1 Enhanced Project Description

1.1 Problems

In the context of identifying intruder aircraft, from the point of view of an opposing aircraft, object detection must be a robust and consistent system. The problem of over-fitting (Section 2.1.2) causes Machine Learning systems to perform worse on their test data [1] - therefore performing worse in real-world scenarios. Since Object Detection is inherently a machine-learning problem, reducing over-fitting is of great importance to ensure these systems can be trusted in this safety-critical environment.

1.2 Goals

This project aims to investigate the efficacy of Data Augmentation to reduce the problem of over-fitting, thus increasing the accuracy and precision of Object Detection systems on their test data. It aims to test several known data augmentation methods on the AVOIDDS dataset [2] to create the best data augmentation process for this problem - by combining the best-performing methods to different extents. This project will also investigate what is the most optimal percentage of a dataset to be augmented data for detecting intruder aircraft.

1.3 Scope

This project will investigate Data Augmentation only; it will not use any other methods to reduce over-fitting (other than cross-validation to ensure fair test results). It will use an open-source object detection system with pre-trained models [3] as well as a pre-existing dataset [2].

2 Literature Review

2.1 Machine Learning for Object Detection

2.1.1 Supervised Machine Learning

Supervised Machine learning involves training a model to learn from labelled training data [4], the model can then predict outcomes to new unlabeled test data. Supervised learning methods usually guarantee good performance [5], therefore, the data set for this project must have matching labels for each image.

2.1.2 Over-fitting and Under-fitting

Over-fitting occurs when a used model is too complex, it may perfectly fit the random noise of the data while attempting to catch the regularities [5]. A model over-fitting the training data will perform well on the training data but not on the test data [1]. Under-fitting occurs when a given model is too simple to capture all regularities in the signal component [5], however it will cause fewer problems in this project, as trusted pre-trained trained models will be used (Figure 15).

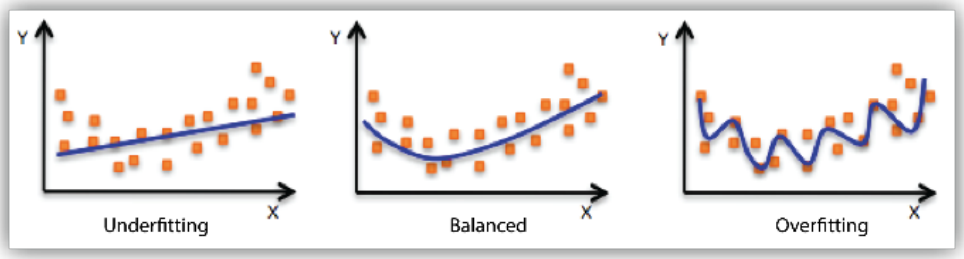

Figure 1: Types of model fitting. [1]

There are many ways to prevent over-fitting [6]. These include:

- **Training with more data**.
- **Data Augmentation** (Section 3.3)
- **Adding noise to the input data**.
- **Feature selection**.
- **Cross-Validation** (Section 3.1.4)
- **Simplifying Data**
- **Regularization**
- **Ensemble Learning**

As stated in Section 2, this project will investigate the efficacy of Data Augmentation to increase the accuracy of Object Detection algorithms, including its ability to decrease over-fitting.

2.1.3 Training and Testing Machine Learning Algorithms

It is an important rule in machine learning not to use the same dataset for model training and model evaluation. If you want to build a reliable machine learning model, the dataset must be split into training, test and validation sets. [7]

The training set is the set of data used to train the model and learn the patterns in the data and should have a diversified set of inputs so that the model is trained in all scenarios. The validation set is separate from the training set and is used to validate our model performance during training. The test set is the set of data used to test the model after training is complete. [7]

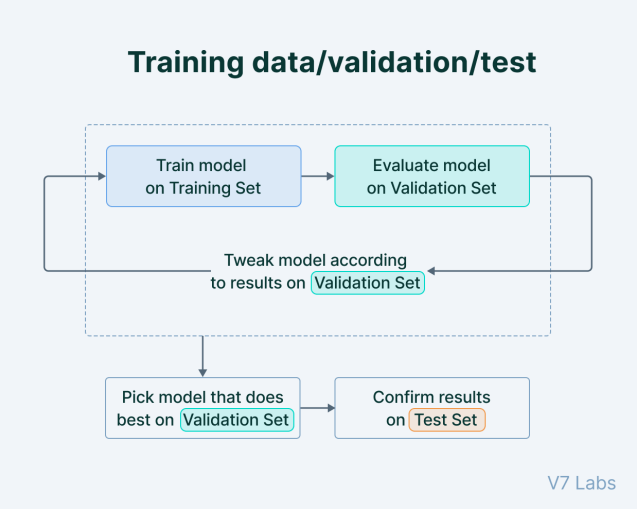

Figure 2: Training data/validation/test. [7]

There is no optimal split percentage for deciding these splits. However, there are two major concerns while deciding on the optimum split:

- If there is less training data, the model will show high variance in training.
- With test/validation data, your model evaluation/model performance statistic will have greater variance.

Despite this, there is a rough industry standard for splitting data sets shown in Figure 3.

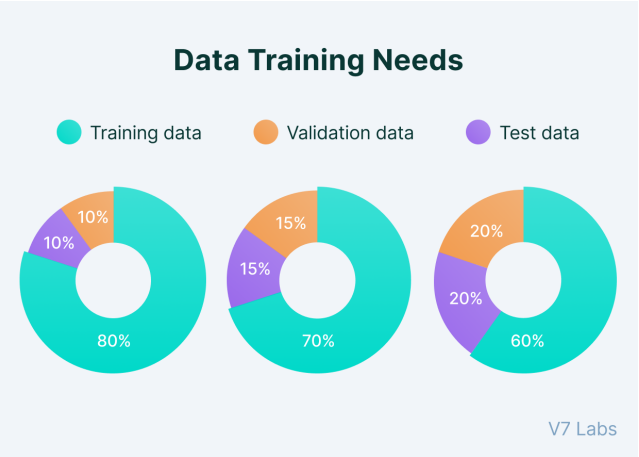

Figure 3: Standard Data Percentage Splits [7]

The process of splitting the dataset into these separate parts is also important to ensure a robust and fair way of testing machine learning models [7]. These advanced techniques for data splitting include:

- **Randomise**. However, a major drawback arises when training with class-imbalanced data sets (i.e., datasets with different numbers of samples per dataset category). Using random allocation could create a bias in this scenario.
- **Stratified**. This solution alleviates the class imbalance problem by preserving the distribution in the train, test and validation sets [7].
- **Cross-Validation** or **K-Fold Cross Validation** is a more robust technique for data splitting, where a model is trained and evaluated 'k' times on different samples [7]. The data is split into K subsets of data (known as folds) and a Machine Learning Model is trained on all but one (k-1) and the remaining fold is used for testing [1]. This process is repeated K times with a different subset reserved for evaluation (and excluded from training each time) [1]. This process is demonstrated in Figure 4.
- **Stratified K-Fold Cross-Validation**. K-Fold Cross-Validation can suffer from the same problem as random sampling as data distribution may get skewed when the folds are created [7]. Stratified K-Fold Cross-Validation avoids these inconsistencies by maintaining the class ratio of the data while generating the folds.

