# DSA4266 Sense-making Case Analysis: Science & Technology


**Group 3:**

Colin Ng Chenyu (A0189871J)

Gary Lee Jia Jin (A0183171J)

Tan Yoong Kang, Colin (A0189819E)

Woon Hao Xuan (A0182700N)

## Table of Content

# Fish Larva Count Project

## Abstract

Singapore has large scale fish production, which is essentially about rearing fish larvae to a size that's ready for our dinner tables. There is a need to actively monitor the stock of the fish larvae.

The officers at Singapore Food Agency (SFA) monitor the stocking density of the fish larvae stocked in larviculture tanks, by counting the number of larvae at regular intervals. This procedure is conducted to derive a suitable amount of live feed for the fish larva. Additionally, the SFA officers also need to count the number of fertilised fish eggs and unfertilised fish eggs to calculate the total number of eggs, fertilisation rate, hatching rate, survival rate etc. for monitoring of the performance of broodstock fish and fish larvae from batch to batch. However, as the task of manually reviewing, labelling and counting is highly labour-intensive, and often subjected to high rate of man-made error, there is a need for a better solution to reduce the manpower required.

Our goal is to train a **predictive model** that is able to **accurately classify** images of petri-dish containing three main classes: *Fertilised Eggs*, *Unfertilized Eggs* and *Fish Larvae*. We are to include an *Unidentifiable* class to handle foreign objects as well. The performance metrics used will be the corresponding mAP scores for the main three classes and their counts when given test images. Additionally, we have developed a webpage where our users can upload their petri-dish images and churn out the predictions of the counts in the image.

An overview of our approach is shown below in Figure 1.

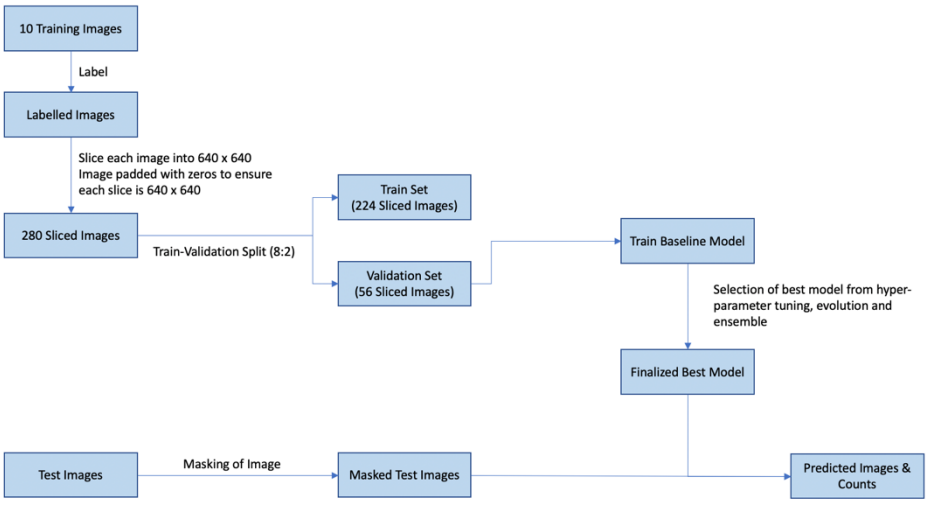



*Figure 1: Overview of approach*

# 1. Data Preparation

## 1.1. Labelling of Data

We were provided 10 raw petri-dish images and given four major classes to label objects within the area of interest. The classes are: 'Fertilised Egg', 'Unfertilised Egg', 'Fish Larvae' and 'Unidentifiable'. We note that the 'Unidentifiable' class serves to label foreign objects present, but have ambiguous features which clearly do not belong to any of the three major classes.

To label our images, we first decided on a threshold for both clarity and size. Firstly, this implies that blurry and faded objects are to be omitted from labelling, hence used by the model to learn about the background class. Secondly, extremely tiny specks and dots are to be omitted from labelling as well. We emphasize that these aforementioned cases are to be ignored and not labelled as 'Unidentifiable', whereby objects labelled by that class are of certain clarity and have substantial size. The program used for labelling of our data is LabelImg, which allows the labels to be saved in a YOLOv3 format. The total counts of labels from the original 10 images are shown in Table 1 below.

| Label Type | Total Number of Labels (all 10 Images) |
|---|---|
| Fertilised Egg | 376 |
| Unfertilised Egg | 244 |
| Fish Larvae | 956 |
| Unidentifiable | 129 |

*Table 1: Table counts of labels from the original 10 images*

Table 2 depicts the counts in each class per image. We note that there is a clear imbalance in classes depending on the image, mainly due to chronological stages of when the photo is taken. Therefore, when conducting our training and validation split, we would want our model to learn from all stages (i.e. from every image given), especially since we are already very limited in the number of training images.

| Image Number | Fertilised Eggs | Unfertilised Eggs | Fish Larvae | Unidentifiable | Remark(s) |
|---|---|---|---|---|---|
| 20210729_131410.jpg (1) | 56 | 3 | 60 | 7 | Low Unfert Eggs |
| 20210729_132515.jpg (2) | 1 | 64 | 41 | 10 | Low Fert Eggs |
| 20210729_132649.jpg (3) | 5 | 137 | 174 | 11 | Low Fert Eggs<br>High Unfert Eggs<br>High Larvae |
| 20210729_134857.jpg (4) | 21 | 3 | 66 | 6 | Low Unfert Eggs |
| 20210729_134912.jpg (5) | 23 | 5 | 64 | 7 | Low Unfert Eggs |
| 20210903_095054.jpg (6) | 42 | 10 | 117 | 34 | High Fert Eggs<br>High Unidentif |
| 20210903_100603.jpg (7) | 90 | 3 | 68 | 13 | High Fert Eggs |
| 20210903_100651.jpg (8) | 43 | 15 | 124 | 13 | High Larvae |
| 20210903_100734.jpg (9) | 76 | 3 | 81 | 18 | High Fert Eggs<br>Low Unfert Eggs<br>High Larvae |
| 20210903_100758.jpg (10) | 19 | 1 | 161 | 10 | High Larvae |

*Table 2: Breakdown of counts of each label*

To resolve this, we decided to slice our images into multiple segments, then perform the training and validation split, allowing the different image conditions to be accounted for in the training phase. This process will be elaborated in the following sections.

## 1.2. Slicing of Training Images and Label Coordinates

### 1.2.1. Phase I Slicing of Training Images and Label Coordinates (Old)

Firstly, we note that the original image dimensions are very large at (4032, 2268) when viewed vertically. Therefore, image slicing is required to allow for better feature extraction when YOLOv3 does its own down sampling of the input image's dimensions by factors of 32, 16 and 8.

Initially, we tried to slice our 10 original images into 64 parts each, resulting in 640 images for training and validation, whereby the dimension for every slice when viewed vertically is (504, 283) as shown below in Figure 2. Afterwards, we had to slice and re-normalize the bounding box coordinates done via LabelImg according to the image slice it belonged to.

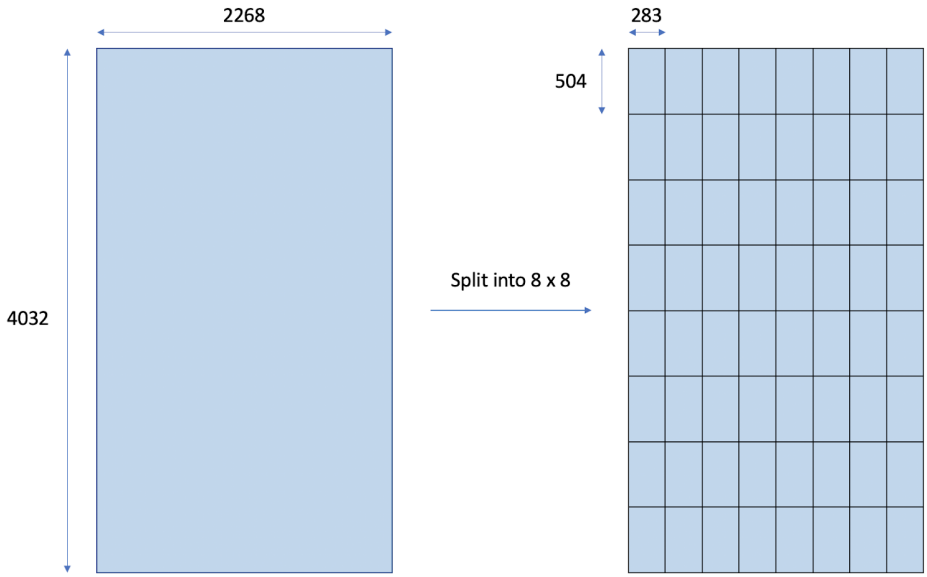



*Figure 2: Old splitting of image into 64 parts*

The original labels were in the format of (x, y, width, height), whereby (x, y) denote the middle of the bounding box drawn. To rescale these coordinates, the following equations were used:

Old Slicing Logic:

$$Number\ of\ Slices\ per\ Image\ =\ 64$$

$$split\ factor\ =\ \sqrt{Number\ of\ Slices\ per\ Image}\ =\ \sqrt{64}\ =\ 8$$

$$interval\ =\ \frac{1}{split\ factor}\ =\ \frac{1}{8}\ [each\ step\ to\ the\ next\ slice\ is\ \frac{1}{8}\ of\ dimension]$$

$$For\ each\ coordinate\ (x_{old},\ y_{old},\ width_{old},\ height_{old}):$$

$$stepsize_x\ =\ np.floor(\ x_{old}\ /\ interval)$$

$$stepsize_y\ =\ np.floor(\ y_{old}\ /\ interval)$$

$$x_{new}\ =\ (x_{old}\ -\ (stepsize_x\ \times\ interval))\ \times\ split\ factor$$

$$y_{new}\ =\ (y_{old}\ -\ (stepsize_y\ \times\ interval))\ \times\ split\ factor$$

$$width_{new}\ =\ width_{old}\ \times\ split\ factor$$

$$height_{new}\ =\ height_{old}\ \times\ split\ factor$$

$$return\ (x_{new},\ y_{new},\ width_{new},\ height_{new})$$

We note while slicing the images naively in our initial method, certain bounding boxes in the original image would be sliced partially as well, especially for slices taking place in the middle of the area of interest. As a result, we get the following numbers of original sliced labels shown in Table 3.

| Label Type | Number of Sliced Labels | Percentage of Total |
|---|---|---|
| Fertilised Egg | 49 | 13% |
| Unfertilised Egg | 24 | 10% |
| Fish Larvae | 152 | 16% |
| Unidentifiable | 13 | 10% |

*Table 3: Old total counts of sliced labels and its percentage*

### 1.2.2. Phase II Slicing of Training Images and Label Coordinates (New)

However, since YOLO's prediction is done on squared slices, we decided to try an alternate slicing method to ensure that the training and predicted slices are of similar aspect ratio and dimensions. Upon investigating the source code for how YOLO handles different image sizes during prediction defined by their dataloader: LoadImages(dataset, img_size = IMAGE_SIZE), we see that the letterbox function (derived from utils.datasets) uses border padding and maintains the scale ratio. Therefore, we will not be expecting much improvement in performance results with this switch in slicing methods. Nonetheless, we will utilize and follow through with this format of slicing for our training, validation and test images, if necessary.

