

Advanced Computer Vision Report

ARI3129: Advanced Computer Vision for Artificial Intelligence

Kian Parnis

kian.parnis.20@um.edu.mt

Evangeline Azzopardi

evangeline.azzopardi.19@um.edu.mt

Mario Vella

mario.vella.20@um.edu.mt

B.Sc. (Hons) Artificial Intelligence
University of Malta

Contents

1	Google Drive layout	3
1.1	CVPart2	3
1.2	RetinaNet_2	3
2	Task 1	4
2.1	Methodology	4
2.2	Evaluation	6
2.2.1	Metrics	6
2.2.2	Values	8
3	Task 2	10
3.1	Introduction	10
3.2	Methodology	10
3.2.1	Dataset Creation	10
3.2.2	Object Detector #1 - YOLOv7	10
3.2.3	Object Detector #2 - RetinaNet	11
3.3	Evaluation	11
3.3.1	Object Detector #1 - YOLOv7	11
3.3.2	Object Detector #2 - RetinaNet	13
3.3.3	Comparison of Both Models	14
4	Task 3	14
4.1	Task 3 Instructions of Use	14
4.2	Task 3 Limitations - YOLOv7	16

1 Google Drive layout

To store relevant files, two google drives were created, *CVPart2* and *RetinaNet_2*. The following is the layout of these two google drives:

1.1 CVPart2

- YoloV7_TrainingWeights/ - File containing weights for each trained version.
- Task 1 Dataset/ - The images used in Task 1.
- Task 2 Dataset YoloV7/ - The images and annotations used in Task 2 (Yolov7).
- Task 3 classes and resnet50 h5 file/ - File containing weights and CSV to run task3's py file.

1.2 RetinaNet_2

- Command_Line/ - Contains .txt files with the command line commands to:
 1. Execute Creating, Converting and Evaluating the models according to version.
 2. Execute RetinaNet_Version7.py
- Evaluations/ - Contains .txt files with the evaluation metrics of the models according to version.
- Images/ - Contains the training data used according to version.
- JSON_Data/ - Contains the testing JSON data for Version 7, created through executing RetinaNet_Version7.ipynb
- keras-retinanet/ - Contains the repository for the RetinaNet with all required to create a custom object detector. Obtained from: <https://github.com/fizyr/keras-retinanet>
- Tensorboard/ - Contains the tensorboard output from the training phases according to version.
- Test_Part3/ - Contains the testing images and JSON data for Version 7, created through executing RetinaNet_Version7.ipynb .
- resnet_coco_best_v21.0.h5 - A pre-trained model for Image Recognition and Object Recognition tasks in ImageAI. Obtained from: <https://github.com/OlafenwaMoses/ImageAI/releases/tag/1.0>
- RetinaNet_Version7.ipynb - A IPython Notebook containing the code to utilize the trained model developed.
- RetinaNet_Version7.pfd - The PDF version of the IPython Notebook.
- RetinaNet_Version7.py - The Python script required for Task 3.
- Roboflow.ipynb - The IPython Notebook used to download the training data for Images/
- TowerCrane.JPEG - Testing Image
- TowerCrane.json - JSON file output form executing RetinaNet_Version7.py. Obtained by executing command in Command_Line/RetinaNet_Version7_Command.txt
- TowerCrane_Predictions.JPEG - Output image from executing RetinaNet_Version7.py. Obtained by executing the command in Command_Line/RetinaNet_Version7_Command.txt
- TowerCrane2.JPEG - Testing Image

- TowerCrane2.json - JSON file output from executing RetinaNet_Version7.py. Obtained by executing variation of command in Command_Line/RetinaNet_Version7_Command.txt
- TowerCrane2_Predictions.JPEG - Output image from executing RetinaNet_Version7.py. Obtained by executing variation of command in Command_Line/RetinaNet_Version7_Command.txt

2 Task 1

2.1 Methodology

For Task 1 of this assignment, we were tasked with making use of 2 pre-trained object detection architectures in order to classify images as ones that include a crane and ones which do not. After this is done, the task was to evaluate both of the architectures and compare their performance to one another. For this task, we decided to make use of the RNN50 and VGG16 architectures.