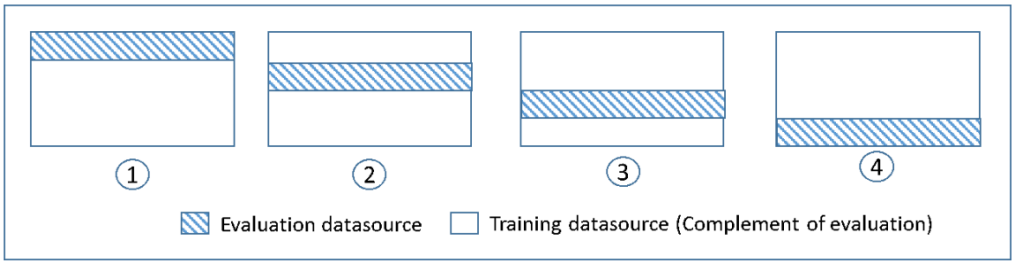

Figure 4: Cross-Validation Process. [1]

2.1.4 Convolution Neural Networks

One type of Neural Network architecture is the Convolution Neural Network (CNN). CNNs are similar to traditional feed-forward networks, however, they are designed to work with grid-structure inputs (i.e., a 2D image) [8]. The convolution operation is a dot-product operation between a grid-structured set of weights (kernel) and grid-structured inputs. This operation is useful for data with a high level of spatial or other locality such as image data [8].

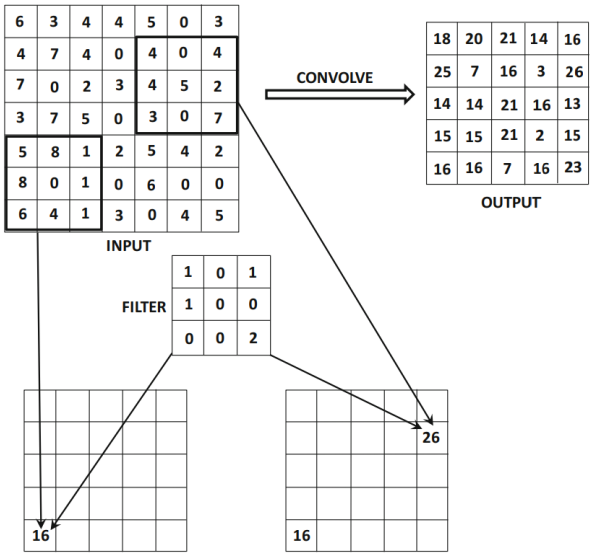

Figure 5: Image Convolution. [8]

2.1.5 Object Detection

Object detection combines image classification (as Stated in 3.1.1) and localization to determine what objects are in the image or video and specify where they are in the image [9].

Object Detection algorithms are broadly classified into two categories: Single-Shot (One-Stage) and Two-Stage detectors (Figure 7). Single-shot detection uses a single pass of the input image to make predictions about the presence and location of objects. This makes them very efficient but generally less accurate

Figure 6: Left - Classification. Right - Object Detection [9]

than other methods. Two-shot detection uses two passes of the input image. The first pass generates a set of potential object locations, and the second is used to refine these proposals and make final predictions. Due to this, two-shot algorithms are generally more accurate [10].

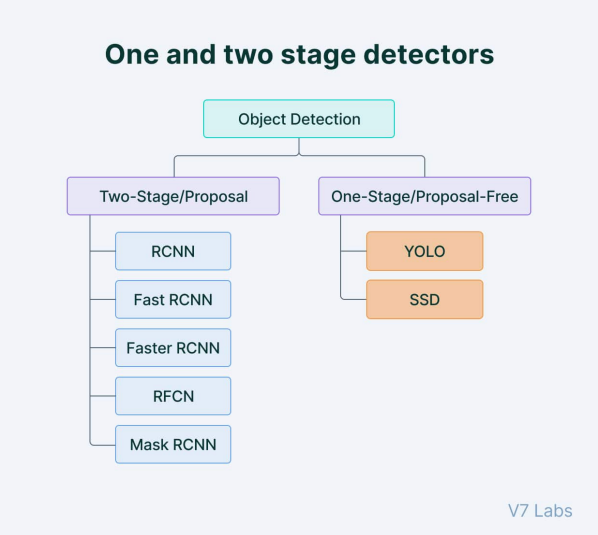

Figure 7: Object Detection Categories. [10]

The two most common evaluation metrics, to compare the predictive performance of object detection models, are Intersection over Union (IoU) and Average Precision (AP). IoU is a popular metric to measure localization accuracy and calculate localization errors. To calculate the IoU between the predicted and ground truth bounding boxes, you divide the intersecting area of the two boxes by their total union area (Figure 8) [10].

AP is calculated using a precision vs recall curve for a set of predictions. Recall is the proportion of actual positive class samples that we identified by the model (i.e., if the test set of a dataset consists of 100 samples in its positive class and 60 were identified, the recall is 60%). Whereas, precision is the proportion of true positives to the total predictions made by the model. The area under the precision vs recall curve gives the AP per class. The average over all classes is

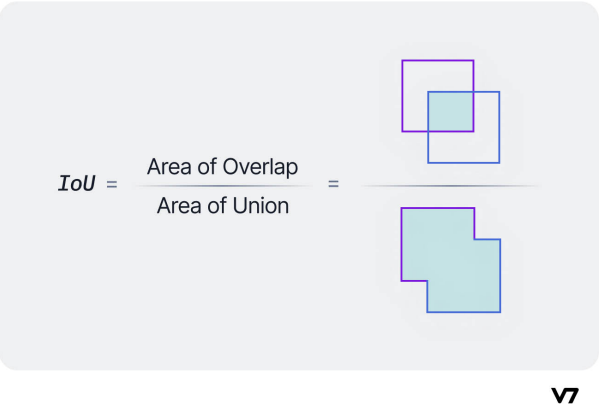

Figure 8: Intersection over Union. [10]

called the mean Average Precision (mAP) [10].

For object detection, precision and recall are used for measuring the decision performance (not for predictions). Instead, an IoU value > 0.5 is considered a positive prediction and IoU < 0.5 is negative [10].

2.2 You Only Look Once (YOLO)

This section researches the chosen state-of-the-art object detection model for this project, named You Only Look Once (YOLO).

2.2.1 What is YOLO?

YOLO is an open-source object detection system. It uses a single CNN (Section 3.1.5) to simultaneously predict multiple bounding boxes and class probabilities for those boxes, as the name states, 'You Only Look Once' at an image [3]. This makes YOLO a one-stage object detection system (Section 3.1.6) as opposed to previous two-stage systems such as the Region-Based CNN (R-CNN), which requires additional refinements, duplicate elimination and re-scoring of boxes based on other objects in the scene [3].

