Group 15

Kieran Brady - 21120688, Leo Pickett - 21120136, Neha Agrawal - 1924792, Zak Ison - 1831146, Noman Paul - 2048790, Yao Li - 21018972, Mohana Krishna Battula - 21088517, Lakshay Talwar - 21113561

Object Classification and Localisation – Group 15

Group Number	15
Project Title	Object Classification and Localisation
Supervisor	Michael Macaulay
Student IDs	21120688, 21120136, 1924792, 1831146, 21018972, 2048790, 21088517, 21113561

Contents

About:

This project investigates finding an appropriate model for classification and localisation on the Pascal VOC 2012 dataset.

The dataset available at http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html.

Throughout this project, the team explores the dataset, and applies a machine learning model to classify the image, before determining where in the image and object is.

This project resulted in only single object classification and localisation.

The dataset consists of 17,125 images, all of which are in RGB colour format.

Assigned Project Roles:

- Descriptive analysis of the dataset and Error analysis
Kieran Brady, Leo Pickett

- Pre-processing and Literature review
Zak Ison, Noman Paul, Yao Li

- Implementation and Results
Neha Agrawal, Mohana Krishna Battula, Lakshay Talwar

Group Rep:
Kieran Brady

*NOTE – The assigned roles do NOT reflect the roles that were actually carried out.

Literature review

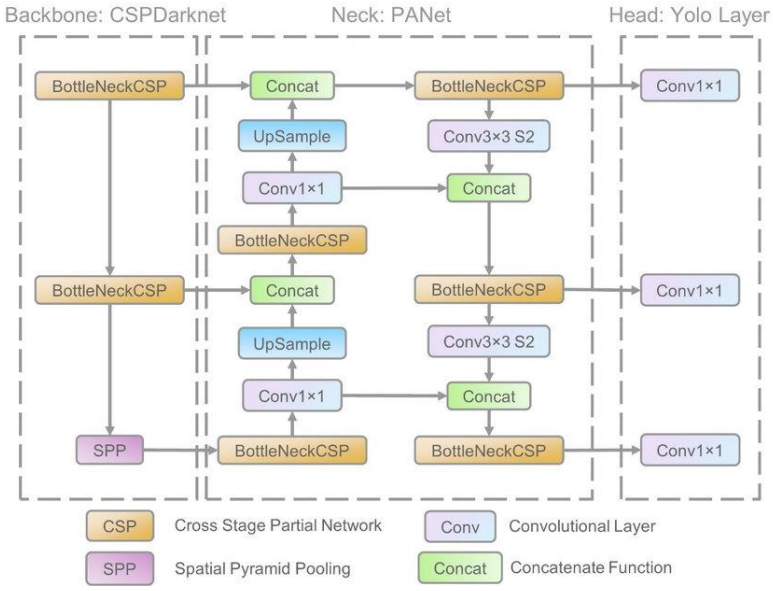
The projects/models most related to this project are:

- Yolo
- Faster RCNN
- Xception
- VGG (16/19 versions)

They are pieces of work that have taken place in the area of computer vision, all with different approaches. Some focus primarily on classification, and others also include object localisation as well.

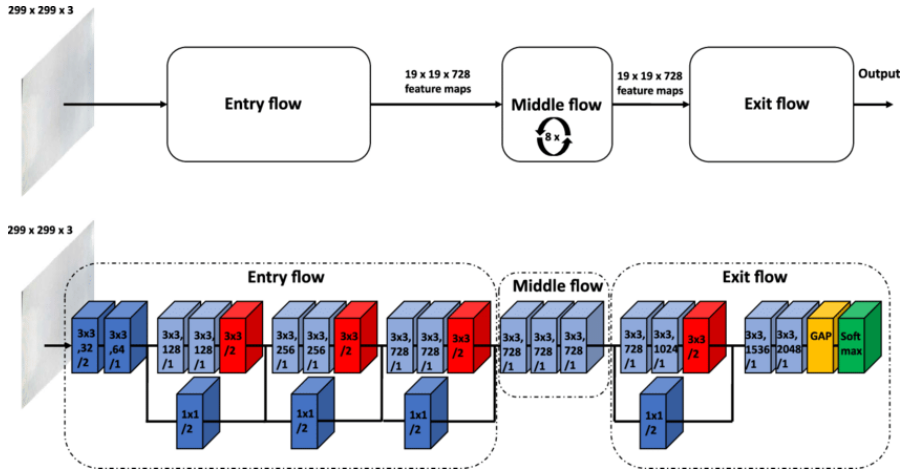
Yolo focussed heavily on statistical analysis of the images, and then applied various methods to tackle some of the issues that they came across. Yolo originally was an improvement to the RCNN model architecture but was able to improve both accuracy and performance speed. Originally trained on the COCO dataset, with includes over 330k images, over 200k of which were annotated, with over 80 object categories. With regards to pre-processing in Yolo v5, the images only need to be the same size for the train and test samples, but there is no specific value required. Yolo models were trained using several augmentations, notably mosaic, blur, mix up, cut mix.

An overview of the Yolo architecture is:



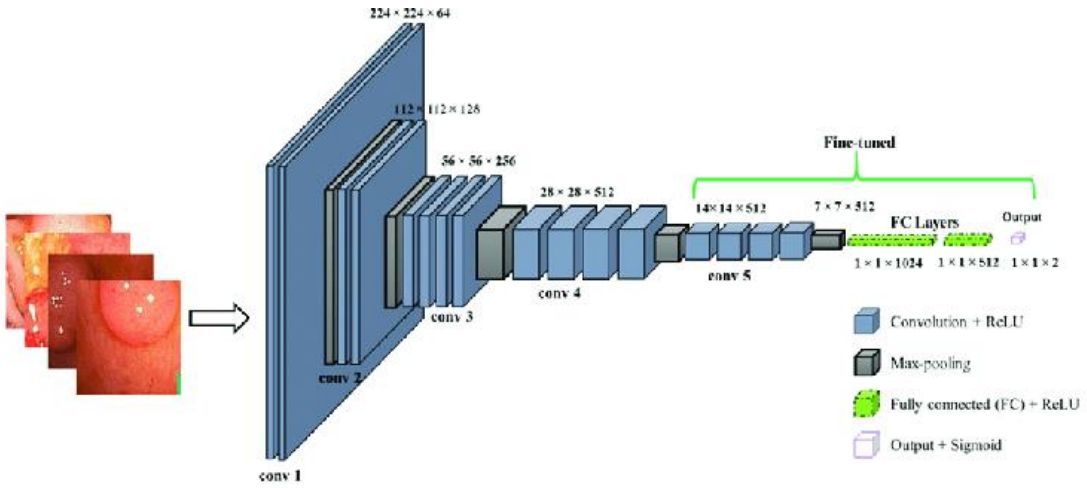
Xception is another great model, created by staff at Google and designed to classify and localise objects. It is an extended version of the Inception model, and it utilises depth-wise distinguishable convolutions. The Inception model was trained on the ImageNet dataset, however the Xception model was trained on a dataset comprising of 350 million images and 17,000 classes. With regards to pre-processing, Xception models require images to be scaled to the same size of 229x229 pixels, as this is what the model was trained on.

An overview of the Xception architecture is:



VGG models are used for classification, however because they yield great classification results, they are often used as a base for other models, and this is known as transfer learning. VGG models were originally trained on the Imagenet dataset, which has 14,197,122 images with 1000 class types. With regards to pre-processing, VGG models were trained using colour altered images, i.e., removing the train sample mean colour from each image. This reduces the amount of information contained in each image, and thus reduces the training time. Also, VGG models require all images to be scaled to the same size of 224x224 pixels, as this is what the model was trained on.

An overview of the VGG19 architecture is:



The main difference between this project and other work going on in this area is the use of transfer learning in conjunction with the testing of various image augmentations. This project builds upon work done on VGG19, and tests various pre-processing techniques such as image augmentations.

The advantages/disadvantages that other projects are:

- Yolo is significantly faster as it only looks at images once, this has the added benefit that it generalises better as it never over learns the training set. Yolo is also capable of multi object classification and localisation.
- VGG is significantly slower but provides great classification accuracy. VGG models are not capable of multi object classification in its raw form, and it was never trained for object localisation.

The advantages/disadvantages of this project are:

- This project utilises the great classification accuracy that the VGG19 model offers and through adaptations to the final few layers, it is able to localise objects with a decent accuracy.
- The main disadvantage of this project is that it does not capitalise on the progress of other projects. Due to technical and time related issues, this project could only employ a few of the great techniques that have been discovered so far in other related work.
- The given dataset is another drawback, for example the statistical analysis in this project was done by utilising the pre-existing annotation information, and this was not extensive enough to apply some other methods applied in other models, i.e., foreground and background balancing solutions.