For this task, a small sample size of 20 images from the ImageNet dataset was used in order to test our architectures. Each image was set to a size of 224x224. Using these measurements, the architectures were given the 'imagenet' weights from TensorFlow and then given the 20 images together with their sizes (224x224) in order to give the predictions of whether a crane is there or not.

In the image set that was used, there were no images that did not have cranes in them, and therefore, all images were expected to have a return that showed a crane prediction.

Once the predictions were done, the output was done as follows from top to bottom:

1. The name of the image is outputted
2. The image is shown
3. The top 5 predictions of this image are given together with the value i.e. crane 0.8689

In the figures below, the output for the same image using both architectures is shown:

n03126707_16122.JPEG



Image n03126707_16122.JPEG's predicted top five:

- 1) crane 0.99135244
- 2) dragonfly 0.0031807597
- 3) radio_telescope 0.0017605656
- 4) pole 0.0007299983
- 5) hook 0.000607116

Figure 1: Output for RNN50 Object Detector

n03126707_16122.JPEG



Image n03126707_16122.JPEG's predicted top five:

- 1) crane 0.944151
- 2) pole 0.01428185
- 3) hook 0.004412151
- 4) tripod 0.0041267164
- 5) flagpole 0.0034948483

Figure 2: Output for VGG16 Object Detector

As can be seen in the figures above, the architectures worked successfully as the prediction for a crane was the highest value from the 5 with a significant value.

2.2 Evaluation

2.2.1 Metrics

The metrics used to compare the performance of each model were evaluated with the use of the model's accuracy score and confusion matrices. Accuracy is the most common evaluation metric used to measure the performance of a machine learning model. It represents the proportion of correct predictions made by the model. In other words, it is the number of times the model correctly predicted the outcome divided by the total number of predictions. For example, if a model correctly predicts 80 out of 100 instances, the accuracy of the model is 80%. The following is the formula for accuracy:

$$Accuracy = \frac{(Number\ of\ Correct\ Predictions)}{(Total\ Number\ of\ Predictions)}$$

A confusion matrix is a table that is used to define the performance of a classification algorithm. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa). The following is the table used for our binary classification problem:

		Predicted Labels		
		p	n	total
Actual Labels	p'	True Positive (TP)	False Negative (FN)	P'
	n'	False Positive (FP)	True Negative (TN)	N'
total		P	N	

- *True Positive (TP)*: These are the cases in which the model correctly predicted the positive class.
- *True Negative (TN)*: These are the cases in which the model correctly predicted the negative class.
- *False Positive (FP)*: These are the cases in which the model predicted the positive class, but the actual class was negative. This is also known as a "Type I Error".
- *False Negative (FN)*: These are the cases in which the model predicted the negative class, but the actual class was positive. This is also known as a "Type II Error".

Using these values, we can calculate other important evaluation metrics such as precision, recall, specificity, and F1-score.

- *Precision*: The proportion of true positives among all positive predictions.

$$Precision = \frac{TP}{(TP + FP)}$$

- *Recall*: The proportion of true positives among all actual positive instances.

$$Recall = \frac{TP}{(TP + FN)}$$

- *Specificity*: The proportion of true negatives among all actual negatives instances.

$$Specificity = \frac{TN}{(TN + FP)}$$

- *F1-score*: The harmonic mean of precision and recall.

$$F1-score = 2 * \frac{Precision * Recall}{(Precision + Recall)}$$

2.2.2 Values

To evaluate both architectures, as mentioned above, we will be using the confusion matrices and the evaluation metrics for Precision, Recall, and F1-score. For both architectures, as mentioned in the methodology above, there were no images that had 0 cranes inside of them. Therefore, in the confusion matrices below, the column which represents all non-crane images, the value will always be 0. The value for specificity is not being calculated because since we are not using any non-crane images, all the values needed for True Negatives and False Negatives will be 0.