Due to its simplicity, YOLO is extremely fast [3]. The base network runs at 45 frames/images per second (fps) when training on PASCAL VOC 2007 [11] with no batch processing. This was using a Titan X GPU, with YOLO fast versions running at more than 150 fps [3] (Figure 9).

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [31]	2007	16.0	100
30Hz DPM [31]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [38]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[28]	2007+2012	73.2	7
Faster R-CNN ZF [28]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Figure 9: Object Detection Systems Speed Comparison. [3]

Error analysis of the VOC 2007 run demonstrates YOLO's accuracy in comparison to R-CNN. For each category at test time, the top N predictions for that category are looked at. The results of this test are shown in Figure 10 with the correctness of predictions being classed as the following [3]:

- **Correct:** correct class and IOU > 0.5
- **Localization:** correct class, $0.1 >$ IOU > 0.5
- **Similar:** class is similar, IOU > 0.1
- **Other:** class is wrong, IOU > 0.1
- **Background:** IOU > 0.1 for any object

Fast R-CNN is narrowly better at correctly predicting objects, however, it makes far more background errors (due to R-CNN using a sliding window and region proposal-based techniques). YOLO struggles to localise objects correctly, with over twice as many localisation errors [3].

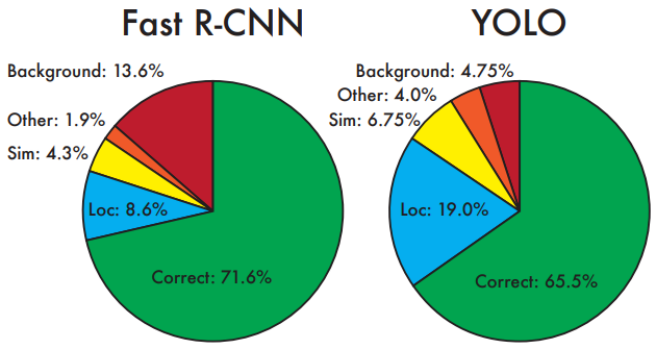

Figure 10: Error Analysis: Fast R-CNN vs YOLO. [3]

2.2.2 Different versions of YOLO

There have been 8 iterations of YOLO since its initial release in 2016, each iteration incrementally improves the speed and accuracy of the system, often by improving the CNN architecture [10]. Version 2 also introduced anchor boxes; these are predefined bounding boxes of different aspect ratios and scales which help to determine final bounding boxes when combined with predicted offsets [10]. YOLOv7 uses nine anchor boxes allowing it to detect a wider range of object shapes and sizes, thus reducing the number of false positives [10]. YOLOv7 also increased the resolution that it processes images at to 608 by 608 pixels, helping it to detect smaller objects and increase overall accuracy [10].

YOLOv8 is currently the latest version of YOLO; it was created by Ultralytics. The code for YOLOv8 is open source and licensed under a GPL license [12].

2.3 Limitations of YOLO

Some limitations of YOLO are important to consider:

- YOLO struggles with small objects and objects that appear in groups [3] (the latter is unlikely to affect this project as each image contains one aeroplane [2]).
- YOLO struggles to generalize to objects in new or unusual aspect ratios or configurations [3].
- YOLO can be sensitive to changes in lighting or other environmental conditions [10].
- YOLO (like all object detection) is computationally intensive, especially on resource-strained devices [10].

2.3.1 How to use YOLO

YOLO is written in Python and can be accessed via installing the Ultralytics package (YOLOv8 [12]) or by cloning the corresponding git repository (YOLOv7 [13] and YOLOv8 [14]). YOLO can be implemented using both the command line interface (CLI) for quick and easy training as well as the Python interface

so that YOLO can be implemented into larger projects [15].

Custom datasets must be formatted correctly to work with YOLO:

- The dataset must be arranged in the correct folder structure (Figure 11) [15].

Figure 11: Dataset Folder Structure. [15]

- Labels must be in the form of a single ***.txt** file per image (if there is no object in the image, no file is required). The text file should be formatted with one row per object in class x_{centre} y_{centre} *width height* format. Box coordinates must be in normalized **xywh** format (from 0 to 1). An example label text file for a specific image is shown in Figure 12 [15].

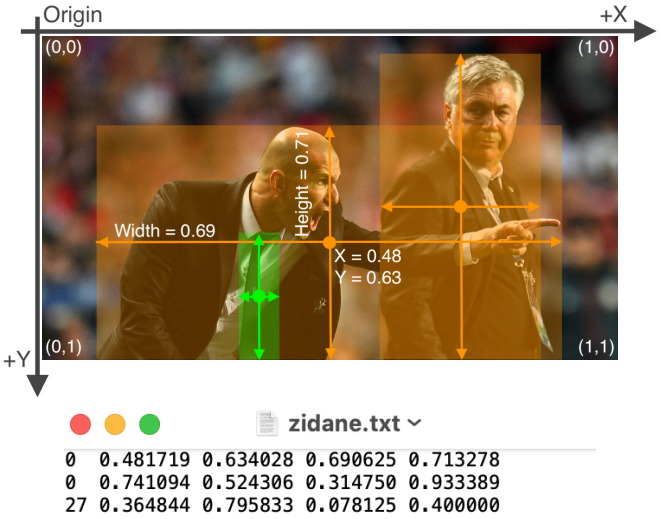

Figure 12: Example label file for image. [15]

- There must be a **data.yaml** file in the root directory of the dataset that describes the dataset (shown in Figure 13) [15].

- Images of a dataset can be optimized to reduce the size of the dataset for more efficient processing. The tools required for this are part of the Ultralytics Python package [15].

```
path: ../datasets/starter_dataset
train: images/train
val: images/valid
names:
  0: aircraft
```

Figure 13: Example .yaml file for AVOIDDS dataset. [2]

Figure 14 shows example Python code for training using YOLOv8.

```
from ultralytics import YOLO

# Create a new YOLO model from scratch
model = YOLO('yolov8n.yaml')

# Load a pretrained YOLO model (recommended for training)
model = YOLO('yolov8n.pt')

# Train model using 'coco128.yaml' dataset for 3 epochs
results = model.train(data='coco128.yaml', epochs=3)

# Evaluate the model's performance on the validation set
results = model.val()

# Perform object detection on an image using the model
results = model('https://ultralytics.com/images/bus.jpg')

# Export the model to ONNX format
success = model.export(format='onnx')
```

Figure 14: Example Python Code for training with YOLO. [15]

A pre-trained YOLO model is used to train the on the **coco128** dataset [16]; the model is trained on the training data of the dataset using 3 epochs. The epoch count defines the number of times a training dataset is to be worked through the learning algorithm [17]. The model (weights) used in this code is the **yolov8n.pt** file however there are many pre-trained (default) models available in YOLO (Figure 15). The **YOLOv8n** model is the fastest (80.4ms with for CPU ONNX test) as well as the least accurate (mAP of 37.3). Whereas, **YOLOv8x** is the most accurate (53.9 mAP) as well as the slowest (479.1ms with for CPU ONNX test).