In detail, the. With some tuning, we have fixed $d = 640$, so each slice would be (640, 640) in dimension. In general, a smaller value of $d$ would yield smaller slices, faster training speed per slice, but an overall increase in the total number of slices. Additionally, more objects may be sliced along the edges as well. These factors are accounted for when we tune this hyperparameter, concluding that $d = 640$ is ideal. Acknowledging that the incoming images may not be perfect squares, the right and bottom of the image is padded with zeros to ensure each slice is of (640, 640). This process can be visualised below in Figure 3.

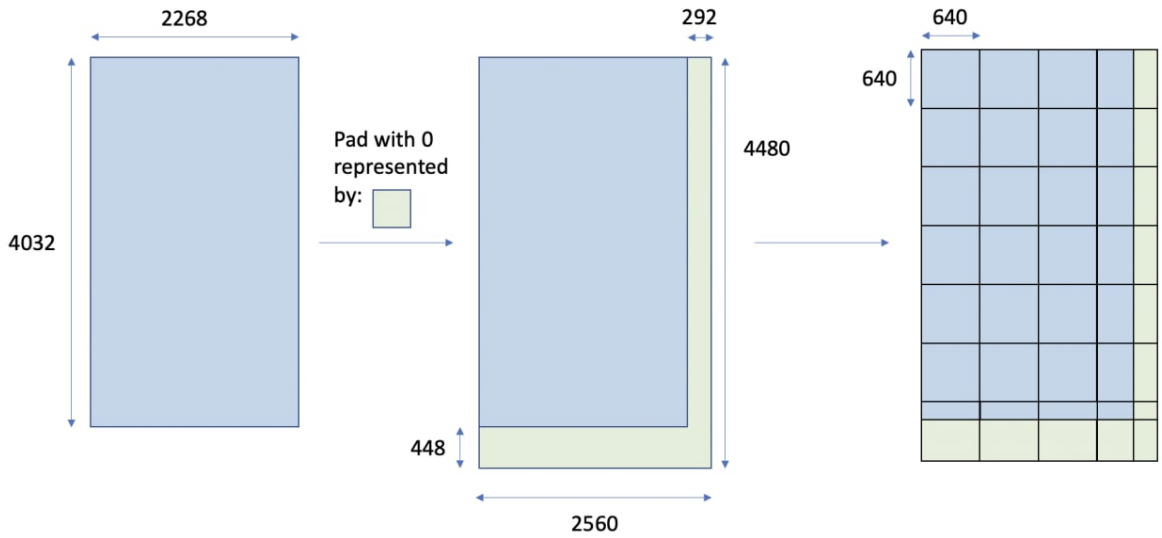



*Figure 3: New splitting of image into 28 parts, each (640, 640) squares*

Similar to the previous slicing method, our next step is to rescale the original coordinates of the bounding boxes drawn using LabelImg. We first determine the amount of pixels, if needed, to be padded

right and bottom to form a square. The labels are then rescaled with a different set of mathematical equations:

New Slicing Logic:

$$Slice\ Size\ Dimension,\ d\ =\ 640$$

$$For\ each\ image\ of\ dimension\ (\ height_{original},\ width_{original}):$$

$$height_{pad}\ =\ int(math.ceil(height_{original}/d)\ \times d\ -\ height_{original})$$

$$width_{pad}\ =\ int(math.ceil(width_{original}/d)\ \times d\ -\ width_{original})$$

$$width_{new,}\ height_{new}\ =\ cv2.copyMakeBorder(image,\ 0,\ height_{pad},\ 0,\ width_{pad}).shape[:2]$$

$$For\ stepsize_{x}\ in\ range(math.cel(width_{new}/d)):$$

$$For\ stepsize_{y}\ in\ range(math.cel(height_{new}/d)):$$

$$x_{start}\ =\ d\ *\ stepsize_{x}$$

$$x_{end}\ =\ d\ *\ (stepsize_{x}+1)$$

$$y_{start}\ =\ d\ *\ stepsize_{y}$$

$$y_{end}\ =\ d\ *\ (stepsize_{y}+1)$$

$$imgslice\ =\ image[y_{start}:y_{end},\ x_{start}:x_{end},\ :]$$

$$For\ each\ label\ (x_{old},\ y_{old},\ width_{old},\ height_{old})\ belonging\ in\ this\ slice:$$

$$x_{new}\ =\ (x_{old}\ \%\ d)\ /\ d$$

$$y_{new}\ =\ (y_{old}\ \%\ d)\ /\ d$$

$$width_{new}\ =\ width_{old}\ /\ d$$

$$height_{new}\ =\ height_{old}\ /\ d$$

$$return\ (x_{new},\ y_{new},\ width_{new},\ height_{new})\ for\ each\ image\ slice$$

Akin to the previous naive slicing method, certain objects would be sliced as they lie on the edges of these slices. But due to our updated technique, we note a large decrease in the number of sliced labels across all classes seen in Table 4, as the number of slice images in total is reduced as well.

| Label Type | Number of Sliced Labels | Percentage of Total |
|---|---|---|
| Fertilised Egg | 28 | 7% |
| Unfertilised Egg | 17 | 7% |
| Fish Larvae | 77 | 8% |
| Unidentifiable | 8 | 6% |

*Table 4: New total counts of sliced labels and its percentage*

Upon referencing Table 3, we see a slight decrease in the total number of sliced labels with respect to the old method. This is ideal as more labels are retained and not removed unnecessarily, allowing for more objects in training. Comparing Table 4 with Table 1 which depicts the original label counts, we also see that the number of sliced labels is of an extremely small percentage. Since we will have sufficient training labels for feature extraction, we have decided to ignore these labels.

Additionally, we note that most images have the central portion as an area of interest containing the objects for prediction, usually indicated by the water blob outline. This implies that many areas of the original image would contain no labels at all (areas at the circumference of the petri dish, extreme corners, etc.) These would be predominantly used to train our model in learning the background class.

## 1.3. Overall Breakdown of Labels after Slicing

Removing the sliced labels while retaining all sliced images leads us to have the overall breakdown of labels for training and validation across 280 slices (the 10 images are divided into 28 each) as shown in Table 5 below.

| Label Type | Overall Number of Labels |
|---|---|
| Fertilised Egg | 348 |
| Unfertilised Egg | 227 |
| Fish Larvae | 879 |
| Unidentifiable | 121 |

*Table 5: Overall counts of labels used after slicing*

## 1.4. Allocation of Train-Validation Split

With our slices and their corresponding labels ready, our next step is to conduct a train-validation split. The main variable to consider is the class-balance ratio, whereby our training set and validation set has to resemble the tallies of the original count as closely as possible. We performed an approximately 80-20 split with random allocation of slices as shown below in Figure 4.

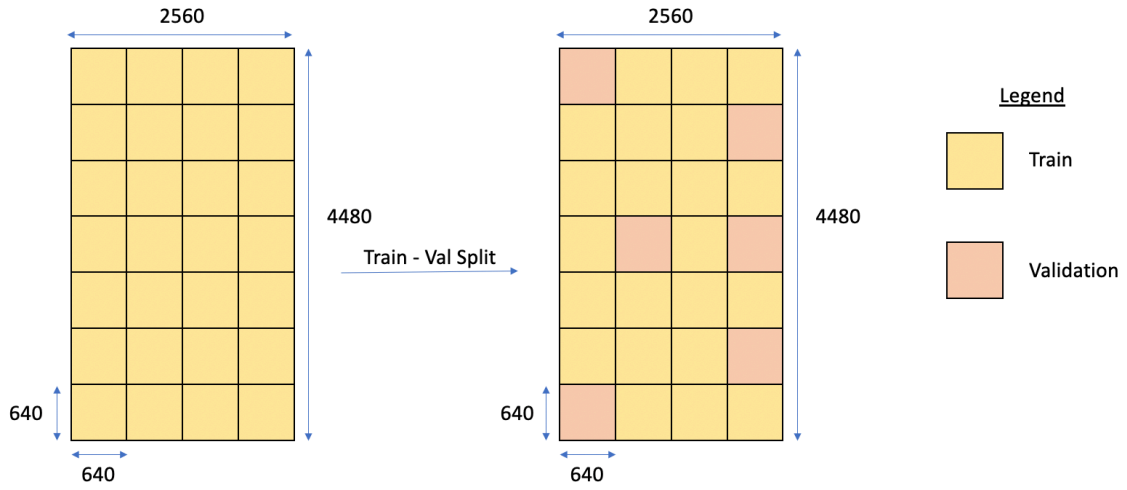



*Figure 4: Random splitting of train-validation split*

We would then verify the class breakdown in the training and validation sets, ensuring that the proportions of the labels are roughly similar. The individual breakdown of counts of labels for training set and validation set are shown in Table 6 and Table 7 respectively.

Training Set [224 Slices]

| Label Type | Overall Number of Labels |
|---|---|
| Fertilised Egg | 286 |
| Unfertilised Egg | 186 |
| Fish Larvae | 695 |
| Unidentifiable | 95 |

*Table 6: Overall counts of labels in the training set*

Validation Set [56 Slices]

| Label Type | Overall Number of Labels |
|---|---|
| Fertilised Egg | 62 |
| Unfertilised Egg | 41 |
| Fish Larvae | 184 |
| Unidentifiable | 26 |

*Table 7: Overall counts of labels in the validation set*

Once we have verified that our training and validation sets are ready, we will proceed with data augmentation of the training data in efforts of increasing our model's accuracy.

## 1.5. Data Augmentation

Noting that certain objects have been relabelled in our dataset when the updated SFA labels were given, we see that there is a larger proportion of Fish Larvae as opposed to Fertilized Eggs and Unfertilized Eggs. Thus, we performed data augmentation on the training data in hopes to increase the amount of training data, and tackle the imbalance of counts of the major three classes. This would imply that the validation set is untouched from the previous segment and still consists of 56 slices.