Below, the confusion matrices for the RNN50 and the VGG16 architectures were drawn showing that the RNN50 was more successful in detecting the probability of a crane in the images as it only had 1 FN compared to the VGG16's 4FN. This means that the RNN50 only found 1 image which it concluded that it did not have the probability of having a crane inside it.

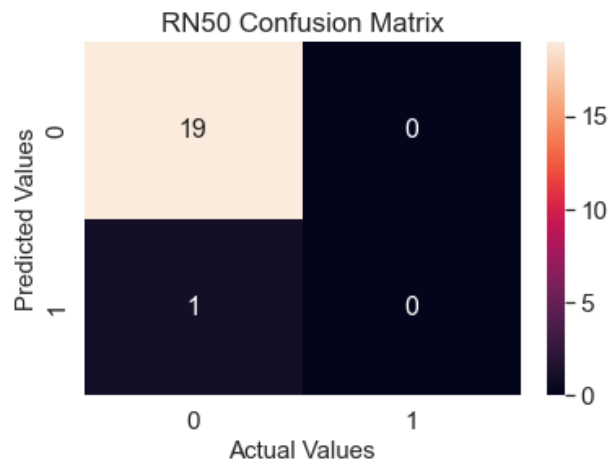


Figure 3: Confusion Matrix for RNN50 Object Detector

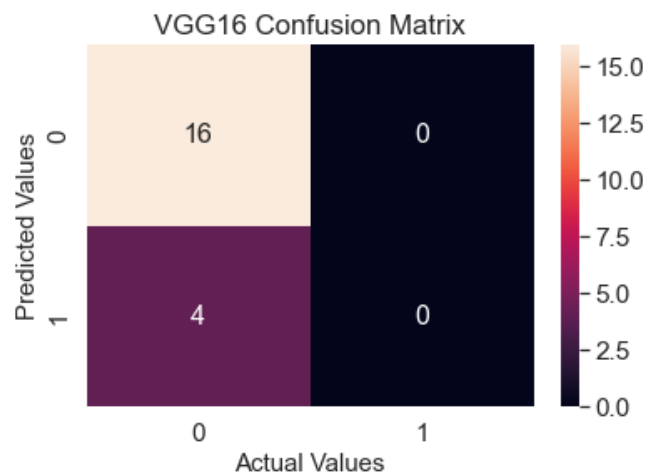


Figure 4: Confusion Matrix for VGG16 Object Detector

After reading the confusion matrices and noting down their values, the metrics below were calculated for both architectures. Firstly, one can see the evaluation metrics for the RNN50 object detector:

$$Accuracy = \frac{19 + 0}{(19 + 1 + 0)} = 0.95$$

$$Precision = \frac{19}{(19 + 0)} = 1$$

$$Recall = \frac{19}{(19 + 1)} = 0.95$$

$$F1-score = 2 * \frac{1 * 0.95}{(1 + 0.95)} = 0.97$$

Secondly, below one can see the calculation of the precision, recall and F1-score of the VGG16 detector:

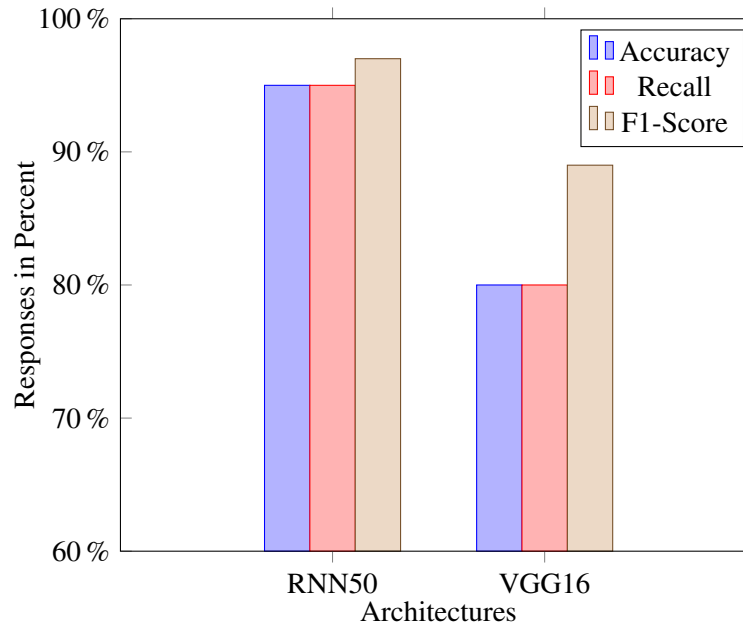
$$Accuracy = \frac{16 + 0}{(16 + 4 + 0)} = 0.8$$

$$Precision = \frac{16}{(16 + 0)} = 1$$

$$Recall = \frac{16}{(16 + 4)} = 0.8$$

$$F1-score = 2 * \frac{1 * 0.8}{(1 + 0.8)} = 0.89$$

Table 1: Comparison of Object Detector Architectures



As can be seen from the metrics above and the bar graph for both architectures, the RNN50 object detector architecture performed better than the VGG16. This was already certain from the confusion matrices drawn above but this has been confirmed with the calculation of the F1-score which shows that the RNN50 is the architecture that outperforms the other.

3 Task 2

3.1 Introduction

In this task, we were tasked with training two object detectors using our own prepared dataset and evaluating them against one another. For this task, we decided to make use of the YOLOv7 Object Detector, and the RetinaNet detector. We decided to make use of these 2 specific models because we wanted to make a comparison between YOLOv7 which is the current state-of-the-art object detector and RetinaNet which is an older object detector released in 2015.

3.2 Methodology

3.2.1 Dataset Creation

When creating our dataset, we started by using the ImageNet dataset which included 1300 images. This dataset was then annotated using "makesense.ai" and then put into roboflow which resized all of the images in order for them to be the same size. 4 Tests were done using this image dataset using 4 different versions of RoboFlow. The below table shows how the images were split. Above this table however we will be explaining the different augmentations done on the images:

- Test 1: Resize all Images to 640x640
- Test 2: Resize all Images to 640x640
- Test 3: Resize Images to 640x640, Crop, Shear, and Mosaic applied to the images
- Test 4: Resize Images to 640x640, Crop, Shear, and Mosaic applied to the images (Added Maltese Horizon Images)

Version	Training	Validation	Testing	Total
2 (Test 1)	556	70	70	696
7 (Test 2)	795	103	102	1000
9 (Test 3)	1400	143	137	1720
11 (Test 4)	1400	199	199	1848

Table 2: Table including all the image numbers from all the versions

3.2.2 Object Detector #1 - YOLOv7

When training this object Detector, we used a GitHub repository that included different python programs needed for the usage of the YOLOv7 object detector. The train python file was run for the 4 different versions of the image dataset as shown in the section above.

3.2.3 Object Detector #2 - RetinaNet

When working with the RetinaNet object detector, the GitHub repository which included the detector we used also included three separate python files which needed to be used in order to train our model with the dataset prepared and mentioned above. In the figure below, one can see that for each type of action i.e. training, validation, and testing, a separate command is being written and used in order to prepare the model well for its predictions. The evaluations extracted when running these python files can be seen in the evaluation section below.

```
Version #2:

Train Model:
python keras-retinanet/keras_retinanet/bin/train.py --freeze-backbone --random-transform --weights resnet50_coco_best_v2.1.0.h5 --epochs 40 --steps 50 --batch-size 8 --
snapshot-path keras-retinanet/snapshots/version2_2 --tensorboard-dir Tensorboard/ csv Images/Cv2-Part2-2/train/_annotations.csv Images/Cv2-Part2-2/train/classes.csv

Convert Model:
python keras-retinanet/keras_retinanet/bin/convert_model.py --backbone resnet50 keras-retinanet/snapshots/version2_2/resnet50_csv_40.h5 keras-
retinanet/snapshots/version2_2/resnet50_csv_40_converted.h5

Evaluate Model:
python keras-retinanet/keras_retinanet/bin/evaluate.py csv Images/Cv2-Part2-2/valid/_annotations.csv Images/Cv2-Part2-2/valid/classes.csv keras-
retinanet/snapshots/version2_2/resnet50_csv_40_converted.h5
```