Ideas that we have used that have been applied in other projects are:

- Object/class balancing
- Single object classification and localisation
- Reducing the colour distribution, i.e., subtracting the mean colour from training sample images
- Mean intersection over union for localisation accuracy
- Various image augmentations

Ideas that could have been explore given more time:

- How certain image augmentation affect certain class types
- Multi object classification and localisation
- Various types of balancing solutions, such as:
    - class imbalance is where certain class types have different levels of representation within the dataset. Having an imbalanced dataset with similar class types can lead to significantly poor performance.
    - scale imbalance is where objects appear in different sizes throughout the dataset. Having some classes only appearing as small objects makes it harder for the model to detect i.e., less feature details.
    - spatial imbalance relates to the judgement on how good a bounding box is, i.e., regression penalty, and IOU score.
    - object imbalance is similar to class imbalance, but focussing how the occurrences of the objects themselves
    - Foreground and background imbalance can highly affect model performance. This is different to scale imbalance, as in it is possible to have a small object in the foreground, or a larger object in the background.
    - Feature level imbalance

Methodology

Below is a description of machine learning methods used within this project:

Exploratory data analysis – This is a deep statistical analysis of a given dataset. The intention is to identify and highlight any important features, imbalances, and errors that may exist.

Class balancing – This is a data pre-processing step that can be used to improve accuracy on minority/under-represented classes.

Transfer learning – This is a method for capitalising on the work of others. The process involves using a pre-train model, and adapting/improving the model, without having to train a model from scratch.

Image augmentation – To reduce overfitting, and to make sure that the models are learning the correct features within an image, augmentations are used. There is an ever-increasing number of techniques and augmentations that can be applied, however in this project only a few are tested (rotations, altering the colour skew, flipping, changing the brightness, and sharpening).

Intersection over union – This is metric that can be used to help the model localise more accurately. Through the comparison of the ground truth bounding box and the predicted bounding box, the IOU can be calculated and the error minimised.

Fine Tuning - This is the process of optimising the model architecture once a stable version has been created. By recording tests made and their associated results, decision making becomes significantly better. In this project, various tests are conducted with regards to augmentations, metrics, model optimisers, and base models.

Here is a brief description of the implementation of both the Yolo v5 and Xception. These models were adapted to the same dataset for a comparison of results.

YoloV5 - The first, and most arduous step was converting the .XML annotation files provided in the PASCAL VOC dataset to YOLO format. The difficulties arose when realising that the .XML files were not consistent in their formatting, which required more expansive code to correct this. We did this locally within a python environment rather than Google Colab, as we thought it would be more stable.

We chose not to do any external pre-processing due to it being a pre-built model, and we wanted to be fair when comparing to our customised models.

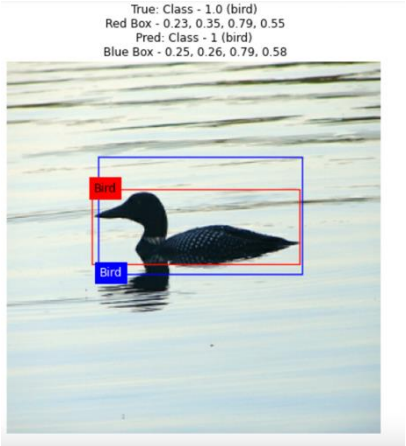
After the conversion, we loaded the dataset into Colab through Google Drive and began training. We chose 50 epochs, though growth began to slow after 25. The choice to include a batch size of just 3 was due to higher batch sizes dramatically increasing runtime. As it was, the total runtime for training exceeded 10 hours. Another parameter of note was hyp.VOC.yaml. This contained the tuning of data augmentation hyperparameters, including <add parameters, can't access from this laptop>.

Xception - In this model, we only chose three classes, the bird, the dog and the cat. We used Xception as a base model, and to achieve both classifier and localiser, we made some adjustments.

We created the convolutional base to combine convolution layers with GlobalAvgPool2D and GlobalMaxPool2D. And add some Dense layers for classification and localisation, Batch Normalization layers to normalise its inputs, Dropout layers helps to reduce overfitting and improve generalisation error. Besides, to avoid overfitting in the neural network, we set an early stopping call-back required in the fit () function.

Then, to define our model's success, we need to define the metric used. So, we used Intersection over Union to evaluate the bounding box accuracy. IoU is an evaluation metric used to measure the accuracy of an object detector on a particular dataset.

We do not pay attention to an exact match of (x, y)-coordinates; we aim to ensure that our predicted bounding box match as closely as possible — Intersection over Union can take this into account. After training the model on Google Colab with a GPU Instance for 65 Epochs (we set 100 epochs, but early stopping) and taking about 2 hours, it achieved a classification accuracy of about 0.9453. The IOU of the two bounding boxes (The actual bounding box is red, and the predicted bounding box is blue) is about 0.6813.

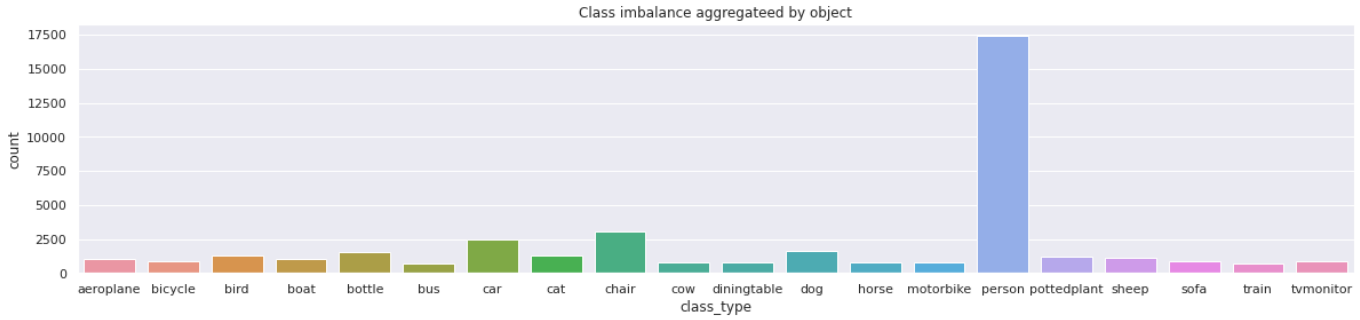


Exploratory Data Analysis

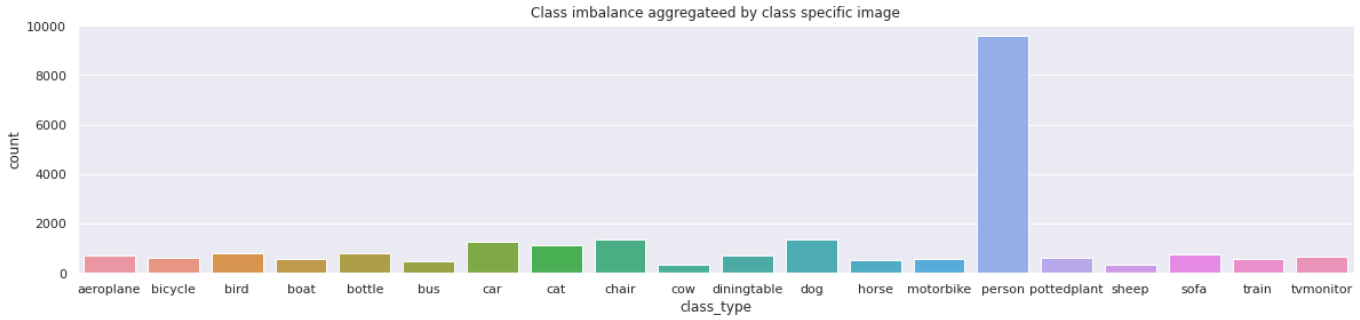
Class Imbalance:

The class imbalance can be seen in the following plots.

There are 40,143 individual objects:



There are 23,739 class specific images (i.e., images that contain only a unique class type):



As is evident from the above plots, the class imbalance is heavily skewed towards the category of ‘person’, thus sample balancing methods are required.

Image Objects:

As the plot shows, most images contain a single annotated object, however there are some images that contain up to 56 annotated objects.