To do so, we first identify slices with higher proportions of Fertilized Eggs and Unfertilized Eggs. We would then increment copies of these slices into our training data and utilize YOLO's augmentation arguments in hyp.scratch to randomly translate, edit hue and contrast etc. This would allow a larger number of objects belonging in the Fertilized Eggs and Unfertilized Eggs classes to be trained and would decrease the number of iterations required for convergence.

We note that the counts of Fish Larvaes will increase a little as well, since these incremented image slices would contain a few Fish Larvaes. However, with a larger training dataset, each training epoch would take a longer time and hence may take slightly more time overall.

With a total of 296 slices, a slight increase from 224 originally, we are able to achieve a more balanced dataset. The final counts of the augmented training dataset is shown in Table 8 below.

| Label Type | Overall Number of Labels |
|---|---|
| Fertilised Egg | 952 |
| Unfertilised Egg | 360 |
| Fish Larvae | 953 |
| Unidentifiable | 137 |

*Table 8: Overall counts of labels in the augmented training set*

We see that the ratio of the major three classes is approximately 3:1:3, which is better than the original proportions listed in Table 5 as there is less skewness towards the Fish Larvae class. Using the augmented training set and the retained validation set, we will now proceed to modelling and tuning.

# 2. Modelling and Tuning

## 2.1. Phase I Modelling and Tuning (YOLOv3)

### 2.1.1. Baseline Model

We will be defining our baseline model by setting the following parameters: batch size 8, epoch 50. The hyp.scratch file will be used at its default state as well. Note that Auto-Anchor is enabled by default (as a parsable parameter for train.py). This deals with setting of anchor box sizes.

Another noteworthy decision is to utilize the default SGD optimizer over Adam, as utilizing Adam would require finer tuning of other hyperparameters (particularly learning rate, momentum, etc.) in the hyp.scratch file.

The results of the baseline model are shown in Figure 4 and Figure 5 below.

```
     Epoch   gpu_mem       box       obj       cls     total    labels  img_size
     49/49     7.67G   0.04068  0.008723   0.01044   0.05985         7       704: 100% 64/64 [02:17<00:00,  2.15s/it]
               Class      Images      Labels           P           R      mAP@.5  mAP@.5:.95: 100% 9/9 [00:07<00:00,  1.19it/s]
                 all         133         341       0.516       0.782        0.68       0.262
     Fertilised Egg          133         115       0.515       0.896       0.782       0.285
   Unfertilised Egg          133          60        0.61       0.917       0.875       0.348
         Fish Larvae         133         136       0.626       0.949       0.847       0.358
      Unidentifiable         133          30       0.314       0.367       0.215      0.0574
50 epochs completed in 2.067 hours.
```

*Figure 5: Results of Phase I baseline model*

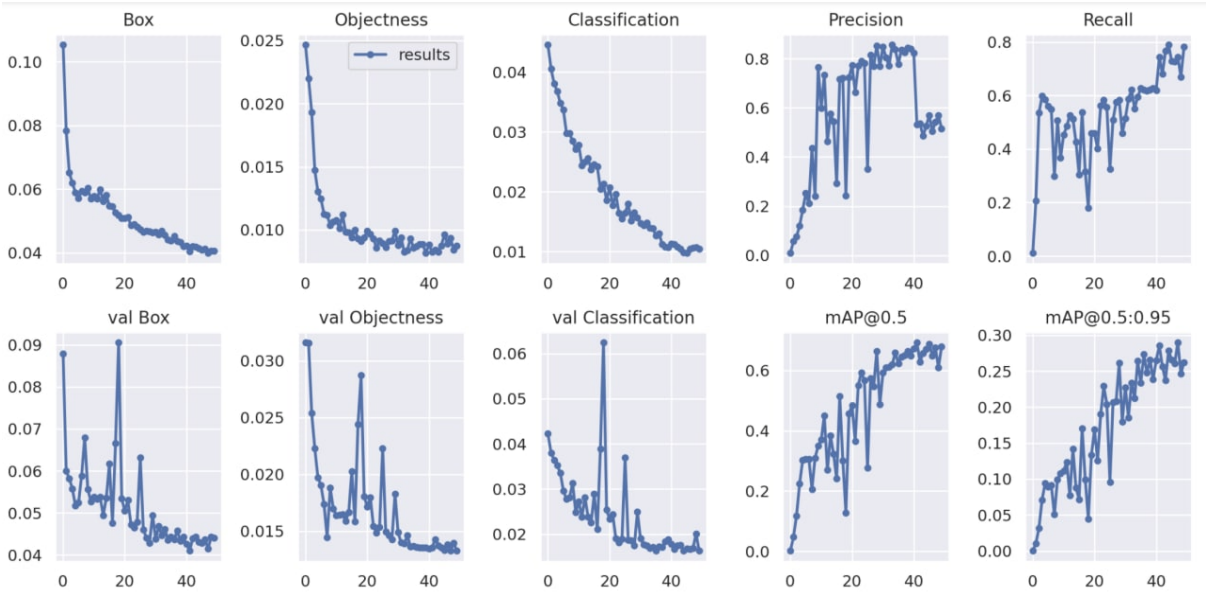



*Figure 6: Various plots of Phase I baseline model*

From our baseline model's result as shown above in Figure 5 and Figure 6, we realized that we do not have convergence in our loss plot - which signals to us to increase the number of epochs during training. As such, we increased our epoch from 50 to 300. Also, the presence of multiple large fluctuations meant that our batch size may be too small and hence, we decided to increase it. We choose to increase the batch size from 8 to 48 for now so as to speed up the time required for training too. With these new hyper-parameters, we trained our revised baseline model and the results are shown in Figure 7 below.

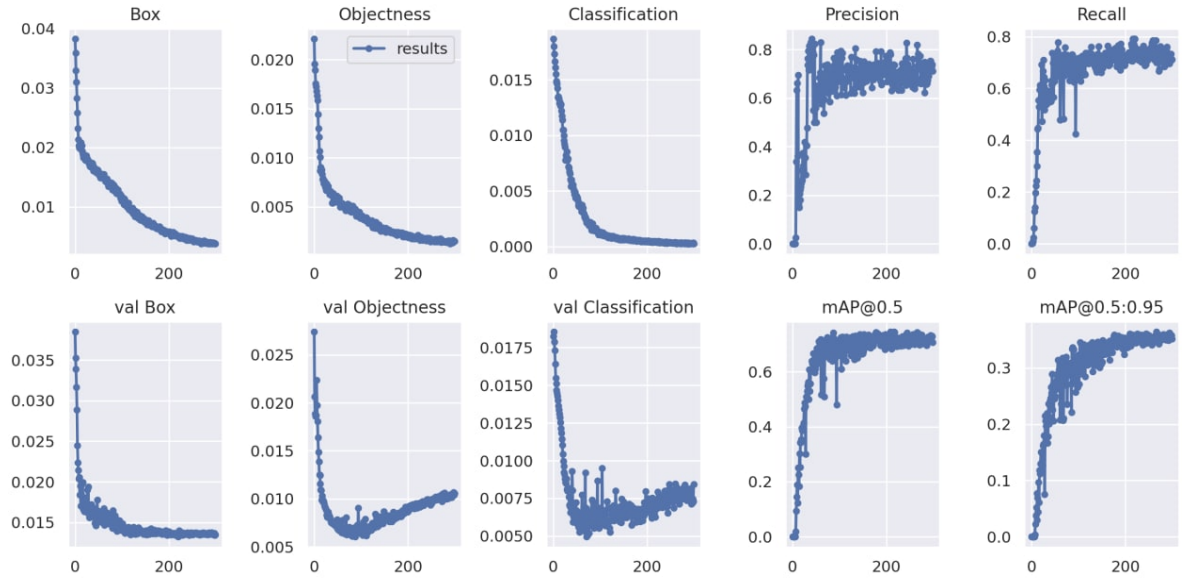



*Figure 7: Various plots of revised Phase I baseline model*

From the loss plots in Figure 7, we noticed that the loss stabilizes at around 120 epoch, and any further training would only lead to overfitting of model. As such, we decided to reduce the epoch to 120 and maintain the batch size of 48 to get ready for the next phase of hyper-parameter tuning using evolution.

### 2.1.2. Hyp.scratch Hyper-Parameters Tuning

Besides the parsable hyper-parameters like epochs and batch size, there are still many hyper-parameters that can be tuned which are found in the Hyp.scratch file. There is an option to perform automated hyper-parameter evolution, which is a method of hyper-parameter optimization using a generic algorithm for optimization. Evolution is performed about a base scenario which we seek to improve upon, and in our case, we initialized the hyper-parameters base scenario with the values in our revised baseline model of 120 epochs and batch size of 48.

The metric that we use to gauge the improvement of our model is mAP, and it is known as the 'Fitness' in YOLOv3. Fitness is the value we seek to maximize. By default, YOLOv3 defines the fitness function

as a weighted combination of metrics - with mAP@0.5 contributing 10% and mAP@0.5:0.95 contributing the remaining 90% of the weights, with Precision and Recall absent.

We performed evolution for 100 iterations and retrieved the best run among the 100 as our leading model. The results of the leading model and evolution are shown below in Figure 8 and Figure 9 respectively.

```
4m[1mautoanchor: [0mAnalyzing anchors... anchors/target = 2.40, Best Possible Recall (BPR) = 1.0000
        119.Class      Images      Labels           P           R      mAP@.5  mAP@.5:.95
              all         133         341       0.602        0.74       0.706       0.351
   Fertilised Egg         133         115       0.629       0.896       0.831       0.342
 Unfertilised Egg         133          60       0.782       0.897       0.928        0.52
      Fish Larvae         133         136       0.743       0.934       0.938       0.492
   Unidentifiable         133          30       0.253       0.233       0.128      0.0499
```

*Figure 8: Leading model from Phase I evolution*

```
lr0: 0.013
lrf: 0.262
momentum: 0.835
weight_decay: 0.0003
warmup_epochs: 1.55
warmup_momentum: 0.484
warmup_bias_lr: 0.0949
box: 0.0263
cls: 0.319
cls_pw: 0.933
obj: 1.65
obj_pw: 1.15
iou_t: 0.2
anchor_t: 2.08
fl_gamma: 0.0
hsv_h: 0.0146
hsv_s: 0.551
hsv_v: 0.232
degrees: 0.0
translate: 0.0
scale: 0.0
shear: 0.0
perspective: 0.0
flipud: 0.0
fliplr: 0.0
mosaic: 1.0
mixup: 0.0
```