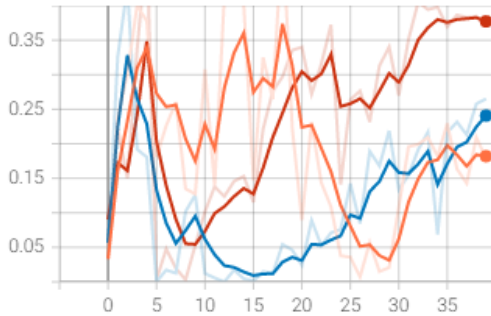
Figure 5: Confusion Matrix for RNN50 Object Detector

3.3 Evaluation

3.3.1 Object Detector #1 - YOLOv7

When evaluating the YOLOv7 object detector, TensorBoard was used in order to plot graphs that show the mean precision, precision, and recall of each of the dataset versions created from Roboflow. In the below figures, one can see the graphs mentioned that are plotted against each other. Due to the fact that TensorBoard did not allow the plotting of more than 3 curves at the same time, in the figures, there will be one graph with the curves for Versions 2, 7, and 9 while the 2nd graph will contain the curve for Version 11.

metrics/mAP_0.5
tag: metrics/mAP_0.5



(a) Mean Average Precision for first 3 Versions

metrics/mAP_0.5
tag: metrics/mAP_0.5

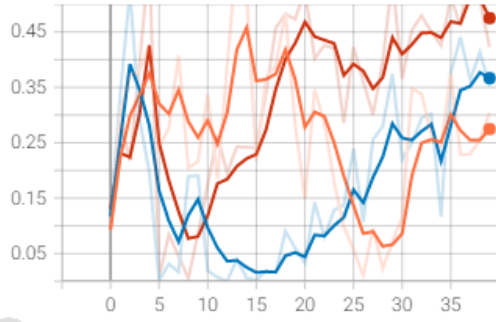


(b) Mean Average Precision for Version 11

Figure 6: Mean Average Precision for All 4 Versions

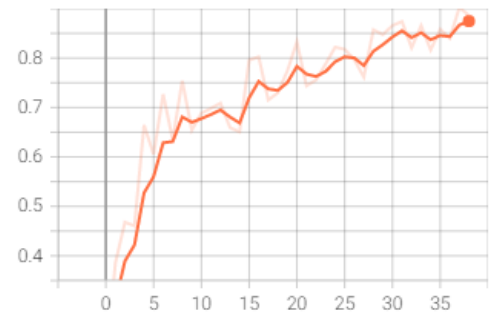
From these curves, one can see that Version 11 has the best mean average precision. This means that as an average value, when training, the Version 11 dataset on the YOLOv7 detector had a much larger ratio of True Positives to the total number of Positives.

metrics/precision
tag: metrics/precision



(a) Precision for first 3 Versions

metrics/precision
tag: metrics/precision

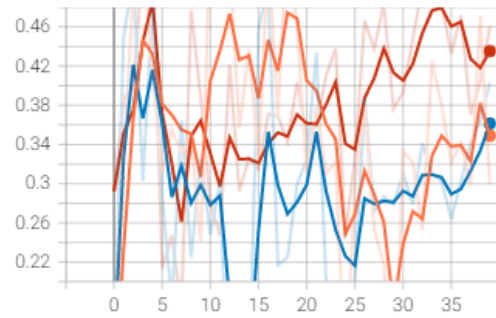


(b) Precision for Version 11

Figure 7: Precision for All 4 Versions

From these curves, one can see that Version 11 has the best precision of around approx 0.9. This was already expected due to the fact that the graphs above were working with the metric of the mean average precision.