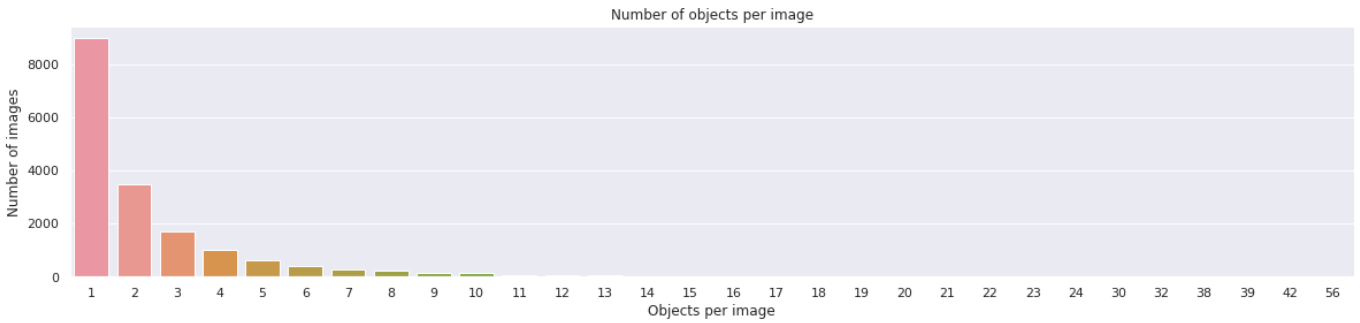
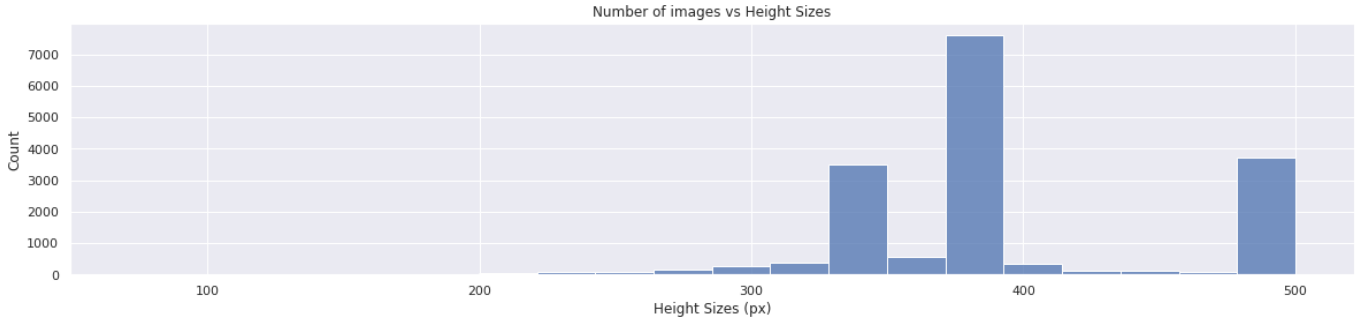
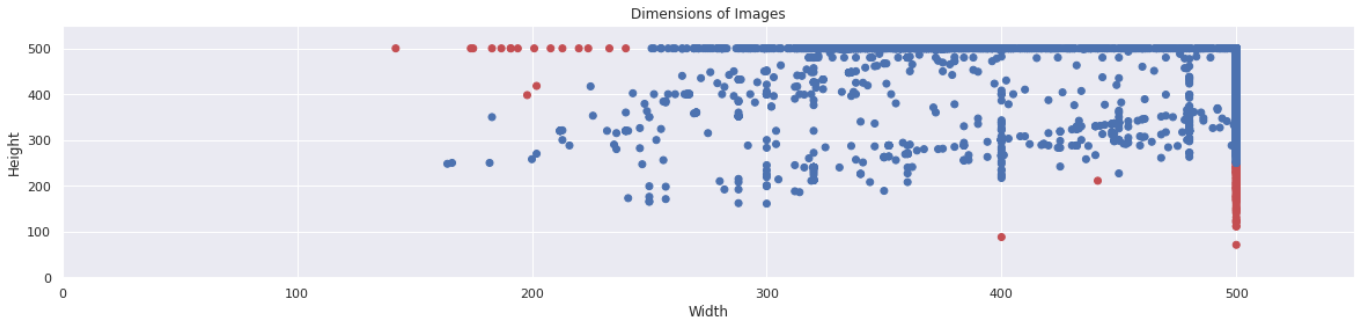


Image Sizes:

The plots below show the distribution of images widths and heights. As is evident, there is huge variation, and thus images will need to be resized during the pre-processing stage.

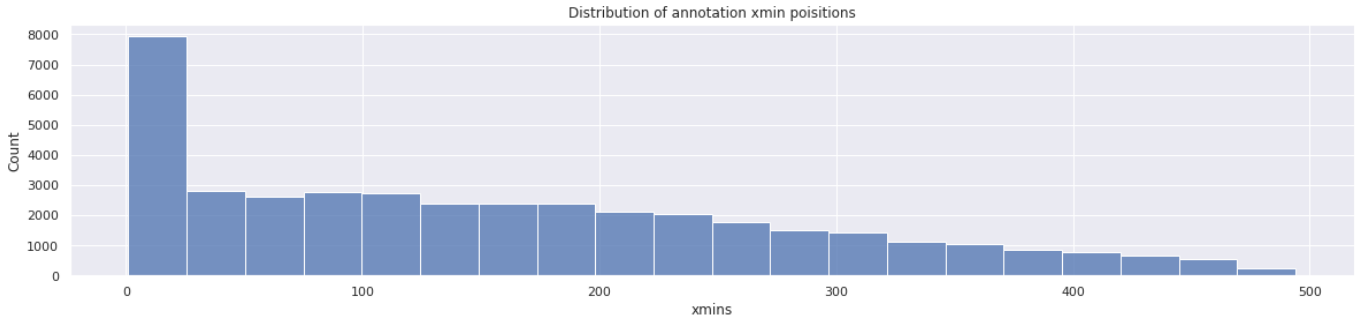


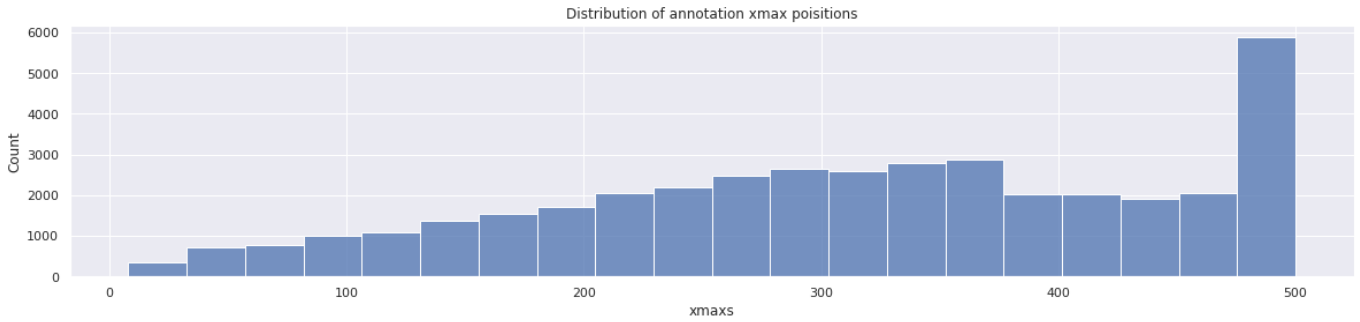
Below is a scatter plot showing the distribution of image sizes, where images that are taller than they are wider are in blue, and images that are wider than they are tall are in red.