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figure 15: Different Models in YOLOv8. [14]

2.4 Data Augmentation

Data Augmentation is the process of extracting more information from an original dataset - via augmentations - to reduce overfitting. This augmented data is added back to the training set to create a more comprehensive set of possible data points, minimizing the distance between the training and validation set [18]. Krizhevsky used data augmentation in their experiments to increase the dataset size by a magnitude of 2048 [19] by horizontally flipping images and changing the intensity of RGB colour channels using PCA colour augmentation. These augmentation methods reduced the model's error rate by over 1% [18].

2.4.1 Different Data Augmentation Methods

The most popular practice for data augmentation is to combine affine image transformations, such as rotation, reflection, scaling and shearing (Figure 16), with colour modification. Colour modification includes geometric distortions or deformations such as histogram equalisation, contrast or brightness enhancement, white-balancing, sharpening and blurring (Figure 17). These methods are proven to be good methods for increasing the training dataset [20].

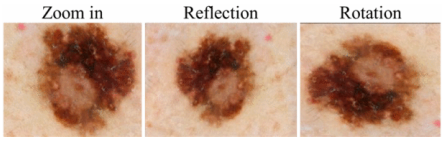

Figure 16: Affine Transformations. [20]

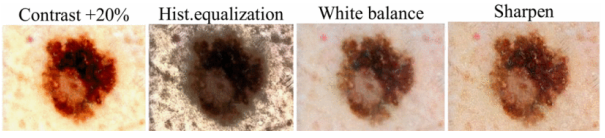

Figure 17: Colour Modifications. [20]

When investigating data augmentation for a deep learning-based model in pathological lung segmentation, M. S. Alam, D. Wang, and A. Sowmya suggest that standard affine transformations provide limited performance improvement within Deep Neural Network (DNN) architecture [21]. Instead, they generated new images using Gaussian filters and employing different levels of contrast and brightness to synthesise images with high opacity and low contrast. They obtained the best performance when using 250% augmented images (from 506 to 1,265). Similarly, Y. Mi, S. Tabirca, and A O'Reilly augmented their data by adding Gaussian noise, creating subtly varied instances to add to the dataset. They found this to mitigate the risk of overfitting in their models [22].

YOLO uses an image processing method called mosaic [15] (which can be seen in Figure 27) where multiple images are batched together while training. This could be considered data augmentation, however, since this will be present in every test and is part of the inner workings of YOLO's CNN, it is not going to be considered a data augmentation method in this project.

2.4.2 Data Augmentation in Python

Metamorphic Relations [23] is a Python package which can be used to perform simple affine transformations. The static functions *flip_horizontal_transform(x)*, *flip_vertical_transform(x)* and *rotate_transform(x, angle)* can be used to easily perform the corresponding transformation using numpy arrays. The package also has the *blur_transform(x, sigma)* to perform a blur colour modification on an image using a Gaussian filter [23].

OpenCV [24] is an open-source computer vision and machine learning software library, it has several optimised algorithms that aid in performing the geometric distortions and colour modifications for this project:

- OpenCV contains the function *cv.equalizeHist()* to perform **traditional histogram equalisation**, however, this can cause contrast issues in the background or foreground. **Adaptive histogram equalisation** can be performed using **Contrast Limited Adaptive Histogram Equalization (CLAHE)** with the OpenCV function *cv.createCLAHE* [25].

- The point processes of multiplication and addition with a constant can be used to control the **contrast and brightness** of images.

$$g(x) = \alpha f(x) + \beta$$

  Adjusting the gain(α) and bias(β) will control the contrast and brightness respectively [26].

- **Sharpening** an image can be performed using image convolution with a specific 2D kernel (Section 3.1.4) [27], figure 18 shows various sharpening kernels. The *cv.filter2D()* function in OpenCV can be used to perform convolution efficiently with a given kernel [28].

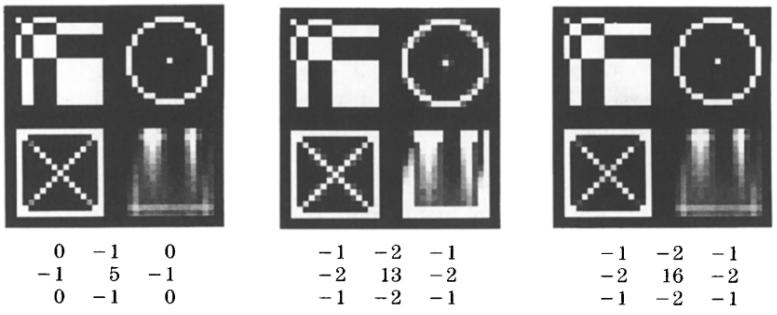

Figure 18: Kernels for Image Sharpening. [27]

- **White Balancing** can be performed in OpenCV using the Grayworld Algorithm [29].

- **Gaussian Noise** can be generated simply using NumPy [30]. Using the *np.random.normal()* function.

2.5 AVOIDDS Data set

The AVOIDDS dataset is a collection of 72,000 images and labels from the ownship's point of view of encounters with intruder aircraft in the airspace [2]. X-Plane 11 is used for data generation to allow for programmatic control of the environmental conditions and intruder locations. The images are distributed equally among 6 weather types, 3 aircraft types and 4 regions (Figure 19).

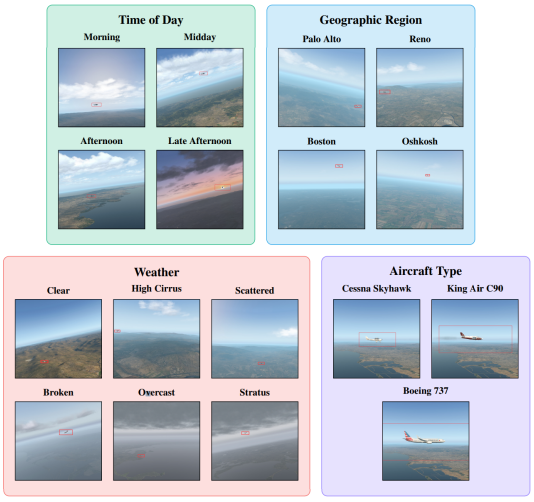

Figure 19: Variation in AVOIDDS images. [2]

A baseline YOLOv8 model was trained on the AVOIDDS dataset for 100 epochs. The training took 73 hours using an NVIDIA Geforce GTX 1070ti [2]. The model achieved an mAP of 0.866 overall with a precision of 0.990 across all categories, the results are shown in Figure 20. The largest difference in performance arose as the aircraft moved further away from the point of view, as well as this, the model has a higher mAP for the larger aircraft (Boeing 737-800, KingAir C90).