metrics/recall
tag: metrics/recall



(a) Recall for first 3 Versions

metrics/recall
tag: metrics/recall



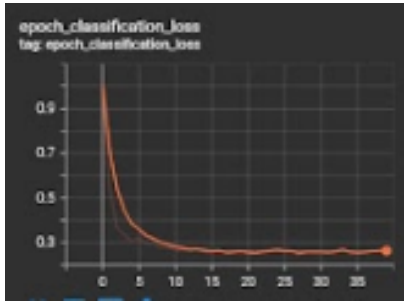
(b) Recall for Version 11

Figure 8: Recall for All 4 Versions

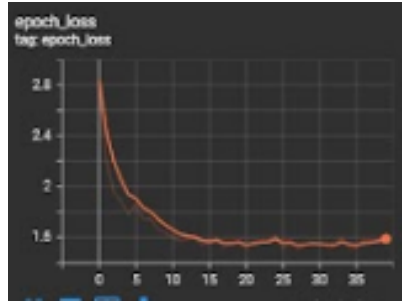
Finally, using these last 2 graphs showing the curves of the recall values, one can see that Version 11 again has the best recall curve of all of them. Furthermore, this can conclude the fact that Version 11 is the best dataset that trained the YOLOv7 model as when comparing both precision and recall values, it had the best curves.

3.3.2 Object Detector #2 - RetinaNet

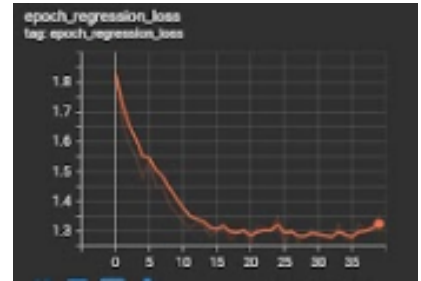
The 4 figures below include the Tensorboard graphs for the losses inside this object detector. When working with this detector it was noted that the metrics saved when training were the loss metrics such as Classification Loss, Loss by Epoch, and Regression Loss.