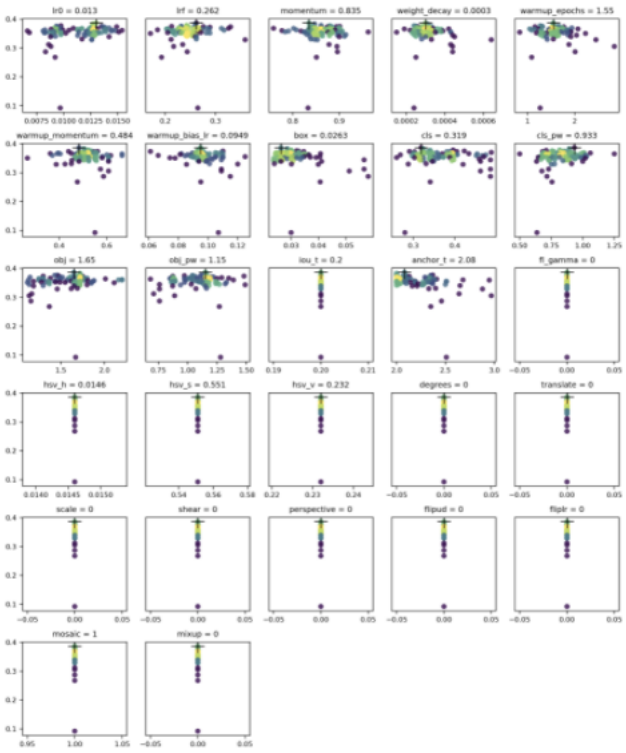


*Figure 9: Results of Phase I evolution*

We noticed that the parameter *Fl_gamma* does not change during evolution. As such, we tried to manually tune it but it did not show much improvement. Also, we noticed that the images are always of similar size and angle and hence we decided to not train on the scale, shear and perspective and setting them to 0.

### 2.1.3. Parsable Hyper-Parameters Tuning

With our leading model from evolution, we further fine-tune our model by varying the batch size whilst keeping the other hyper-parameters constant. We tested out batch sizes of 24 and 30 and the results are shown in Figure 10 and Figure 11 respectively below.

| 110.Class | Images | Labels | P | R | mAP@.5 | mAP@.5:.95 |
|---|---|---|---|---|---|---|
| all | 133 | 341 | 0.665 | 0.76 | 0.729 | 0.341 |
| Fertilised Egg | 133 | 115 | 0.69 | 0.922 | 0.821 | 0.332 |
| Unfertilised Egg | 133 | 60 | 0.775 | 0.917 | 0.871 | 0.464 |
| Fish Larvae | 133 | 136 | 0.813 | 0.934 | 0.924 | 0.431 |
| Unidentifiable | 133 | 30 | 0.38 | 0.267 | 0.301 | 0.139 |

*Figure 10: Results of Phase I leading model with batch size 24*

| 118.Class | Images | Labels | P | R | mAP@.5 | mAP@.5:.95 |
|---|---|---|---|---|---|---|
| all | 133 | 341 | 0.714 | 0.737 | 0.719 | 0.326 |
| Fertilised Egg | 133 | 115 | 0.71 | 0.878 | 0.794 | 0.299 |
| Unfertilised Egg | 133 | 60 | 0.772 | 0.85 | 0.857 | 0.428 |
| Fish Larvae | 133 | 136 | 0.842 | 0.956 | 0.911 | 0.439 |
| Unidentifiable | 133 | 30 | 0.532 | 0.265 | 0.312 | 0.141 |

*Figure 11: Results of Phase I leading model with batch size 30*

Our model with a batch size of 24 did better in average mAP than batch size 30, both for 3 main classes and overall. As such we finalized our leading model to have the hyper-parameters from evolution with 120 epochs and a batch size of 24.

On top of that, in an attempt to correct for the noticeable class imbalance as shown in Table 5, we attempted to use label smoothing to calibrate the model for any overconfidence when learning/classifying classes that are most frequent. However, while we note that model training tends to converge faster compared to models without label smoothing enabled, the final mAP at the end of training tends to be comparable to the latter thus offering no improvement in prediction performance. As such, we have decided to drop label smoothing from the subsequent models.

### 2.1.4. Phase I Ensemble (Bagging)

Empirically, Ensembling methods have been employed to improve prediction performance by reducing model variance. In our experiment, we have trained 10 instances of YOLOv3 object detectors initialised with different sets of hyperparameters but with the same pretrained weights. These 10 sets of weights were randomly chosen from the 100 iterations of the Evolution algorithm from Section 2.1.2. The training data was generated from bootstrap sampling.

| Class | Images | Labels | P | R | mAP@.5 | mAP@.5:.95 |
|---|---|---|---|---|---|---|
| all | 133 | 341 | 0.627 | 0.77 | 0.703 | 0.307 |
| Fertilised Egg | 133 | 115 | 0.68 | 0.922 | 0.782 | 0.309 |
| Unfertilised Egg | 133 | 60 | 0.734 | 0.967 | 0.885 | 0.487 |
| Fish Larvae | 133 | 136 | 0.712 | 0.926 | 0.891 | 0.376 |
| Unidentifiable | 133 | 30 | 0.384 | 0.267 | 0.256 | 0.0566 |

*Figure 12: Summary of Phase I ensemble performance on validation set*

The outputs of the 10 models were automatically aggregated within the test.py file and a final inference summary was produced as shown in Figure 12.

### 2.1.5. Phase I Final Model

Comparing the results in Figure 12 with Figure 10, we can clearly see a reduction in performance of the ensembled model. Understandably, due to the high correlation between each of the 10 models used in ensembling, the total variance reduction did little to compensate for the increased bias in the ensembled model where weaker learners were used. This resulted in a poorer prediction performance compared to the single best model. As such, we default to the single best model achieving mAP@0.5 of 0.872 for the 3 main classes as shown in Figure 10.

## 2.2. Phase II Modelling and Tuning (YOLOv5)

In the second phase of training, we looked towards training with a different architecture with YOLOv5 as it was supposedly more accurate and faster than YOLOv3.

### 2.2.1. Baseline Model

Similar to our Phase I training, we begin with a baseline model with YOLOv5 to try out. We used YOLOv5s weights, along with image size 640, batch 8 and epoch 50 using SGD. The results of our baseline model is shown in Figure 13 below.

```
Model Summary: 213 layers, 7020913 parameters, 0 gradients, 15.8 GFLOPs
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95:
                 all         56        313      0.898      0.583      0.651      0.298
      Fertilised Egg         56         62      0.904      0.757      0.843      0.384
    Unfertilised Egg         56         41      0.847       0.78      0.844      0.376
          Fish Larvae         56        184      0.839      0.793      0.857      0.402
        Unidentifiable         56         26          1          0     0.0604     0.0283
```

*Figure 13: Results of Phase II baseline model*

### 2.2.2. Training Flow & Hyper-parameter Tuning

We experimented with a total of 4 different weights: YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x, along with either SGD or Adam optimizer as shown in Table 9 below. The different experiments are shown below in Table 9. In all the experiments that we ran, we used the default evolution settings of 300 iterations.

| Weights | Image Size | Batch Size | Epochs | Optimizer | mAP |
|---|---|---|---|---|---|
| YOLOv5s | 640 | 30 | 150 | SGD | 0.71606 |
| YOLOv5m | 640 | 30 | 150 | SGD | 0.71772 |
| YOLOv5l | 640 | 30 | 150 | SGD | 0.74272 |
| YOLOv5x | 640 | 30 | 150 | SGD | 0.7116* |
| YOLOv5s | 640 | 30 | 150 | Adam | 0.7406 |

| | | | | | |
|---|---|---|---|---|---|
| YOLOv5m | 640 | 30 | 150 | Adam | 0.7410 |
| YOLOv5l | 640 | 30 | 150 | Adam | 0.7424 |
| YOLOv5x | 640 | 30 | 150 | Adam | 0.7347 |

*Table 9: Various experiments of different YOLOv5 weights*

The experiment with YOLOv5x with mAP 0.7116 was truncated early at the 200$^{th}$ generation in evolution due to max wall time. From Table 9, we select the model using YOLOv5l (SGD) as it has the highest mAP, and this is to be expected as YOLOv5l is the densest model with the most number of weights. This however comes with a trade-off for time as the YOLOv5l model takes up 4x as much time as compared to the smaller YOLOv5s model.

Considering these trade-offs, we decided to continue fine-tuning the 2 models (with YOLOv5l and YOLOv5s weights) from the results of the evolution to accommodate users with different priorities. The model with YOLOv5l weights (Model L) would be our main model as it has the highest mAP score and will be able to provide the most accurate counts/predictions. On the other hand, for users with a time constraint, can use the model with YOLOv5s weights (Model S), which has a slightly lower mAP (2% drop), but is able to make predictions almost 4 times faster. After trying out a range of different batch sizes, we finalized both Model L and Model S with results shown in Figure 14 and Figure 15 respectively below.

Model L:

- Trained using the optimal hyperparameters of YOLOv5l
- Best batch size 14 and epoch 188
- mAP of 0.743
- ~37 seconds for full prediction on a single image

| Class | Images | Labels | P | R | mAP@.5 | mAP@.5:.95 |
|---|---|---|---|---|---|---|
| all | 56 | 313 | 0.727 | 0.727 | 0.743 | 0.395 |
| Fertilised Egg | 56 | 62 | 0.723 | 0.774 | 0.839 | 0.432 |
| Unfertilised Egg | 56 | 41 | 0.948 | 0.893 | 0.962 | 0.57 |
| Fish Larvae | 56 | 184 | 0.849 | 0.973 | 0.908 | 0.474 |
| Unidentifiable | 56 | 26 | 0.389 | 0.269 | 0.264 | 0.103 |

*Figure 14: Results of Model L*

Model S

- Trained using optimal hyperparameters of YOLOv5s
- Best batch size 8 and epoch 181
- mAP of 0.722
- ~10 second for full prediction on a single image

| Class | Images | Labels | P | R | mAP@.5 | mAP@.5:.95 |
|---|---|---|---|---|---|---|
| all | 56 | 313 | 0.724 | 0.711 | 0.722 | 0.382 |
| Fertilised Egg | 56 | 62 | 0.839 | 0.774 | 0.83 | 0.432 |
| Unfertilised Egg | 56 | 41 | 0.905 | 0.927 | 0.964 | 0.559 |
| Fish Larvae | 56 | 184 | 0.848 | 0.913 | 0.907 | 0.459 |
| Unidentifiable | 56 | 26 | 0.305 | 0.231 | 0.187 | 0.0769 |

*Figure 15: Results of Model S*

### 2.2.3. Phase II Ensemble

Similar to our ensemble method in Phase I (Section 2.1.4), we performed BAGGING using 10 separately trained models, each initialized with hyperparameters randomly sampled from an evolution run. However for Phase II of our ensembling model, we experimented with different aggregation techniques, namely: max, mean, and NMS.