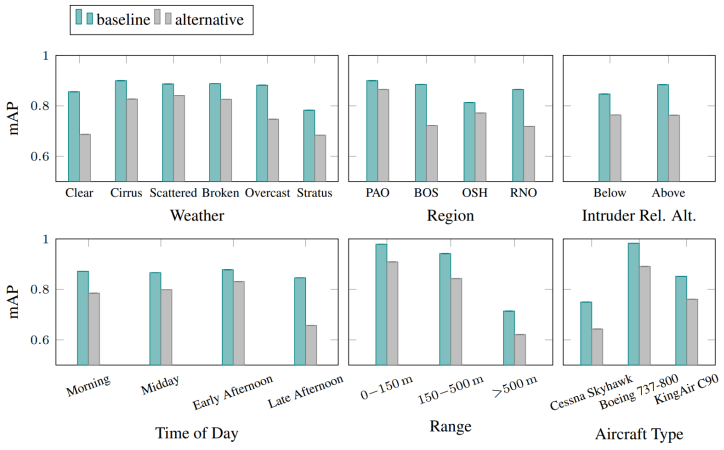

Figure 20: Results of YOLOv8 baseline model on AVOIDDS. [2]

3 Risk Analysis

This section analyses each risk with a probability & severity rating and how we plan to mitigate each one.

ID	Risk	P (1-5)	S (1-5)	RE (P × S)
R1	Training time of YOLOv8 is high	5	4	20*
R2	YOLOv8 over-performing on dataset	2	3	6
R3	Data Augmentation does not improve performance	3	2	6
R4	Model over-fits training data	3	3	9
R5	Bias arising from unwanted sources	3	4	12
R6	Pressure from other modules	4	2	8

Table 1: Risk Analysis. P - Probability, S - Severity, RE - Risk Exposure *Largest Risk Exposure

R1: Train on smaller training sets from the dataset for initial tests. For the final tests, make use of remote high-performance servers [31] (expanded upon in Section 4). Can also train for fewer epochs if training time is still too long.

R2: If YOLOv8 performs too well on initial tests, leaving less room for improvement, then using an older version of YOLO would limit initial performance. Once again, training for fewer epochs can achieve this.

R3: Important to document and make conclusions on all results to find possible underlying reasons for this.

R4: Use stratified cross-validation where possible to limit overfitting and to ensure each test is fair.

R5: Make sure variables such as training/test set sizes and category distributions, number of epochs, the YOLO model used and the number of folds in cross-validation are all controlled between tests and documented.

R5: Organise the schedule (Section 6) so that it allows time for other module commitments when necessary.

4 Design

This section outlines the initial proposed design for this project.

4.1 Test Strategy Design

The flowchart below (Figure 21) outlines the design for the test strategy.

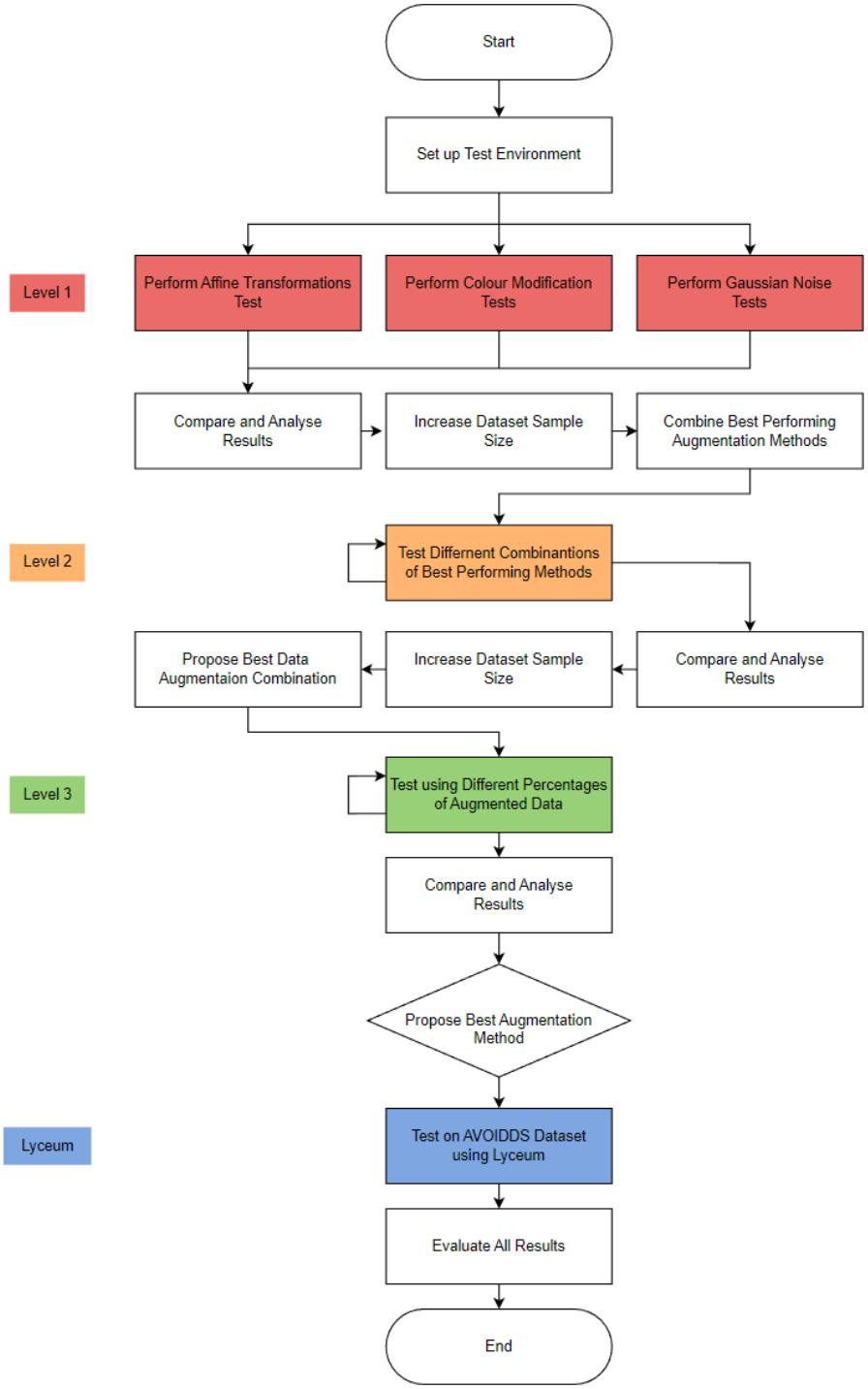

Figure 21: Flowchart for Test Strategy.

The Lyceum Teaching Cluster is a service that allows for substantially greater computational power or memory than is available on individual PCs [31]. This will allow for training on the entire AVOIDDS dataset. To apply, the application form must be filled out by the project supervisor.

The flowchart below outlines the design for an individual test which will be used at each test level in Figure 21.