(a) Version 2: Classification Loss

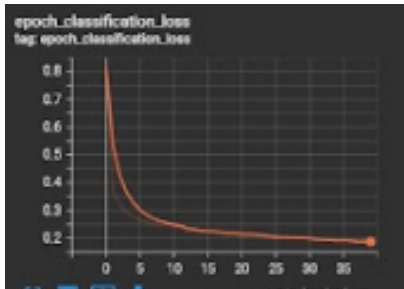


(b) Version 2: Loss

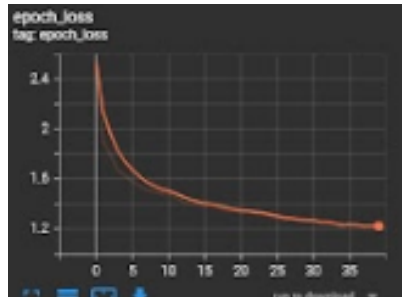


(c) Version 2: Regression Loss

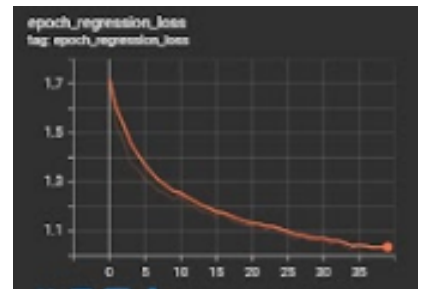
Figure 9: Losses for Version 2



(a) Version 7: Classification Loss

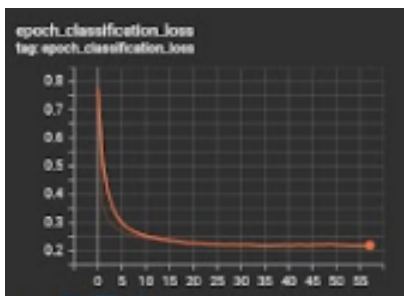


(b) Version 7: Loss

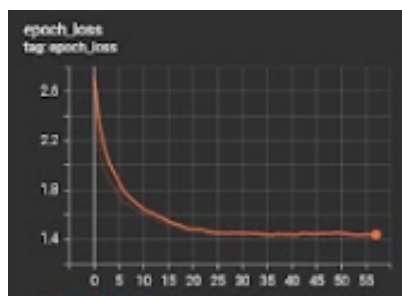


(c) Version 7: Regression Loss

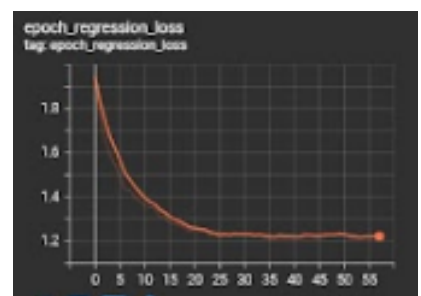
Figure 10: Losses for Version 7



(a) Version 9: Classification Loss

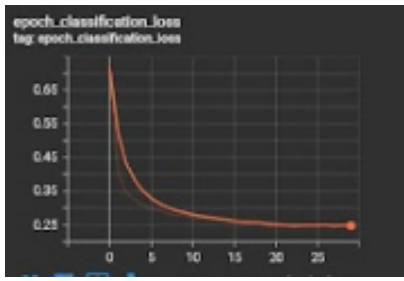


(b) Version 9: Loss

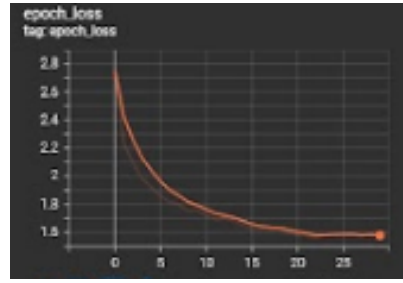


(c) Version 9: Regression Loss

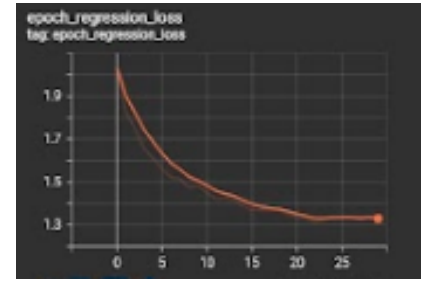
Figure 11: Losses for Version 9



(a) Version 11: Classification Loss



(b) Version 11: Loss



(c) Version 11: Regression Loss

Figure 12: Losses for Version 11

After looking through all the graphs of all these versions, it was seen that the best model to work with and to move onto Task 3 with was Version 7 as it had the best accuracy and had the least loss when training.

3.3.3 Comparison of Both Models

Model and Version	Precision	mAP_0.5
YOLOv7 Version 2	0.52	0.5
YOLOv7 Version 7	0.59	0.42
YOLOv7 Version 9	0.58	0.58
YOLOv7 Version 11	0.89	0.89
RetinaNet Version 2	0.69	0.69
RetinaNet Version 7	0.78	0.78
RetinaNet Version 9	0.55	0.55
RetinaNet Version 11	0.54	0.54

Table 3: Table Precision and mAP_0.5 for YOLOv7 and RetinaNet

As shown in Table 3, the best-performing detector out of the two was as expected, YOLOv7. An interesting observation is that RetinaNet achieved its best score when trained using no augmentation, whilst YOLOv7 achieved its best results when all the images that were used came from ImageNet, the Maltese data-set, and augmentation.