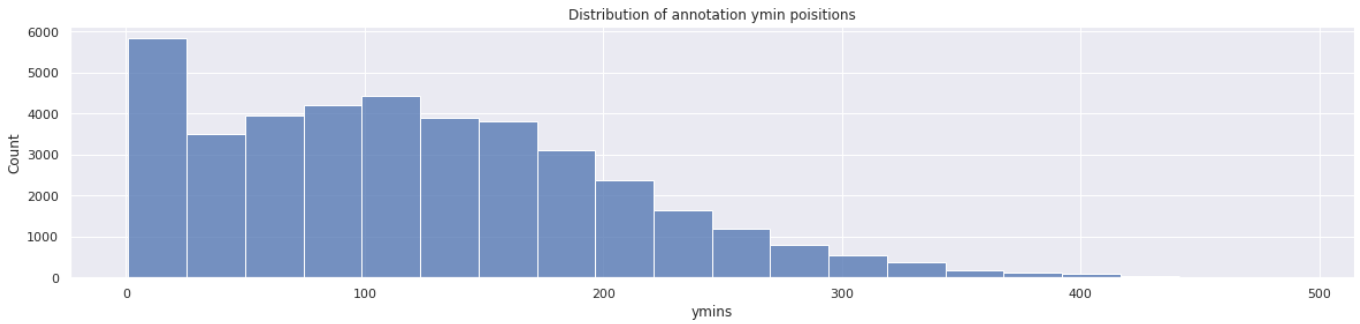


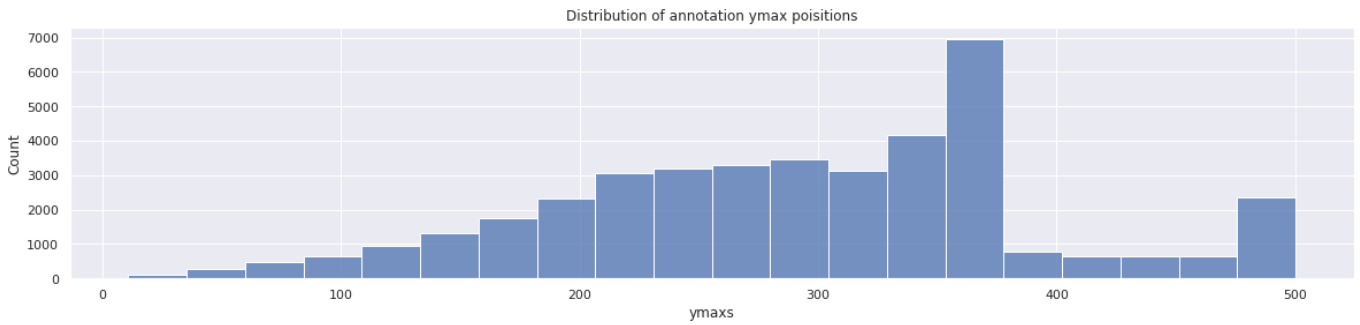
Annotation Positions and Sizes:

The following plots show the distributions of annotation positions across the dataset (in pixels). As the images require resizing, the 'ground truth' annotation boxes will also require scaling, and this will occur during the pre-processing stage.

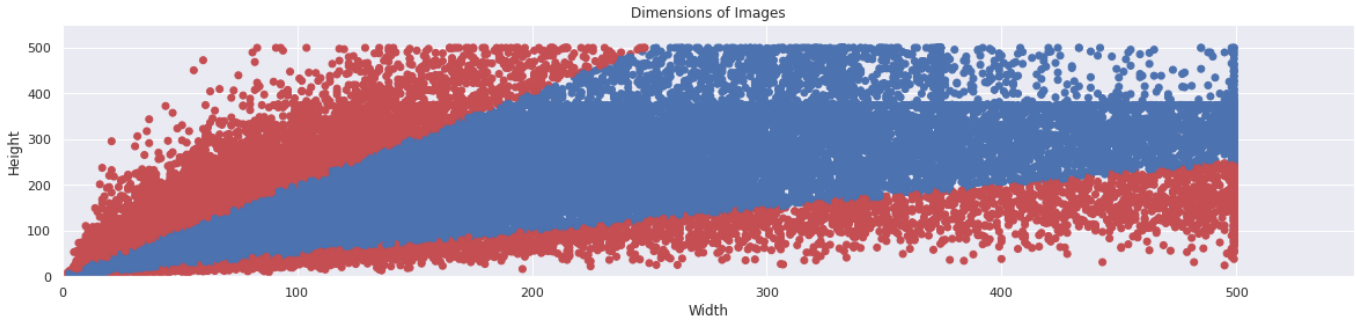




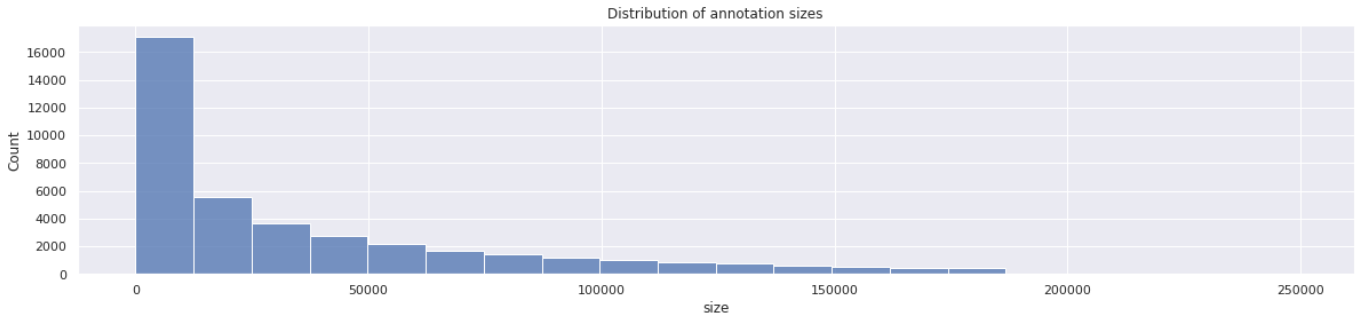




Below is a scatter plot displaying the dimensions of annotation sizes, where annotations that are taller than they are wider are in blue, and annotations that are wider than they are tall are in red.



Below shows the distribution of annotation sizes (measured in pixels squared). It is evident that the sample data set is skewed towards small annotation sizes, and thus small object sizes.



Missing Data:

After comparing the images file names and the annotation files names, no missing images were identified.

Pre-processing

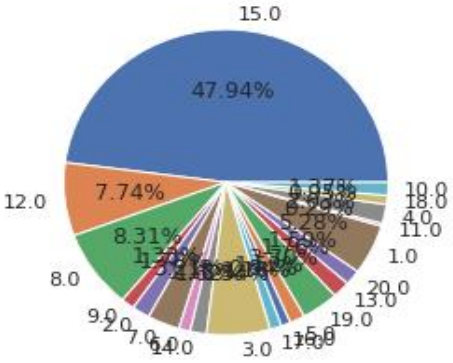
It became evident early on that there would be huge technical challenges with regards to the code required to implement a model that required multiple inputs and that provided multiple outputs. To this date, a workable example for multiple object classification and localisation, has not been identified. Therefore, it was decided that the focus would be on single object classification and localisation, and thus the first data pre-processing included reducing the data sample down to single object images only. This reduced the sample size from 17,125 images to 8000 images.

Following this, a class balancing method was required, due to a significantly skewed sample in favour of the category 'person'. It must be noted that only one class sampling method should have been used at this point, however due to Google Colab runtime errors, specifically RAM usage, that two class balancing methods were used.

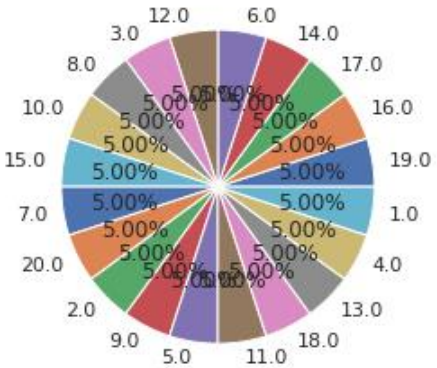
For class types with greater than 400 class specific images, random under sampling was applied, and for class types with less than 400 class specific images, random oversampling was applied. The results of which can be seen in the below plots.

Labels are as follows: ('aeroplane' - 1, 'bicycle' - 2, 'bird' - 3, 'boat' - 4, 'bottle' - 5, 'bus' - 6, 'car' - 7, 'cat - 8', 'chair' - 9, 'cow' - 10, 'diningtable' - 11, 'dog' - 12, 'horse' - 13, 'motorbike' - 14, 'person' - 15, 'pottedplant' - 16, 'sheep' - 17, 'sofa' - 18, 'train' - 19, 'tvmonitor' - 20)

Before balancing method applied:



After balancing methods applied (400 images per class):



After testing various models, overfitting was noticed, as in the models were only obtaining sufficient results on the training samples, therefore a solution for overfitting was required. At this stage various image augmentations were tested, such as:

- Rotations
- Flipping (vertically and horizontally)
- Increasing and decreasing brightness
- Sharpening
- Altering colour skew

Below are examples of the above augmentations:

Original

Rotated 45 degrees

Increased Brightness

Decreased Brightness

Flipped Horizontally

Sharpened

Subtracted mean colour

Once again however, RAM issues prevented the use of many of these being applied at the same time within Google Colab, effectively the x_train arrays would become too large. Therefore, only few were applied, altering the colour skew, sharpening some images, and horizontally flipping some images (every other image to be exact). These were chosen as they yielded the greatest results on the test sample, i.e., enabled the model to generalise well.

The train / test split of the 8,000 sample sizes were:

Train – 6,400

Test – 1,600

Validation was split during the model training phase, set to 10% of the train sample size, i.e. 640.