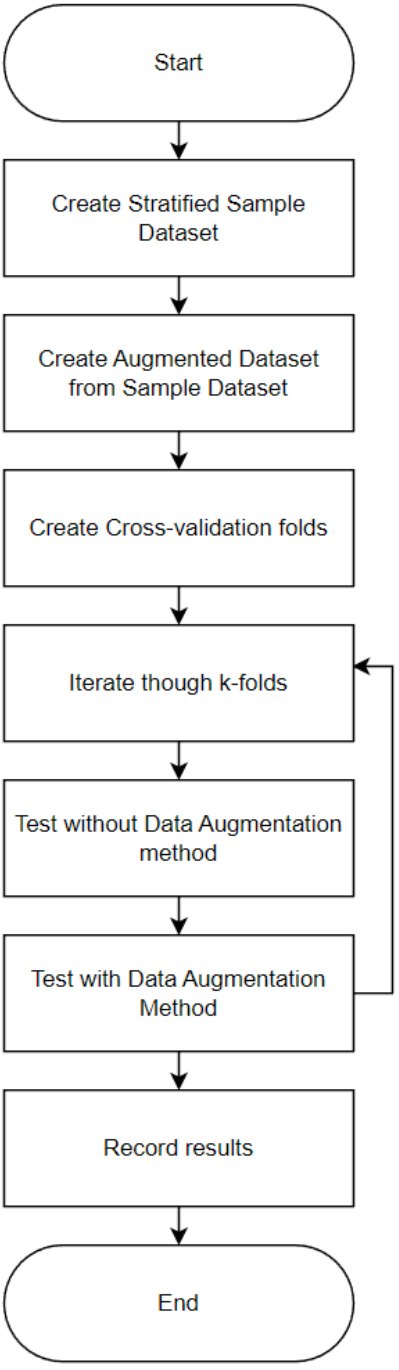

Figure 22: Flowchart for Individual Test.

As stated in Section 2.1.3, the sampled datasets must be stratified, maintaining the class distribution from the whole dataset. Section 2.1.3 also stated the importance of cross-validation for fair testing, therefore, it has been included as part of the test design.

4.2 System Design

Below (Figure 23) is the class diagram for the proposed system design to implement the test strategy.

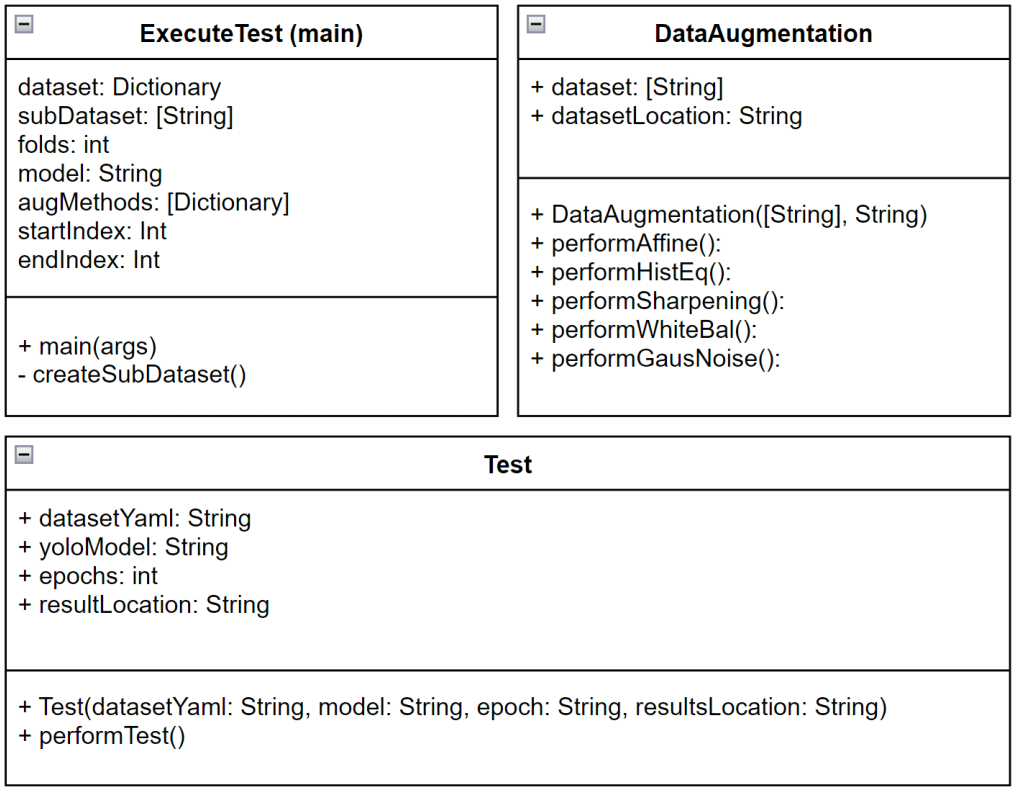

Figure 23: Class Diagram for executeTest.

The design currently contains three classes. ExecuteTest is the main class, this class will take a YAML configuration file as a parameter which contains all the information required to perform a test; the dataset, number of cross-validation folds, which augmentation methods are being used and their parameters, the size of the sub-dataset and which YOLO model is being used. As of writing this report, the YOLOv8n model will be used for training (Section 2.3.1), this is because it is the fastest model and reducing the time taken to perform a test is more important than optimising the mAP (allowing for a larger potential improvement).

The metadata.json file will represent the AVOIDDS dataset as a Python dictionary, which can be used to create a smaller sub-dataset (using a stratified approach to keep class distribution, Section 2.1.3). The Data Augmentation class will then be used to perform the required augmentation methods defined in the augMethods dictionary (Figure 24), creating an augmented version of the

sample dataset. This dataset will then be split into k folds (defined in the configuration file). ExecuteTest will then create instances of the Test class to carry out train/test runs using YOLOv8 saving the results of each test to a results directory. It will iteratively do this (with then without data augmentation) k times, performing cross-validation.

augMethods Dictionary

```
{
 "method": "affine",
 "params": {
  "imagePct": 20,
 }
},
{
 "method": "heqEq",
 "params": {
  "imagePct": 20,
 }
}
```

Figure 24: Example augMethods Dictionary.

5 Technical Progress and Investigation

To familiarise myself with YOLO and its performance, I initially used YOLOv7 to train individual images from AVOIDDS to see how an untrained model would perform. As can be seen in Figure 25, when an aircraft is close to the the point of view of the camera, YOLOv7 performs well with a mAP of 0.93. Whereas, in Figure 26, when it is further away (centre-right), it is not detected whatsoever.

Figure 25: Successful Detection on Individual Image.

