydrive/Cattle_YOLOv4-tiny/SplitDataset.py ./

# run the SplitDataset.py to generate and populate train-validation files
!python SplitDataset.py

# get pre trained convolutional layer darknet weights from github repository
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-tiny.conv.29

# train object detection model with obj.data, custom yolo config and pretrained weights
!./darknet detector train data/obj.data cfg/yolov4-tiny-custom.cfg yolov4-tiny.conv.29 -dont_show -map

# defining a helper function for darknet's chart.png
def imShow(path):

    import cv2
    import matplotlib.pyplot as plt
    %matplotlib inline
    image = cv2.imread(path)
    height, width = image.shape[:2]
    resized_image = cv2.resize(image, (3*width, 3*height), interpolation = cv2.INTER_CUBIC)
    fig = plt.gcf()
    fig.set_size_inches(18, 10)
    plt.axis("off")
    plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
    plt.show()

# plot graph of training loss
imShow('chart.png')

```