Implementation

Implementation for the project was technically challenging and various errors were encountered, the main errors being:

- Getting access to the .tar files for the project
  This solved by researching ways to access the files without locally downloading, i.e., removing the possibility of data leaks

- GPU runtime usage limits in Google Colab
  This was solved using multiple Google accounts, and changing runtime type to run locally

- RAM limitations in Google Colab
  This was solved by; deleting local variables, pre-processing images and storing them locally, enabling memory growth on the virtual machine, selecting only a few augmentations to apply

- Model training time
  Only using the available GPU time for training and storing the model and weights locally, all other programming was done with runtime type set to 'None'.

Initially, attempts were made to learn and practice object classification on all the available classes, however this resulted in poor 'test accuracy' models. Then attempts were made to practice on only two classes, as to get a better understanding on object classification. This approach taken, was to identify a model that achieves an appropriate level of test accuracy, before attempting to learn about object localisation. Eventually, it was discovered that using pre trained models such as VGG16 and VGG19 yielded far greater classification results than any of the self-built models tested.

Thus, it was decided that a pre-trained model would now act as a base model for this project, although these models did require alteration, i.e., Transfer learning.

The next step was to identify ways in which we could manipulate these base models to allow regression training to take place in parallel. Thanks to the literature review, various partial solutions were found, through sources such as 'Pyimagesearch', 'Kaggle', 'Keras', 'Tensorflow' and multiple 'Stackoverflow' questions.

The first pre-trained model used was the VGG16 model (for classification only), which is a model that has been trained on images that have undergone a colour altering augmentation during the pre-processing stage, hence the reason for selecting a colour augmentation for this project. Loading this model was fairly straight forward thanks to the Keras library, and a 3-hour long tutorial from 'Tensorflow' on Youtube (https://www.youtube.com/watch?v=qFJeN9V1ZsI).

Once loaded, the model had to be adjusted to work with the dataset. The original dataset that VGG16 was trained on, had roughly 1000 unique classification types, and thus the last dense layer had to be reduced to fit this project. For the purpose of learning and practice only, this implementation was applied to only two class types, cats & dogs. Practicing like this reduced loading and training time significantly, until such time all the necessary tweaks had been made. Once an established implementation was ready, the process was applied to all class types, with no significant drop in model accuracy.

An issue that arose when testing this pre-trained model was 'training time', but it soon became apparent that this could be reduced by setting the layers to 'non-trainable' and also setting the

learning rate to a low value for the amended dense layers. This had the added benefit of not negatively impacting the pre-trained weights.

The next step was to implement the parallel regression training. This was achieved by adding additional layers and outputs to the base VGG16 model. The two outputs were 'classifier' and 'bounding box'.

```
classifier_head = keras.layers.Dropout(0.3)(x)
classifier_head = keras.layers.Dense(21, activation='softmax', name='label')(classifier_head)
reg_head = keras.layers.Dense(128, activation='relu')(x)
reg_head = keras.layers.Dense(64, activation='relu')(reg_head)
reg_head = keras.layers.Dense(32, activation='relu')(reg_head)
reg_head = keras.layers.Dense(4, activation='sigmoid', name='bbox')(reg_head)
model_pre_trained = keras.models.Model(inputs=[inputs], outputs=[classifier_head, reg_head])
```

It was at this stage that, focus and attention was required on the image augmentations, as the training set now had to include both labels and ground truth bounding boxes. These 'ground truth' bounding boxes also had to follow the same augmentation process as the images, i.e., if an image was flipped horizontally, then so too did the bounding boxes.

As for the models' complier attributes, two different optimisers were tested

- Stochastic Gradient Descent
- Adam Optimiser

The Adam optimiser consistently yielded greater accuracy results during this project, and it was therefore selected as the final optimiser of choice, in conjunction with a low learning rate, as to not significantly impact the pre-trained weights.

As for the loss attribute, two different parameters were required, one for the classification and another for the regression. For classification, sparse categorical cross entropy was selected, as the labels were not 'One Hot Encoded'. As for the regression, the Mean Square Error was selected, as the aim was to minimise this as much as possible, the distances between the true values and the predicted values.

Loss weights had to be implemented equally as to ensure that the model was not focussing / learning on one output more than the other, therefore the loss weights for both classification and regression were set to 1.

With regards to the metrics, three metric types were evaluated:

- Accuracy
- Intersection Over Union
- Mean Intersection Over Union

Whilst building and evaluating the very first models, accuracy was the main metric used for both classification and regression, and indeed it did provide some decent results. Following this, intersection over union was evaluated. Overall, it was technically challenging to apply this within the model training itself, the only results achieved with this method was on a single image with two dummy bounding boxes. Therefore, it was decided that the Keras model parameter for Mean Intersection over Union would be used, and this yielded greater test sample accuracy than the accuracy metric.

One Keras feature that was helpful was the 'save model' function. Due to RAM issues, i.e., being very close to full usage, the 'save model function' was used to store the model locally in case of the runtime automatically being restarted.

Once a stable model had been created, the stage of fine tuning could begin. Fine tuning for this project meant trying various:

- Image augmentations
- Learning rates
- Optimisers
- Epoch sizes
- Batch sizes
- Pre-trained models

Before finalising on a model, fine tuning tests were conducted. All tested models below have the same optimiser learning rate of 0.001, the same number of training epochs equal to 10 (early stopping with patience of 5 on classification validation loss) and the same training batch size of 40 (due to RAM issues, batch sizes could not be larger). Also, all the images were augmented by removing the train sample mean colour from each training image.

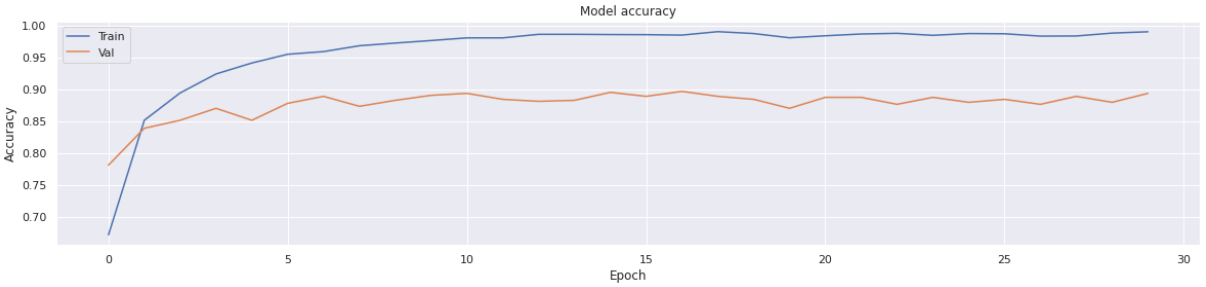
Base Model	**Additional Augmentations**	**Optimiser**	**Average Recall**	**Average Precision**	**Average F1 Measure**
VGG16	50% of training sample sharpened	SGD	0.812	0.822	0.814
VGG16	50% of training sample flipped horizontally	SGD	0.78	0.79	0.78
VGG16	50% of training sample sharpened	Adam	0.86	0.86	0.86
VGG16	50% of training sample flipped horizontally	Adam	0.83	0.84	0.83
VGG19	50% of training sample sharpened	SGD	0.83	0.83	0.82
VGG19	50% of training sample flipped horizontally	SGD	0.77	0.78	0.77
VGG19	50% of training sample sharpened	Adam	0.86	0.87	0.86
VGG19	50% of training sample flipped horizontally	Adam	0.83	0.84	0.83

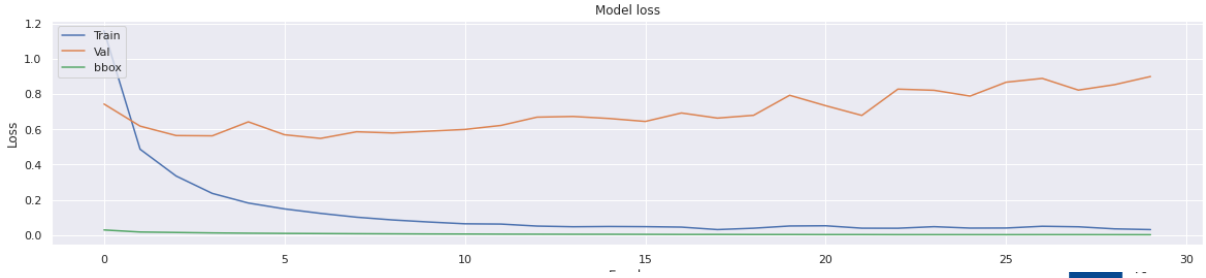
Testing both augmentations did not yield better results than sharpened alone:

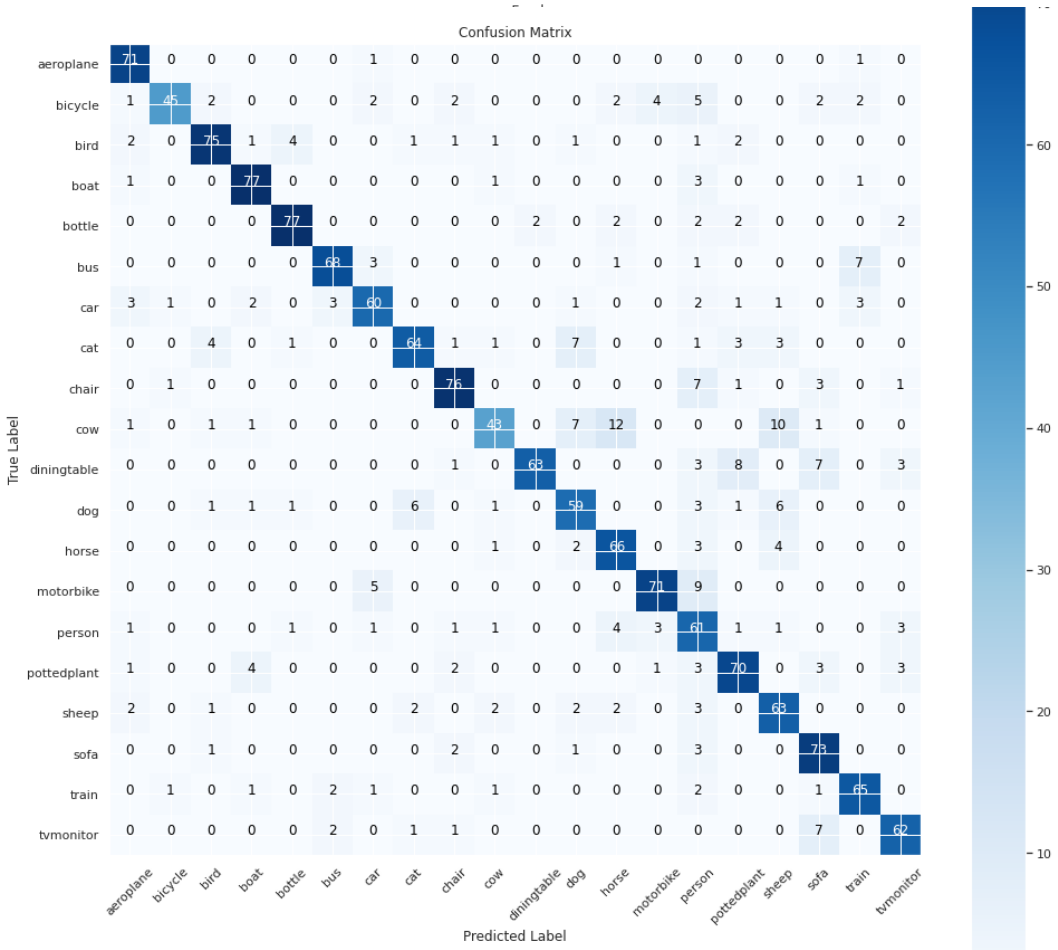
Base Model	**Addition Augmentations**	**Optimiser**	**Average Recall**	**Average Precision**	**Average F1 Measure**
VGG19	50% of training sample sharpened and the other 50% flipped horizontally	Adam	0.85	0.86	0.85

The early monitor parameter was set to ‘bbox_loss‘ on 30 epochs (patience 5) for the following tests:

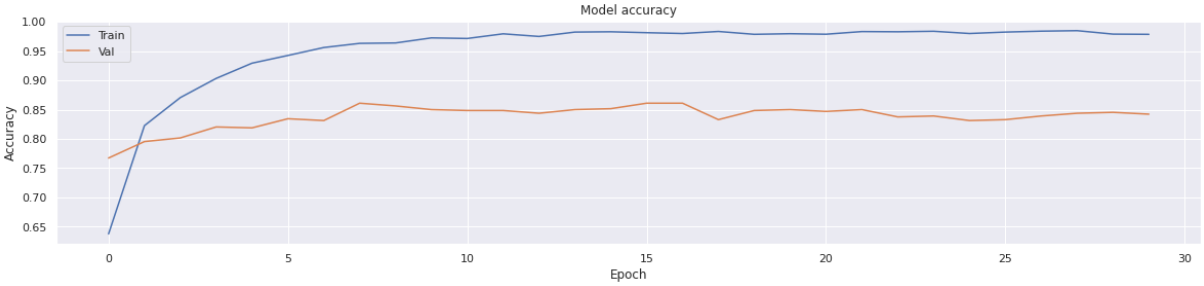
Base Model	**Addition Augmentations**	**Optimiser**	**Average Recall**	**Average Precision**	**Average F1 Measure**
VGG19	50% of training sample were flipped horizontally	Adam	0.82	0.83	0.82

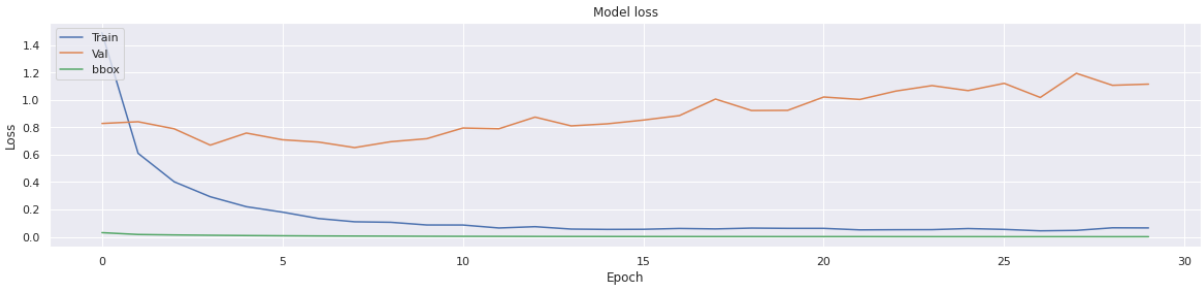


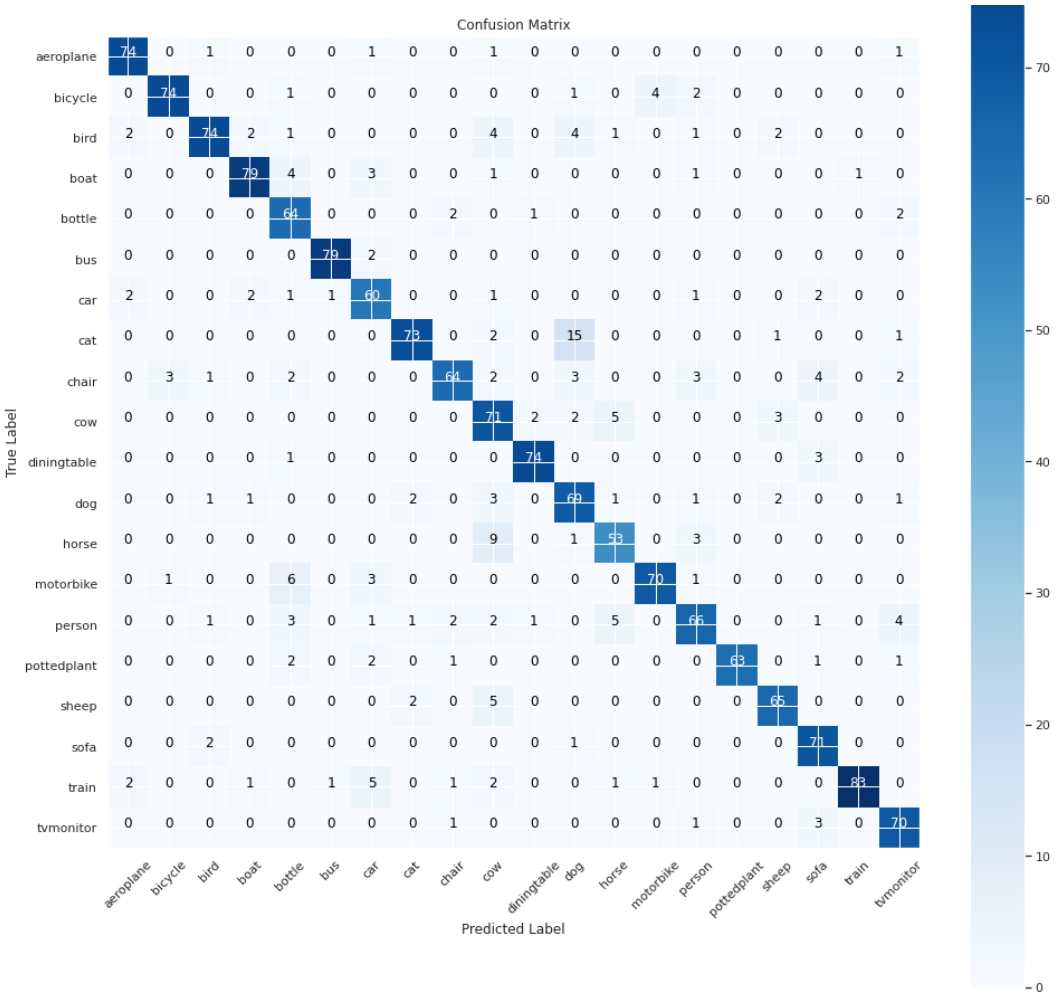




Base Model	**Addition Augmentations**	**Optimiser**	**Average Recall**	**Average Precision**	**Average F1 Measure**
VGG19	50% of training sample were sharpened	Adam	0.88	0.88	0.87







By comparing the two confusion matrices, it is possible to identify those different augmentations affect different object types, for example flipping horizontally yielded worse results for cows and sheep, whereas sharpening yielded worse results for cats and dogs. Further investigation could be taken here with more time.