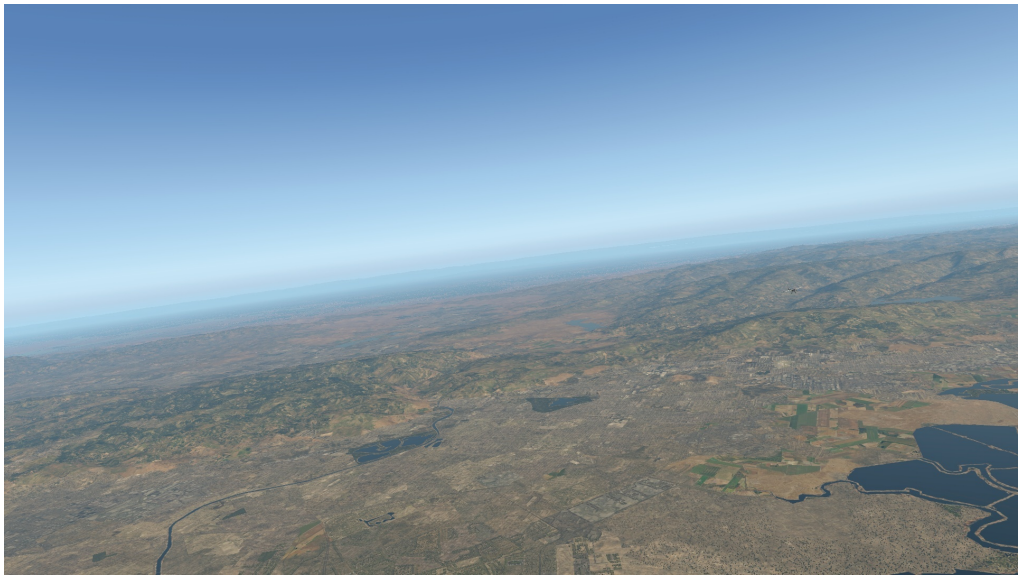

Figure 26: Unsuccessful Detection on Individual Image.

I then installed YOLOv8 with the goal of training on a small sample of the AVOIDDS dataset. I did this by forking the Ultralytics GitHub repository [15] and adding it as a sub-module in the repository for this project. Before any

training could take place, PyTorch [32] had to be reinstalled to allow for CUDA usage (GPU utilisation).

Before training against AVOIDDS, I first trained YOLOv8 against the **coco128** [16] dataset for 2 epochs. Figure 27 shows an example train batch of trained images from the run and Figure 28 shows the precision-recall curve (showing that a mAP of 0.613 was achieved over all classes). Figure 29 shows several graphs and plots generated from this run which can be used to evaluate its performance.

Figure 27: Train Batch 1 from Coco128 run.

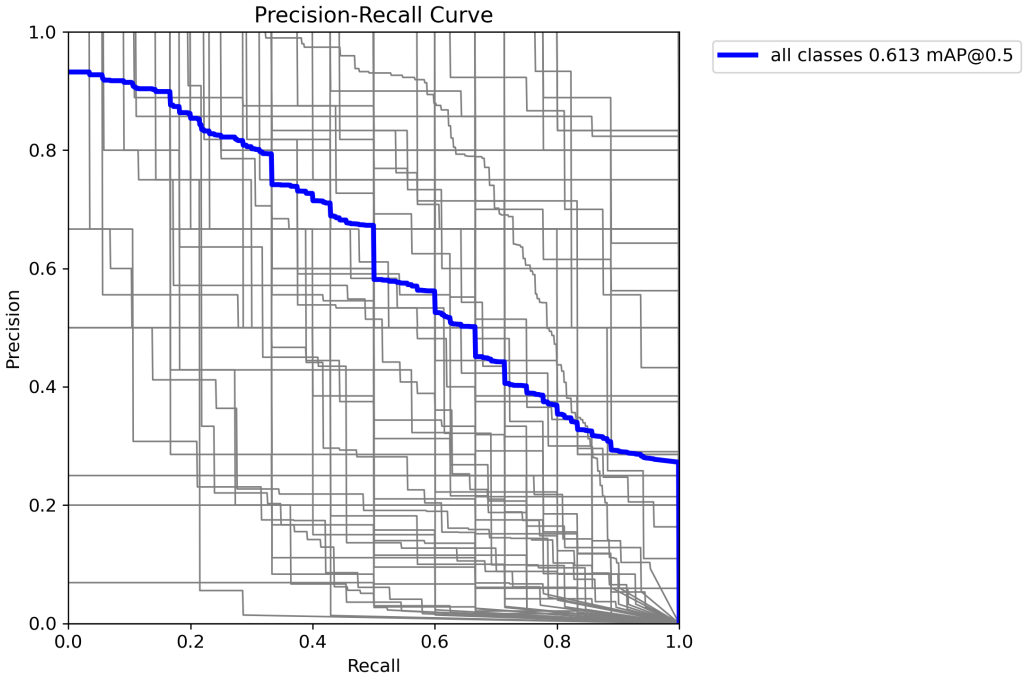

Figure 28: Precision-Recall Curve from Coco128 run.

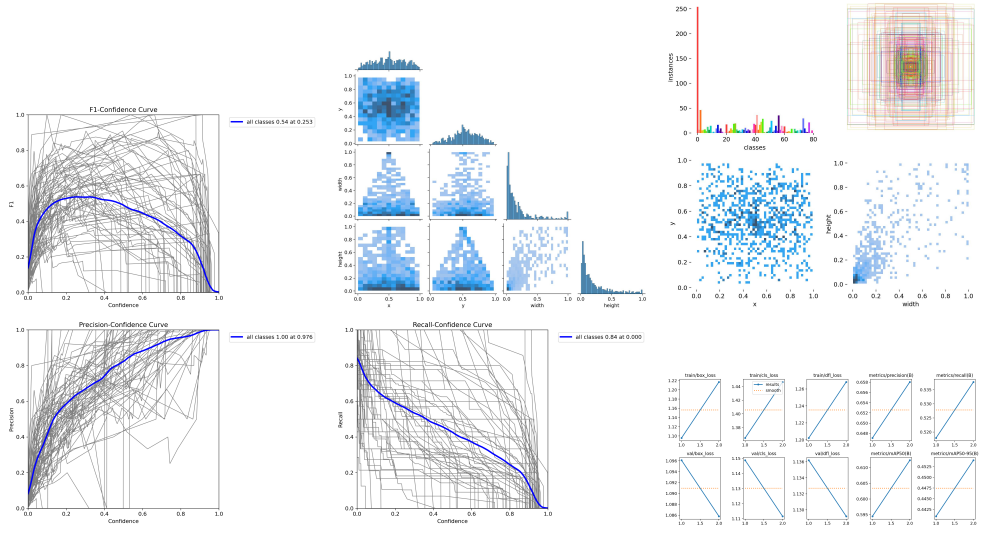

Figure 29: Plots generated from coco128 run.

After familiarising myself with how to train YOLOv8 against a dataset, I created numerous samples of the AVOIDDS dataset and trained them for different epochs. The results (Table 2) give a good representation of the time it takes for YOLO to train on an individual PC, however, the mAP values may not be representative of a real test as the datasets are not stratified.