- In summary, given a set of predictions for an object (output by the different learners in the ensemble), max ensemble takes the maximum dimension of the bounding boxes together with the maximum confidence scores for all classes. The output is then passed into NMS for further processing.
- Mean ensemble takes the mean of all predicted bounding box dimensions, as well the mean confidence for all classes. The output is then passed into NMS for further processing.
- NMS ensembling simply performs NMS over the predictions.

| Class | Images | Labels | P | R | mAP@.5 | mAP@.5:.95: |
|---|---|---|---|---|---|---|
| all | 56 | 313 | 0.732 | 0.728 | 0.754 | 0.366 |
| Fertilised Egg | 56 | 62 | 0.672 | 0.79 | 0.871 | 0.425 |
| Unfertilised Egg | 56 | 41 | 0.901 | 0.878 | 0.923 | 0.45 |
| Fish Larvae | 56 | 184 | 0.821 | 0.935 | 0.933 | 0.451 |
| Unidentifiable | 56 | 26 | 0.533 | 0.308 | 0.29 | 0.137 |

*Figure 16: Results of ensemble model evaluation on validation set*

From our experiments, mean ensembling appears to perform the best with a top-3 class mAP of 0.909 as shown in Figure 16 above.

### 2.2.4. Phase II Final Models

We decided to use both models in our solution to provide our users a choice. The parameters and results of both models are shown below in Table 10, using the Hyp.scratch file as shown in Figure 17.

| | **Model L** | **Model S** |
|---|---|---|
| Weights | YOLOv5l weights | YOLOv5s weights |
| Epoch | 188 | 181 |
| Batch Size | 14 | 8 |
| mAP | 0.743 | 0.722 |
| Time taken for prediction of 1 image | ~ 37 seconds | ~ 10 seconds |

*Table 10: Parameters and results of Model L and Model S*

```
lr0: 0.0112
lrf: 0.13431
momentum: 0.95777
weight_decay: 0.00042
warmup_epochs: 3.86085
warmup_momentum: 0.72888
warmup_bias_lr: 0.1599
box: 0.02928
cls: 0.56774
cls_pw: 0.7867
obj: 1.21781
obj_pw: 0.96222
iou_t: 0.2
anchor_t: 3.03373
fl_gamma: 0.0
hsv_h: 0.0
hsv_s: 0.0
hsv_v: 0.0
degrees: 0.4
translate: 0.12363
scale: 0.0
shear: 0.0
perspective: 0.0
flipud: 0.4
fliplr: 0.4
mosaic: 0.92469
mixup: 0.0
copy_paste: 0.0
anchors: 3.8006
```

```
lr0: 0.0131
lrf: 0.10497
momentum: 0.95434
weight_decay: 0.00054
warmup_epochs: 3.56534
warmup_momentum: 0.95
warmup_bias_lr: 0.06869
box: 0.06644
cls: 0.35134
cls_pw: 1.08745
obj: 0.65214
obj_pw: 1.20571
iou_t: 0.2
anchor_t: 4.27132
fl_gamma: 0.0
hsv_h: 0.0
hsv_s: 0.0
hsv_v: 0.0
degrees: 0.4
translate: 0.0935
scale: 0.0
shear: 0.0
perspective: 0.0
flipud: 0.4
fliplr: 0.4
mosaic: 0.88111
mixup: 0.0
copy_paste: 0.0
anchors: 3.90463
```

*Figure 17: Hyp.scratch file of Model L (left) and Model L (right)*

# 3. Testing

## 3.1. Masking

Motivation: While sieving through the predicted images in our validation set, we note that sometimes objects outside the area of interest (i.e. outside the water blob or around the edges of the petri dish) receive a predicted bounding box as well. Therefore, for prediction of test images, we attempted to make use of masking to blacken regions not in the water blob outline. This will ensure that all predictions on test images only occur within the area of interest.

### 3.1.1. Phase I Masking


*Figure 18: Before and after masking of images*

However, to implement masking, there are a few assumptions that we have made:

- Image are fairly consistent in intensity, resolution
- Area of interest is located in the centre of image

Original Image

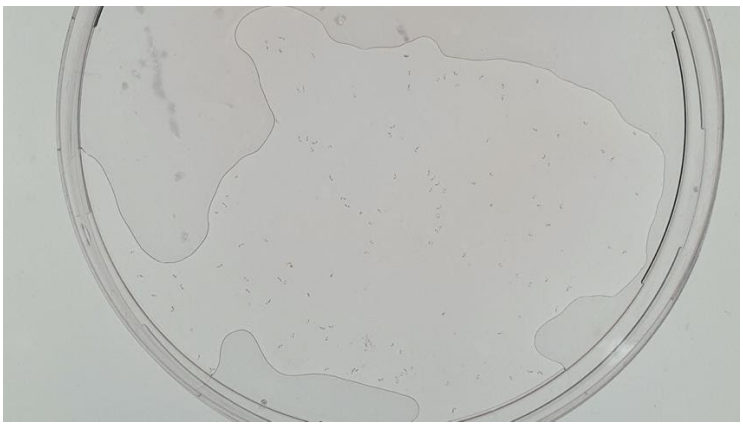


*Figure 19: Example of original image*

Processing Steps

Histogram stretching

- Histogram stretching is a combination of adjusting brightness and contrast so as the spread pixel values over the full range (0-255)
- This makes the features in the image more obvious as the changes in shades will be more distinct.
- From the histogram of intensity against pixel count in Figure 20, we see a majority of the image lies within a very small range. Thus, histogram stretching is very effective.

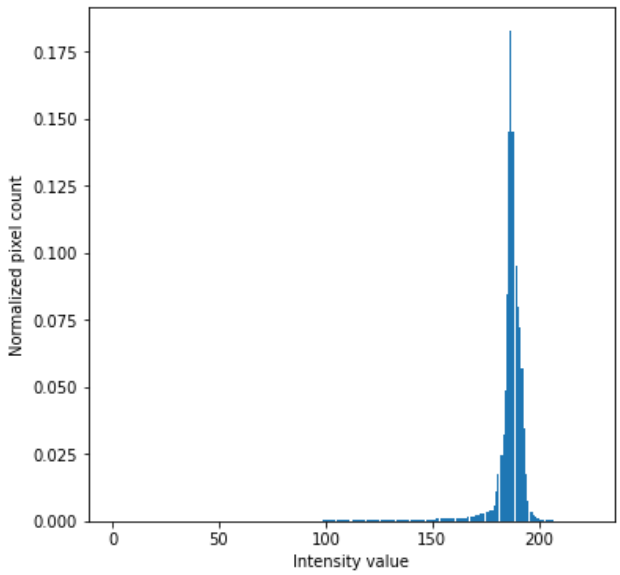



*Figure 20: Histogram of intensity against pixel count*

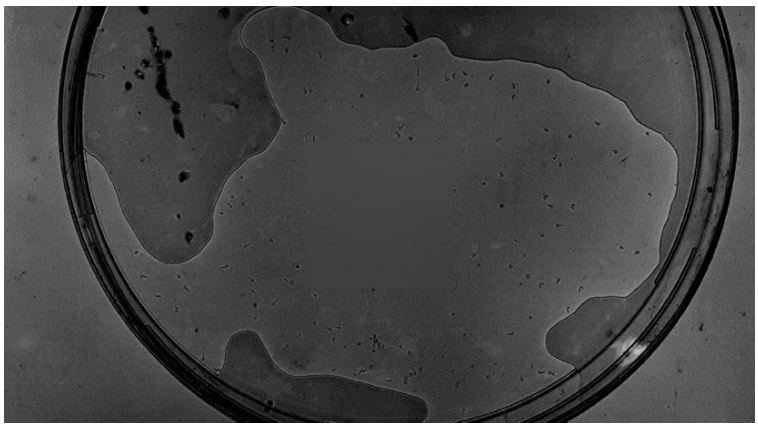


*Figure 21: Applying histogram stretching*

Detect edges with Canny edge detector

- Canny detector identifies edges by finding areas with sudden change of intensity
- Before applying this process, we blurred the centre of the image to reduce the likelihood of an edge case that will be mentioned later.

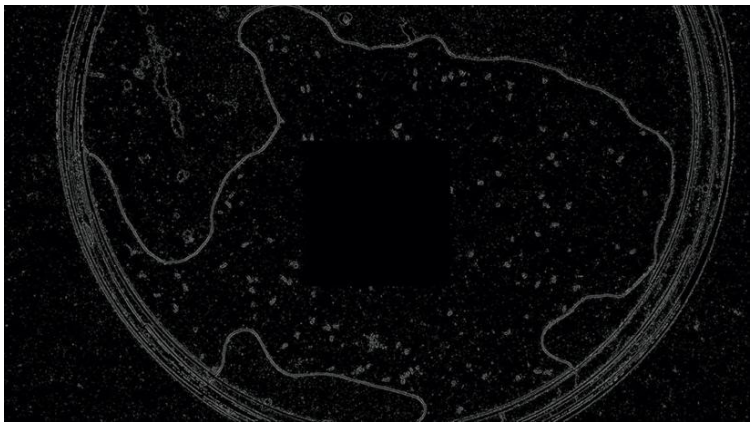


*Figure 22: Canny edge detector*

Decrease resolution link up boundary + flood fill from centre

- Although the Canny edge detector is pretty effective at finding edges. There tend to still be small pixel gaps between the borders. This prevent us from flood filling to be of any use as it will overflow outside of the boundary
- To handle this, we decrease resolution of the image by down sampling with max pooling in order to keep the border pixels.
- After that, we then flood fill from the centre to and keep the flooded portion identified as the area of interest as shown in Figure 23 below.