4 Task 3

4.1 Task 3 Instructions of Use

In this section, we were tasked with adapting one of our previous models into a python file that could take an image as input, detect a crane, set bounding boxes, save the information of the boxes into a JSON file, and give good console feedback to the user. For this task, we decided to work with the RetinaNet object detector.

Below, one can see 2 examples of the code at work as a Jupyter notebook in order to see the output clearer. For this output, the first figure is showing when a crane is detected and so, the bounding box is shown in the image, and the coordinates of the box are outputted to the user. Furthermore, a JSON file is made with the bounding box

information and the path is given to the user. If no predictions have a value larger than the threshold value set by the user, then no bounding boxes will be set and therefore, a JSON file will not be created.



```
A JSON file WILL be generated!

Path Passed:  JSON_Data\version_7
Modified Path :  JSON_Data\version_7\{}.json
Final Path:  JSON_Data\version_7\n03126707_12489_JPEG.rf.994a0ec0264bbd105270b6
75e139e14.jpg.json
Image:  n03126707_12663_JPEG.rf.4c2701b6b95f0b6b5b8b22424a2daac4.jpg
Data Format:  '[' Predicted bounding box, Score, Object Label ']'

Predicted Bounding Box Data:  [array([144.05429,  0.      , 485.55655, 639.2
], dtype=float32), 0.8861829, 0]
```

(a) Output when a Crane is Found



```
No predictions with a score greater than the threshold value!
A JSON file WILL NOT be generated!

Image:  n03126707_13600_JPEG.rf.144156f57d5fe76abcb42c965c416cae.jpg
Data Format:  '[' Predicted bounding box, Score, Object Label ']'

Predicted Bounding Box Data:  [array([ 68.89827, 119.50724, 367.86786, 515.2961
], dtype=float32), 0.83017945, 0]

Predicted Bounding Box Data:  [array([453.03174, 287.70068, 639.2      , 493.1682
], dtype=float32), 0.756801, 0]
```

(b) Output when a Crane is Not Found

Figure 13: Output for RetinaNet Version 7

When working with the python file, the process to us includes a series of arguments in a python command on the command line interface. As can be seen below, the arguments needed for the command to work are:

- Name of the Python File
- Path of the Image to be used
- The Name of the Image
- Path to save the JSON file
- Threshold Value

Format of command to execute RetinaNet_Version7.py:

```
python <name of .py script> <path to image> <image name> <path to save .json file> <threshold value>
```

Example:

```
python RetinaNet_Version7.py TowerCrane.JPEG TowerCrane .\ 0.5
```

Once the command above is passed, the following output is presented:

NOTE: An image with the predicted bounding boxes is presented seperately.

Number of arguments: 5 arguments.

Argument List: ['RetinaNet_Version7.py', 'TowerCrane.JPEG', 'TowerCrane', '.\\', '0.5']

Figure 14: Example of how the python script needs to be used

When the program is run and cranes are found with a value above the threshold, the JSON file will be created, the coordinates will be given to the user on the console, and a window will pop up which shows the original image together with the bounding boxes placed on it.

4.2 Task 3 Limitations - YOLOv7

When working on this task, YOLOv7 was trained on google colab due to hardware limitations, as a result, when any of the trained weights are downloaded locally and tested using a py file they do not work as intended. This is because the model uses PyTorch and not Tensorflow which held the program back from being able to output the data needed. Even though this was unattainable, this program was still implemented as a jupyter notebook inside google colab and run to show that as a model it is still perfectly adequate to be used as an object detector and set a number of bounding boxes to the object being detected.

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

~~I/~~ We*, the undersigned, declare that the [~~assignment~~/ Assigned Practical Task report /~~Final Year Project report~~] submitted is ~~my~~ / our* work, except where acknowledged and referenced.

~~I/~~ We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Evangeline Azzopardi
Student Name



Signature


Kian Parnis
Student Name


Signature

Mario Vella
Student Name


Signature


Student Name


Signature

ARI 3129
Course Code

Applied Computer Vision Task
Title of work submitted

23rd January 2023
Date