Dataset Size	Epochs	Run Time (hours)	mAP50	mAP50-95
100	4	0.024	0.002	0.001
100	50	0.302	0.680	0.249
1000	5	0.315	0.689	0.277
1000	10	0.651	0.654	0.317

Table 2: Successful Runs of AVOIDDS Sample Datasets. All Datasets used an 80/20 train split.

When the trained model predicts a validation image (Figure 30), the differences between the predicted and actual bounding boxes can be seen. Also, the model sometimes mistakenly identifies the background landscape as an aircraft.

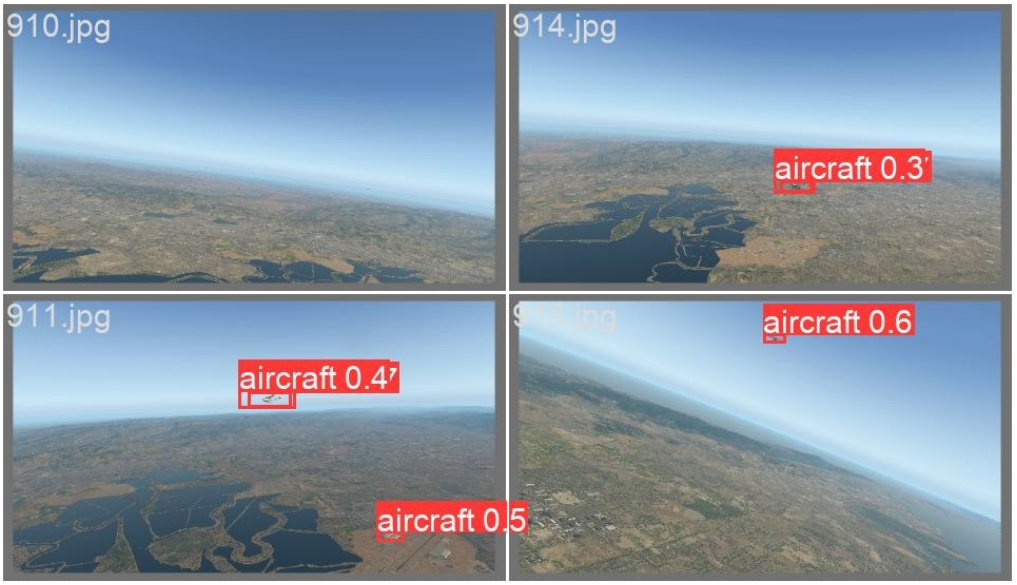

Figure 30: YOLO Prediction on Validation Image.

6 Project Management

6.1 Supervisor Contact

There has been constant communication with this project's supervisor via Microsoft Teams as well as face-to-face meetings. We chose Microsoft Teams firstly due to its professional text-based chat which made quick contact easier - as compared to using email. Secondly, Microsoft Team's video chat allows for online meetings and screen sharing for progress meetings.

6.2 Account of work

Below (Figure 31) is the Gantt chart for completed work as of the handin date for this report. Some tasks such as reviewing literature and system design overran, this is partly due to a slight under-estimation of the size of the task, as well as, other commitments such as coursework for other modules taking longer than first expected.

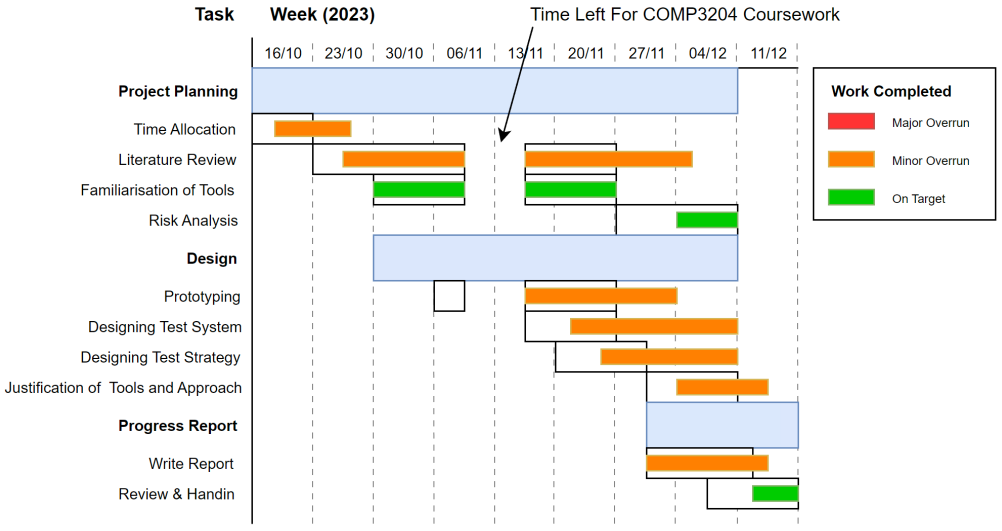

Figure 31: Gantt chart for completed work in Semester 1.

6.3 Plan of remaining work

Below (Figure 32) is the Gantt chart showing the plan for the remaining work of this project. I have allocated the most time to implementing the system design, followed by equal time to run each test phase, before evaluating and writing the final report.

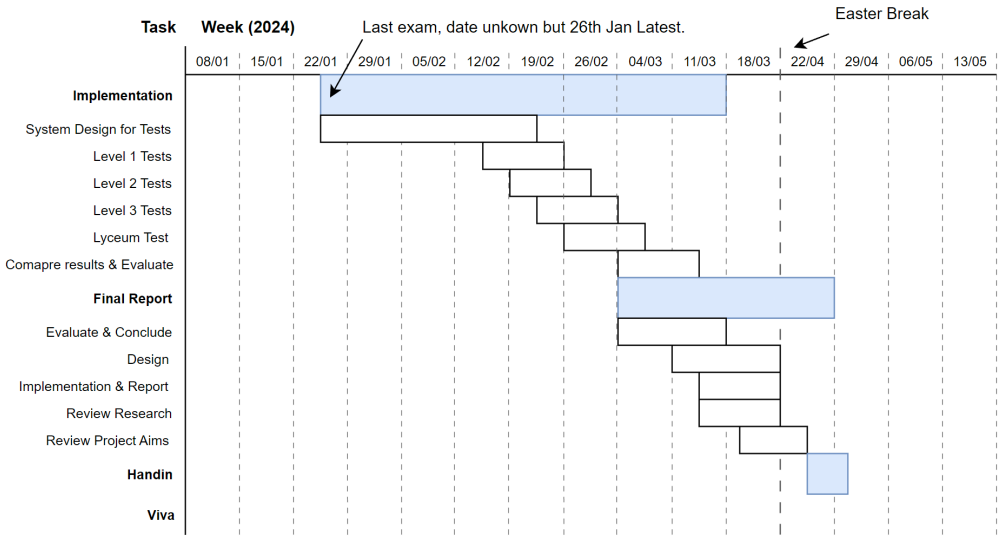

Figure 32: Gantt chart for planned work in Semester 2.