At this stage it was decided that a VGG19 base model would be used, along with a sharpened and subtracted mean colour augmentations. As is evident from the fine-tuning tests above, VGG19 performed better than VGG16, and the Adam Optimiser performed better than the Stochastic Gradient Descent. The difficult choice that remained was augmentation types, but overall sharpened fared better in the metrics. It must be noted that there were minor fluctuations in results every time a different test sample was loaded, this could of course be minimised by increasing the test sample size.

Including image augmentations, meant a better generalisation for bounding boxes. It is also worth mentioning that oversampling in general would also affect classification accuracy as well, however because a pre-trained model was used, any apparent overfitting with regards to classification was not as big of a concern as getting a better generalisation on the bounding boxes.

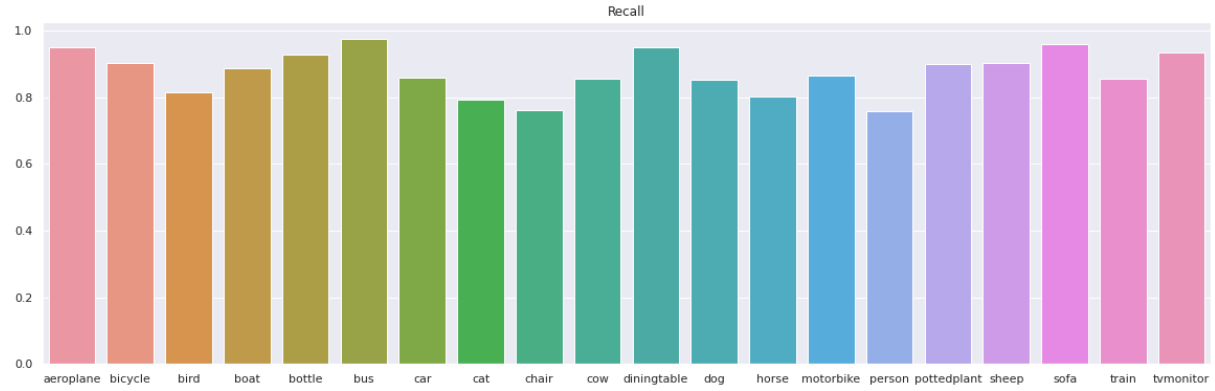
As a side note, it was difficult to analyse the Keras Mean IOT, as it converts the continuous predictions to its binary value, i.e., taking the binary digit before decimal point as predictions (0.99, 0.50, 0.01 become 0, and 1.99, 1.01 become 1).

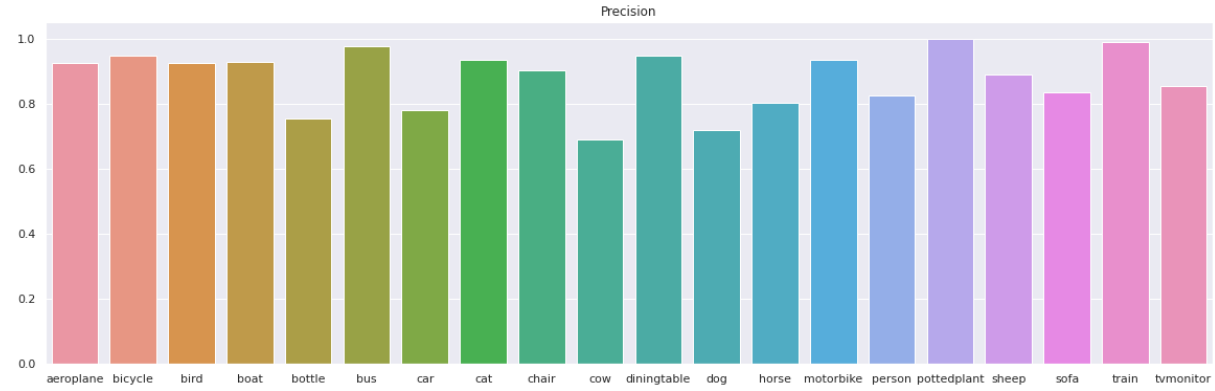
Results

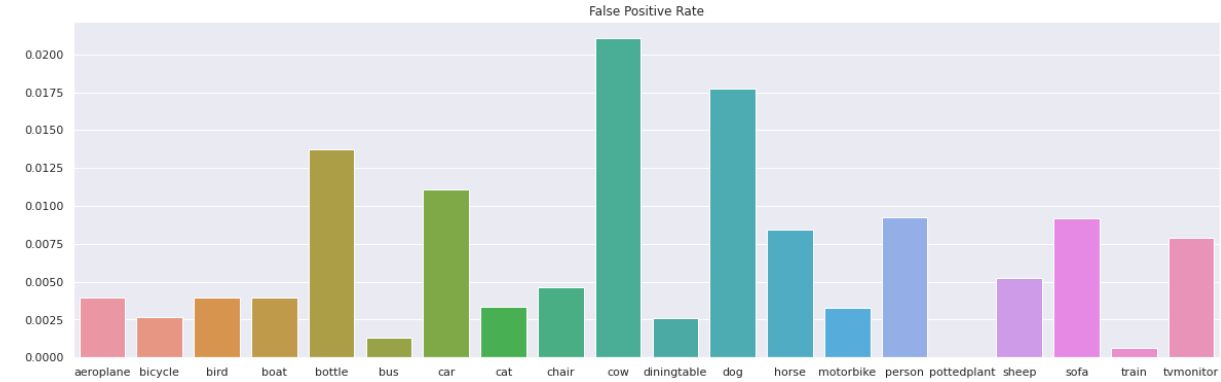
The results below are from the final model selected – VGG19 Base, 30 epochs (patience 5 on bound box loss), using the Adam optimiser and using both the subtracted mean colour and sharpening augmentations.

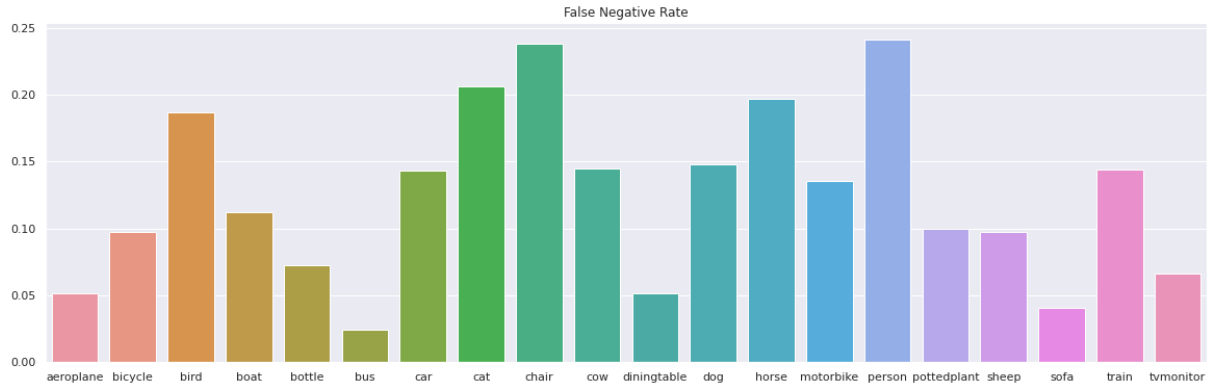
Average Recall	Average Precision	Average False positive rate	Average False Negative Rate	Average F1 Measure
0.876	0.88	0.0067	0.1250	0.87