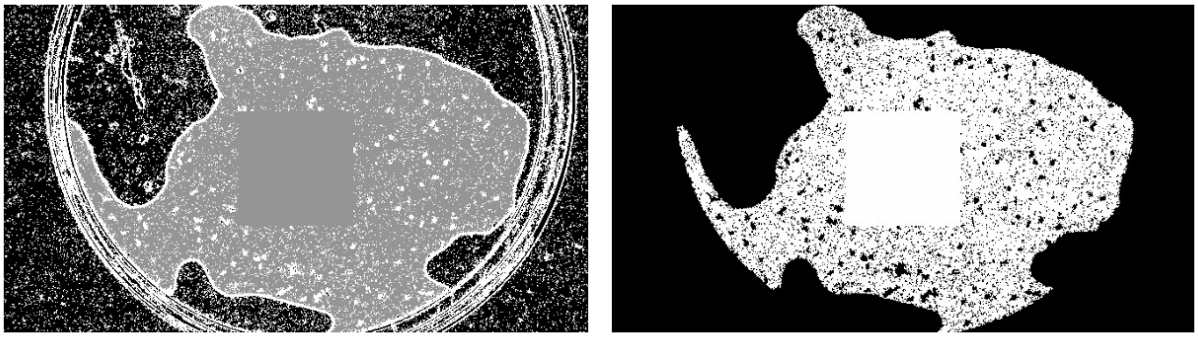


*Figure 23: Decrease resolution link up boundary and flood fill from centre*

Perform image closing (the process of dilation + erosion)

- After we have obtained the area of interest from the flood fill. There are still obviously many gaps. This is due to the noisy contours from the small objects.
- We fill up these gaps using the process of image closing. This process consists of 2 main operations, image dilation and image erosion.
- Image dilation increases the boundaries of any white pixels while erosion decreases the boundaries of any white pixels.
- By performing these back to back, it essentially removes holes from within the area of interest.

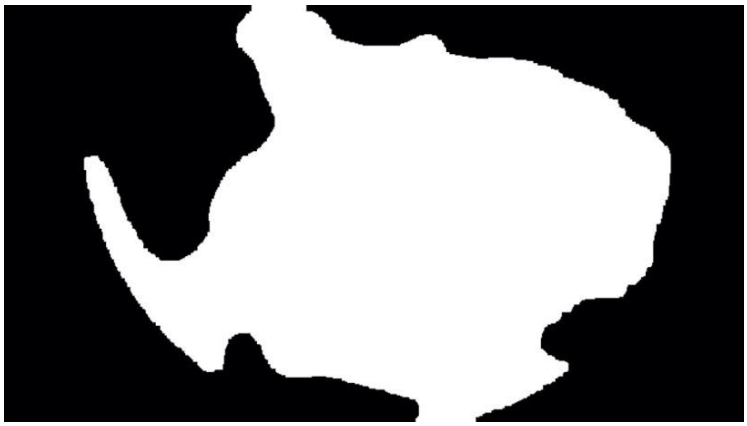


*Figure 24: Performing image closing*

Apply mask on image

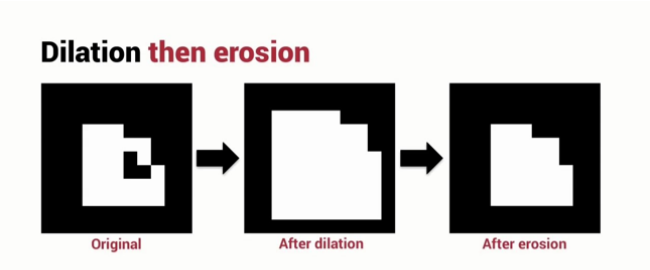
- Finally after obtaining the mask, we simply mask it over the original image, only keeping the area of interest.

*Figure 25: Applying mask on image*

**Limitations**

Edge detection not always perfect

- Any gaps in border will prevent total masking
- However, since we flood fill from centre, in worst case, (will mask less rather than into mask more into area of interest)




Edge case, if larva line up back to back forming a complete border to block an entire section, may result in imperfect masking

- We have minimized chance of this happening with blurring and image closing

### 3.1.2. Phase II Masking (Improvements)

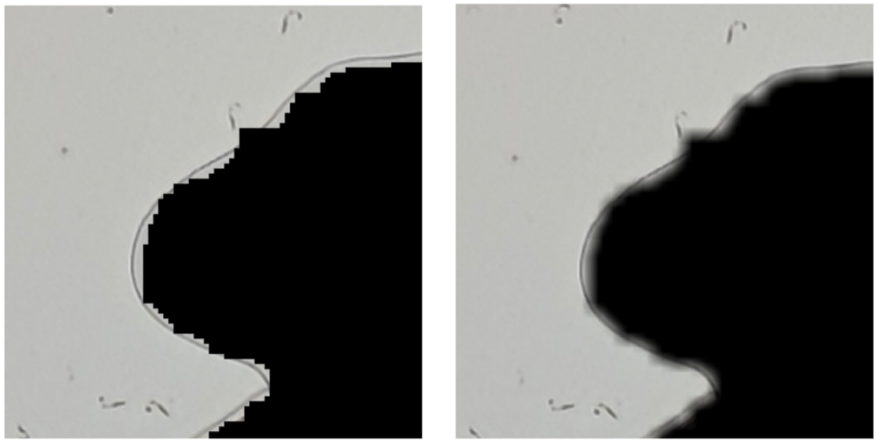


*Figure 26: Phase I masking (left) vs Phase II masking (right)*

Our masking pre-processing is mainly the same as in phase I. However, we noticed an obvious flaw, the edges of the masking are very uneven and do not lie along the area of interest nicely. We also noticed an additional problem where in certain rare cases, some of these jagged edges were getting picked up by the model as objects.

In phase II, we made some improvement to masking. By adding some degree of blurring to the mask, and better fine-tuning of the image closing step, we were able to obtain a mask which is smoother and better aligned with the area of interest.

## 3.2. Testing

### 3.2.1. Phase I Testing (Slicing with Overlap)

In our previous assessment, we trialled testing our model on a full test image and noticed that there is a decrease in performance (mainly on object detection). This is possibly due to inputting a large test image (if left in its original state, dimension would be (4032, 2268)). This motivates us to slice our images for prediction, then merge back the slices and their corresponding bounding boxes. Similar to how we slice images for the training data, we would slice the incoming test image into parts, so they would have identical dimensions as the ones used for training.

To account for the objects lost during slicing, we propose slicing the test images with overlap, having leeway on the edges of each part. After predicting bounding boxes for these slices with overlap, we would remap and merge the coordinates back onto the original test image. Finally, we plan to refine the overlapping bounding boxes using Non Max Suppression (NMS). The image below summarises the process:

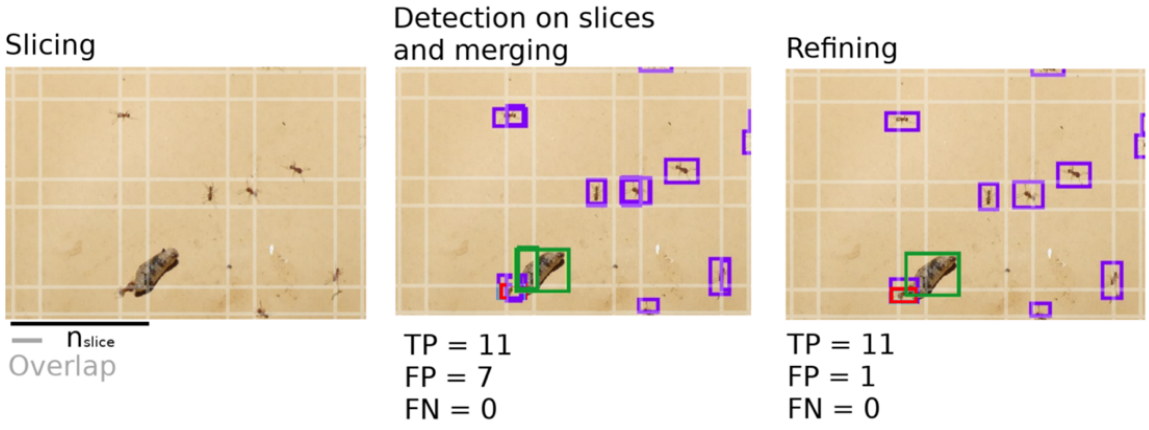



*Figure 27: Slicing with overlap, detection and refining process*

When attempting the whole process, we verified that predicting the sliced test images performs much better than parsing the whole test image without slicing.


*Figure 28: Prediction on sliced test image (left) vs full sized test image (right)*

This can be illustrated by the two images above, whereby the left image is able to correctly identify and predict more objects than the prediction shown on the right. However, this slight decrease in performance when predicting full-sized test images can be attributed to architectural flaws in YOLOv3.

### 3.2.2. Phase II Testing (Full Image)

As mentioned in Section 2.2, we have switched from using YOLOv3 to YOLOv5. With sufficient model tuning and training, we have rectified this issue and are now accurately able to predict full-sized images, regardless of having sliced images as training data. Therefore, we are no longer slicing with overlap and redrawing the bounding box coordinates for test predictions, and have decided to simply predict on the full image using YOLOv5.

This is to minimize errors during the refining phase, as introducing a second layer of amending double counting could lead to multiple boxes drawn. To prevent these edge cases, predicting on the full image will be a better option. We note that the arguments for Non Max Suppression (NMS) required some fine-tuning since we will rely on this single layer to remove overlapping bounding boxes during prediction. In detail, we have increased the *confidence threshold* to 0.45, decreased the *IOU threshold* to 0.15 and increased the *number of maximum detections* to 1000 from the default 300, since most images have more than 300 foreign objects in them each. Fine tuning the confidence threshold was

based on two factors: comparison with the true counts attained and the statistical behaviour of the confidence scores across all predictions.

| Min | 25th Percentile | Mean | 75th Percentile | Max |
|---|---|---|---|---|
| 0.455 | 0.789 | 0.814 | 0.860 | 0.953 |

*Table 11: Percentile Statistics of the Confidence Scores at Confidence Threshold = 0.45*

By increasing the confidence threshold, we reduce the predictions with low confidence that could occur. From Table 11 above, we see that a large majority (approximately 75%) of the predictions have confidence scores near 0.80. Decreasing the IOU threshold meant a stricter amount of overlap that could occur on the bounding boxes.

Finally, predicting on the full image directly would save on predicting time and computational space, as we no longer need to rescale the coordinates for each slice label and redraw them on the original test image. Therefore, we would not require storage of these original coordinates and performing mathematical rescaling on them to obtain a new set of coordinates, increasing the speed of our predictions without losing accuracy.

# 4. User Interface (UI) of Webpage

To productionize our prediction model, we have developed a user-friendly front-end UI for our users to easily interact and use our model.

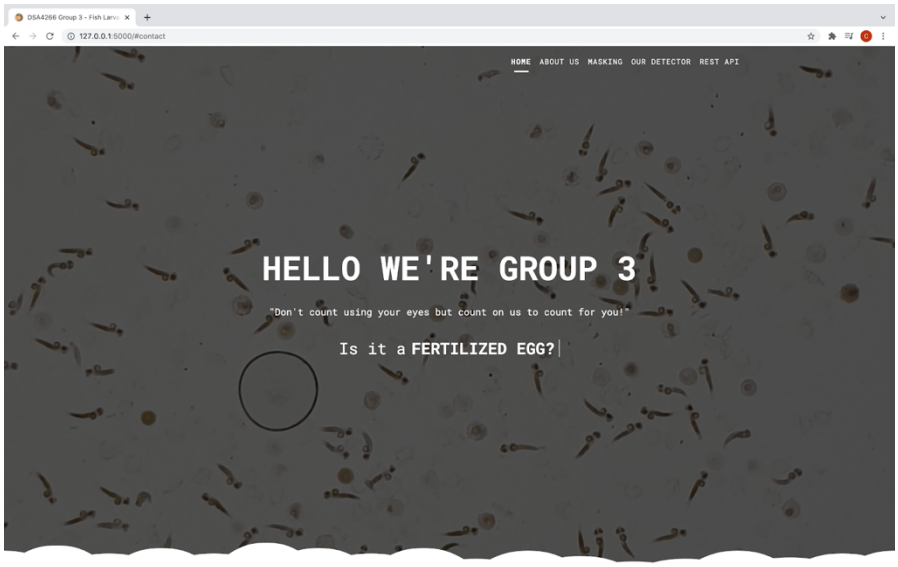



*Figure 29: Display of UI of webpage*

## 4.1. Detector

Our detector (Figure 30) is based on the final model from our project which is the YOLOv5 model as described in Section 2.2.4.

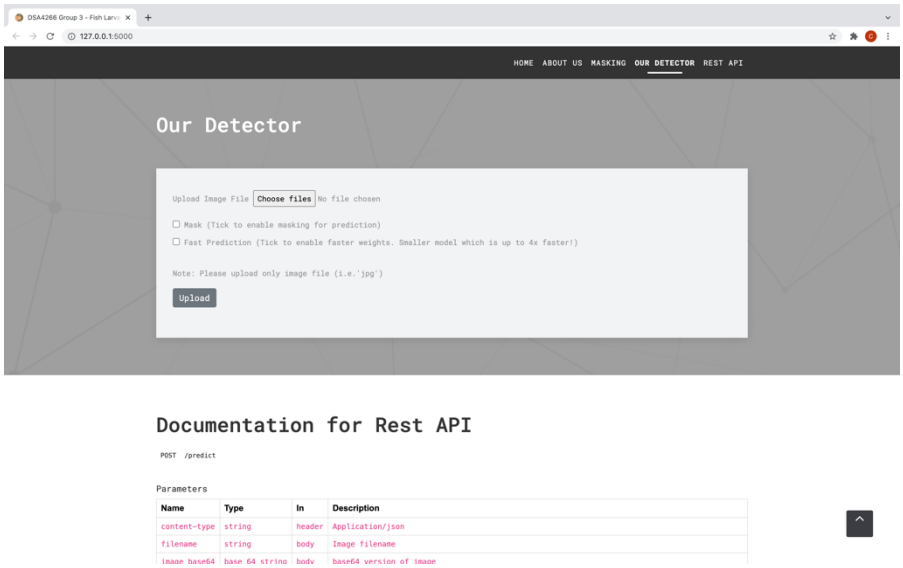



*Figure 30: Detector*

A pop-up will appear whenever a user uploads into our detector. If the file uploaded is not an image, a red pop-up will appear, informing the user to only upload image files (Figure 31). If the file uploaded is an image, a green pop-up will appear - informing the user that the upload is successful and to wait to be redirected to download the outputs (Figure 32). Once the inference is completed, the user will be redirected to the download page (Figure 33).

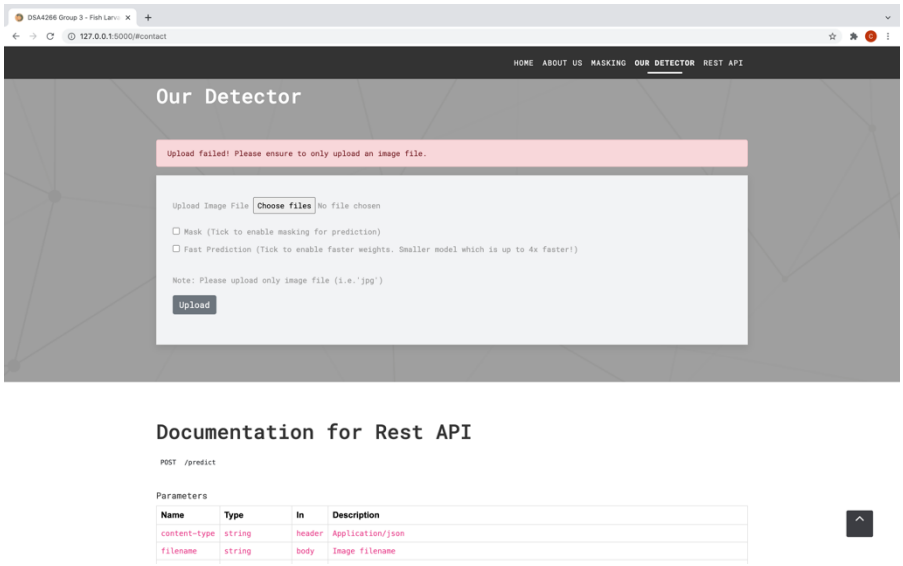



*Figure 31: Unsuccessful upload into detector on the webpage*

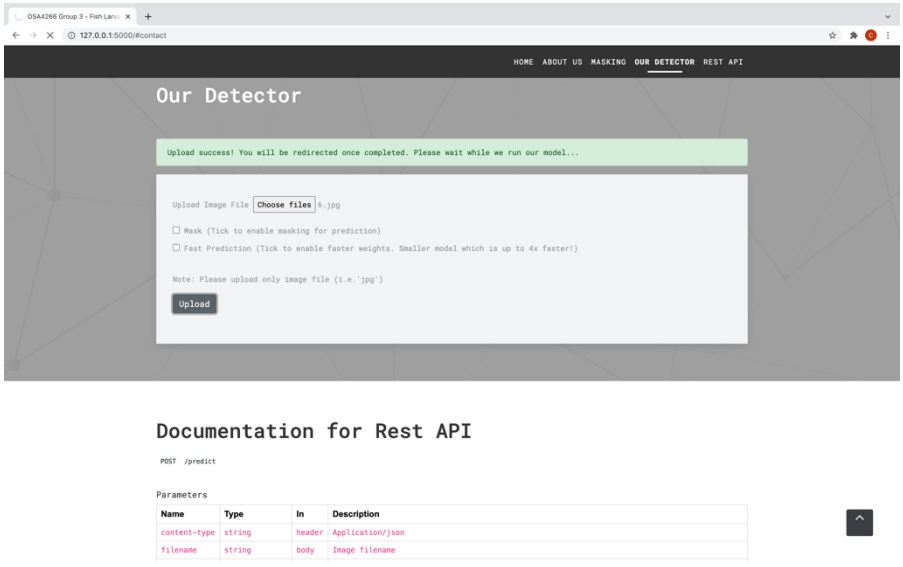



*Figure 32: Successful upload into detector on the webpage*

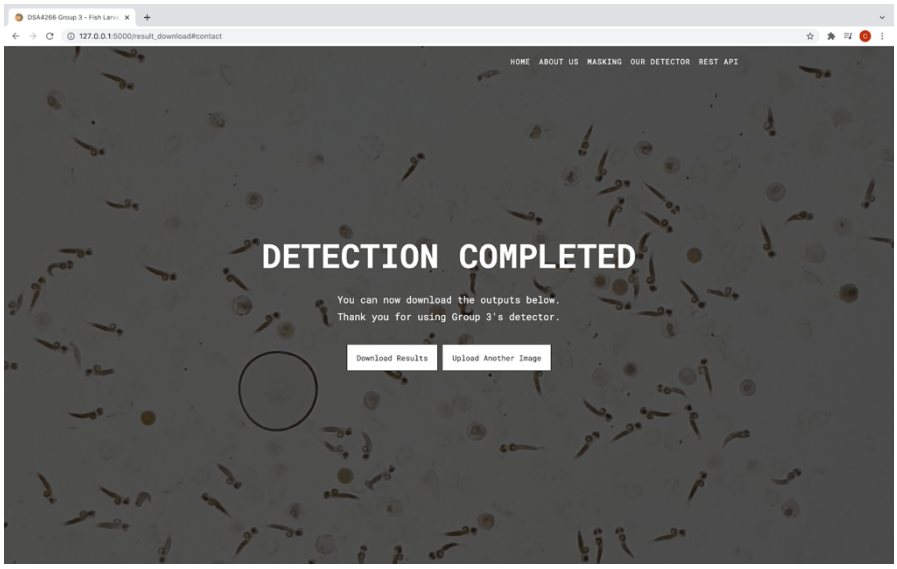



*Figure 33: Download page on the webpage*

## 4.2. Additional Features

Apart from the main function of detecting, predicting and accurately classifying the images of the petri-dish, we have included additional features to enhance the user's experience.

### 4.2.1. Preview of Masking

We included a preview function where users can upload an image to instantly view the effect of masking as shown in Figure 34 and Figure 35 below.