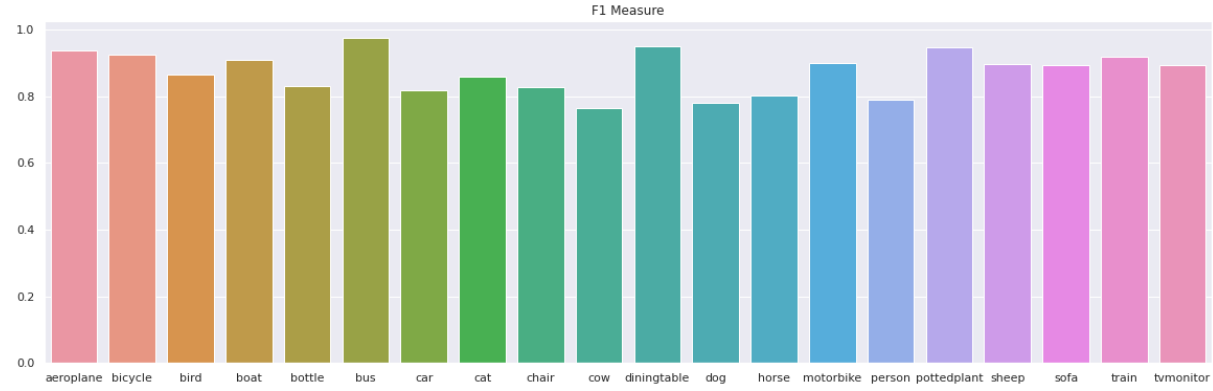
Individually there are as follows:



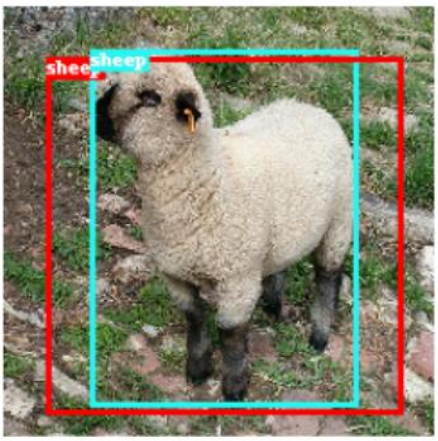






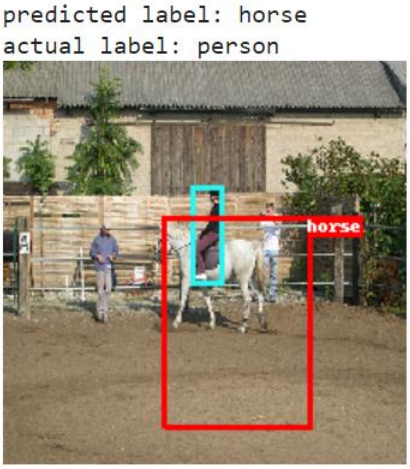


One of the interesting points about these results is the false positive and false negative values, as there appears to be a correlation with images that were under sampled and slightly higher false readings. Below are some examples of well classified and localised objects (Blue is ground truth, and Red is the prediction):

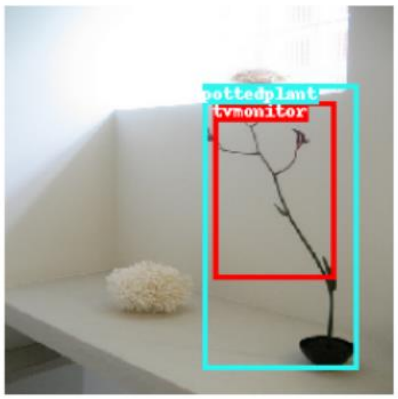


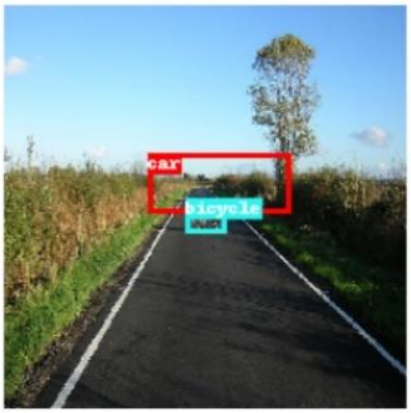
Error Analysis

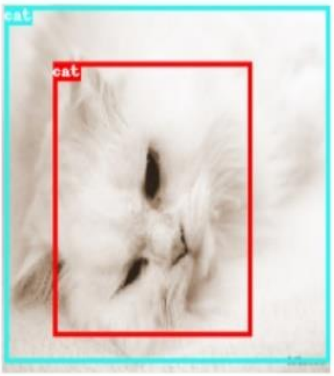
Below are some interesting results from all the models tested before the final model was chosen. They demonstrate potentially difficult images to classify and localise objects within, due to multiple objects, abnormal colours and textures, and even similar shapes. For example, the first image has a person and a horse, but the image was only annotated as a person. Another example is the black cat next to a statue of a black cat, this is challenging for the model to distinguish between. Some of these results are fascinating, and probably highlights examples of purposefully including difficult images to train models on by the original project authors, i.e., the fourth picture below includes a rabbit, which isn't even an object class for this project.

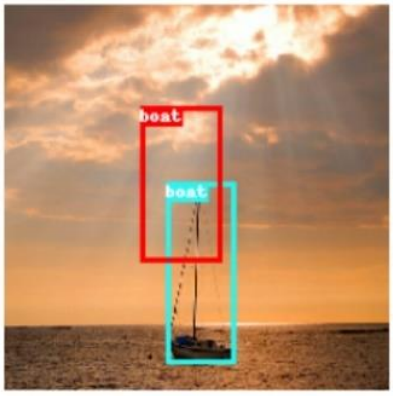


Below are more mis classified images, the first one identifies a ‘TV monitor’, therefore one can only assume there were multiple images of TVs with cracked screens.









Conclusion and future improvements

Overall, the project has met its intended objective, to classify and localise objects within images. The final model does this does this with a fairly decent degree of accuracy, however it is certainly better for classification than it is for localisation. It is also worth mentioning that it appears that the classification and localisation predictions are not coupled together, i.e., they are independently learning object features.

Improvements would be:

- Adapting the code to do multi object classification and localisation
- Investigating how different augmentations affect certain classification types
- Improving the regression predictions

References

Tensorflow: The most useful resource used in this project was a video posted by Tensorflow on Youtube (https://www.youtube.com/watch?v=qFJeN9V1ZsI). This 3-hour long tutorial explains a significant amount of information in granular detail. This resource by itself help to establish the base model used for this project.

Other useful Tensorflow documents were:

- https://www.tensorflow.org/guide/keras/transfer_learning
- https://www.tensorflow.org/api_docs/python/tf/keras/Model

Keras: Documentation from Keras was critical to this project, although admittedly it would better if they had mode examples, specifically for models that require multiple inputs and outputs.

- https://keras.io/api/metrics/segmentation_metrics/#meaniou-class
- https://keras.io/api/losses/
- https://keras.io/api/applications/vgg/

The following resources were also instrumental to this project, as examples from these resources enabled this project to transform from just a classification task to also include localisation. They clarified that localisation is something that is done in parallel with classification and not done as an afterthought. Although many resources were read regarding this, the main webpages are:

PyImageSeach - https://pyimagesearch.com/2020/10/05/object-detection-bounding-box-regression-with-keras-tensorflow-and-deep-learning/

PyImageSeach - https://pyimagesearch.com/2020/10/12/multi-class-object-detection-and-bounding-box-regression-with-keras-tensorflow-and-deep-learning/

PyImageSeach - https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

Kaggle - https://www.kaggle.com/code/vbookshelf/keras-iou-metric-implemented-without-tensor-drama/notebook

Kaggle - https://www.kaggle.com/code/arpitjain007/bounding-box-regression/notebook

Towards Data Science - https://towardsdatascience.com/object-localization-using-pre-trained-cnn-models-such-as-mobilenet-resnet-xception-f8a5f6a0228d

Towards Data Science - https://towardsdatascience.com/object-localization-using-pre-trained-cnn-models-such-as-mobilenet-resnet-xception-f8a5f6a0228d

Towards Data Science - https://towardsdatascience.com/object-detection-with-neural-networks-a4e2c46b4491

Towards Data Science - https://towardsdatascience.com/faster-rcnn-object-detection-f865e5ed7fc4#:~:text=Faster%20RCNN%20is%20an%20object,SSD%20(%20Single%20Shot%20Detector).

Medium - https://medium.com/nerd-for-tech/building-an-object-detector-in-tensorflow-using-bounding-box-regression-2bc13992973f