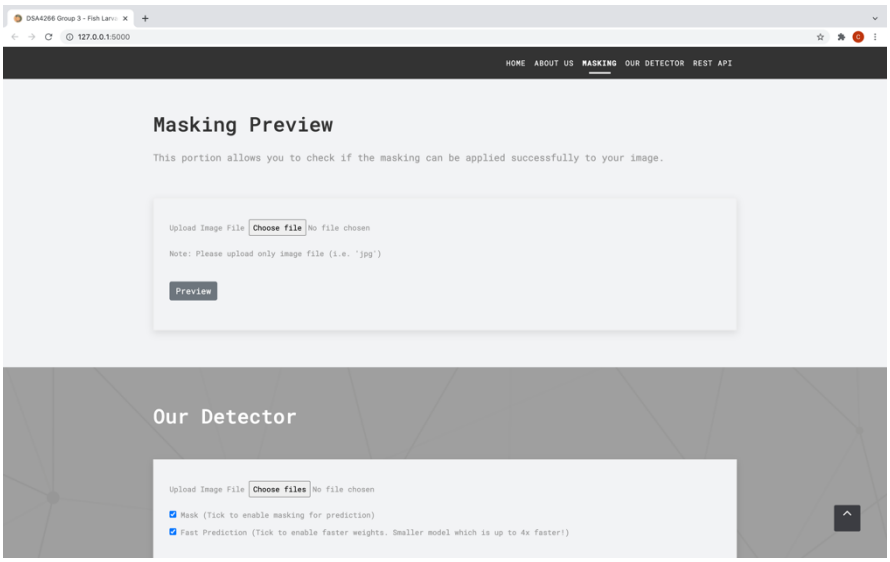



*Figure 34: Preview of Masking of UI on the webpage*

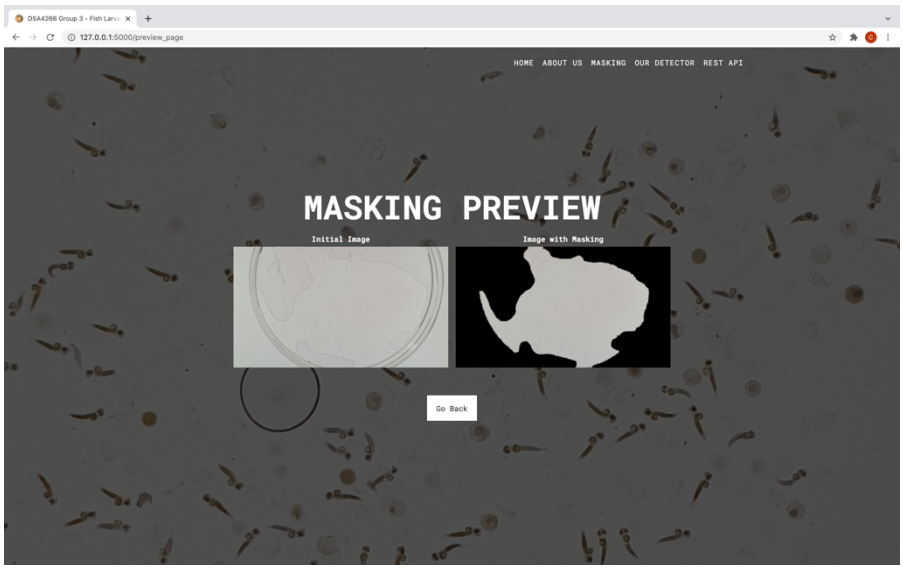



*Figure 35: Effect of Masking of UI on the webpage*

### 4.2.2. Option for Masking/Faster Weights

We have also provided an option for users to allow/disallow masking upon prediction, and use faster weights as shown in Figure 36. The idea to allow more flexibility for our users can also be extended to other parameters that can affect the model.

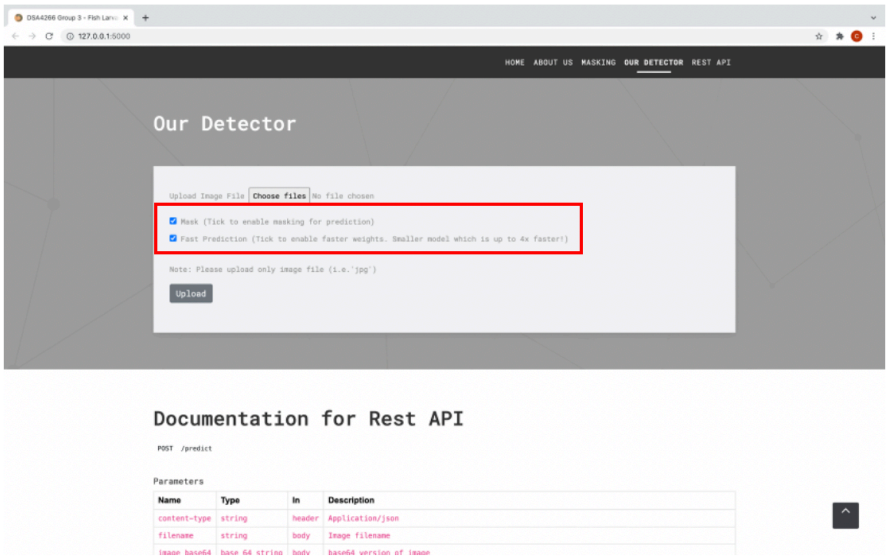



*Figure 36: Option of Mask at the Detector of UI on the webpage*

## 4.3. Rest API

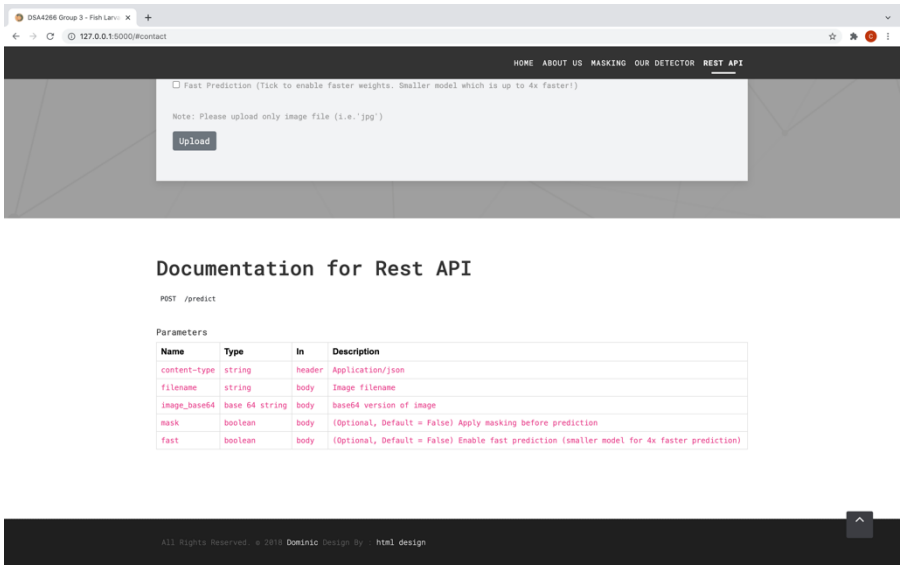



*Figure 37: Rest API*

In order for users to automate the process, we have also provided a REST API to run our prediction model. The user would simply have to send a POST request to /predict with following parameters as stated in Table 12 below.

| POST /predict | | | |
|---|---|---|---|
| **Name** | **Type** | **In** | **Description** |
| content-type | String | header | application/json |
| filename | String | body | Image filename |
| image_base64 | Base 64 string | body | Base64 version of image |
| mask | Boolean | body | (Optional: default is True) Apply masking to image before prediction |

*Table 12: Rest API*

**Response:**

If completed successfully, the user should receive a status 200 response with the following json as output.


```
{
    "filename": "1.jpg",
    "image": "/9j/4AAQSkZJRgABAQAAAQABAAD/ ... RjG3v3pkQBUn0p6qCTntQB//9k=",
    "predictions": [
        {
            "predicted_class": 2,
            "confidence": 0.943808,
            "bounding_box": [
                0.433160007,
                0.773327,
                0.00434,
                0.004401
            ]
        },
        {
            "predicted_class": 3,
            "confidence": 0.932472,
            "bounding_box": [
                0.2053570002,
                0.694982,
                0.004216,
                0.003081
            ]
        }
    ]
}
```


*Figure 38: Correct response as output*

If any error occurs such as:

- Invalid json input
- Missing fields
- Invalid base64 image

The user will receive a status 400 response with the following json, specifying the type of error and a brief description.


```
{
    "error": {
        "type": "TypeError",
        "message": "Content type is not json"
    }
}
```


*Figure 39: Wrong response as output*

# 5. Future Works (Scalability)

In terms of scalability, we had 2 main goals in mind: 1) model extensibility and 2) ease of training pipeline in the future. With that in mind, we introduce 3 ways to accomplish this:

## 5.1. Auto-Balancing of New Dataset

| Label Type | Overall Number of Labels |
|---|---|
| Fertilised Egg | 348 |
| Unfertilised Egg | 227 |
| Fish Larvae | 879 |
| Unidentifiable | 121 |

*Table 13: Class counts before balancing*

| Label Type | Overall Number of Labels |
|---|---|
| Fertilised Egg | 952 |
| Unfertilised Egg | 360 |
| Fish Larvae | 953 |
| Unidentifiable | 137 |

*Table 14: Class counts after balancing*

It is generally expected that any new incoming dataset would be highly skewed. This would present a problem where the model would predict the over-represented class too confidently; and underpredict the under-represented classes.

To this end, our data augmentation technique allows for automatic resampling of images that contain the under-represented classes. Thus, calling for a simpler, more scalable pipeline for future training.

## 5.2. Transfer Learning for Efficient Training of New Images

```
127     # Freeze
128     freeze = [f'model.{x}.' for x in range(freeze)]  # layers to freeze
129     for k, v in model.named_parameters():
130         v.requires_grad = True  # train all layers
131         if any(x in k for x in freeze):
132             print(f'freezing {k}')
133             v.requires_grad = False
```

*Figure 40: Freezing for Transfer Learning*

Transfer learning has been shown to produce fast training times whilst preserving model accuracy. Essentially, the upstream feature extraction layers of an initially-trained model are first frozen before training on a new sample of fish images. The downstream layers are then allowed to learn the variations in the new sample of fish images and hence, the model "updates" itself with every new batch of images.

This technique would allow us to significantly reduce training times. Nonetheless, while it is recognized that training solely on new batches of fish images (the old fish images are not used in the update process) would result in model drift, we can monitor such drift and periodically unfreeze the upstream layers to perform a full retraining of the model.

## 5.3. Speed Accuracy Trade-off

We have shown - as a proof of concept - that ensembling improves model performance allows us to achieve the best predictive performance of 0.909 mAP for the 3 main classes. While we were constrained to single-model solutions for this assignment, we can extend our solution to include ensembling in the future with better hardware (e.g. GPUs for inference). This would allow better performance when large datasets are tested, overall ensuring that our solution is able to output accurate results with sufficient resources.

# 6. Conclusion

In conclusion, the main accuracy of our model ultimately depends on the true labels on the training data. With regards to data pre-processing and model architecture, after experimenting with both YOLOv3 and YOLOv5, we have chosen to utilize YOLOv5 to produce the final predicted images as it provides relatively better accuracy for the major three classes, and takes lesser amount of time to run the inference on the same image. As we conclude our report, we would like to thank Assistant Professor Lim Chingway for moderating this module DSA4266, our Group 3's mentor Yadong, as well as all of the GovTech team.

# References

Object detection algorithm for high resolution images based on convolutional neural network and multiscale processing, http://ceur-ws.org/Vol-2864/paper12.pdf

ACF Based Region Proposal Extraction for YOLOv3 Network Towards High-Performance Cyclist Detection in High Resolution Images, https://www.researchgate.net/publication/333767339_ACF_Based_Region_Proposal_Extraction_for_YOLOv3_Network_Towards_High-Performance_Cyclist_Detection_in_High_Resolution_Images

Insect interaction analysis based on object detection and CNN, https://hal.archives-ouvertes.fr/hal-02361210/document