Wheat Head Detection Challenge

CSML 1020 - 2020

Stanislav Taov  
York University  
 Ontario Canada  
 stastaov@gmail.com

Luiz Augusto de Carvalho  
 York University  
 Ontario Canada  
 lcarvalho@wanopt.com

FirstName Surname  
 Department Name  
 Institution/University Name  
 City State Country  
 email@email.com

ABSTRACT

Wheat head detection has an enormous value for the agricultural industry as well as for agricultural researches. We are proposing a number of deep learning techniques that can help with building a robust and accurate computer vision model. These techniques encompass basically manipulating three types of factors:

1. Hyper parameters
2. Minimization of specific training errors
3. Training exposures.

Among these factors the more determining one was the increasing the amount of training exposures. This increasing was obtained through deploying several image augmentation methods to create synthetic images.

CCS

Computing methodologies, Artificial intelligence, Computer vision, Computer vision problems, Object detection

KEYWORDS

Mask RCNN, EfficientDet, Object detection, Data Augmentation.

INTRODUCTION

Agricultural researchers have been using numerous techniques to observe different stages of plant development even though many of those techniques are automated some still need manual involvement. For example, the measurement of the number of wheat heads performed by counting them manually from digital images. This is a very slow and tiresome and challenging task. However, information extracted from those images has tremendous value for researches and farmers since it can help estimate the maturity level, density, enable farmers to make management decisions and provide health conditions of the crop.

Global Institute for Food Security in collaboration with nine research institutes from seven countries: France, UK, Switzerland, Canada, Australia, Japan, and China organized Global Wheat Head Challenge and developed Global Wheat Head Dataset with an aim to create an accurate machine learning model capable of counting wheat heads from digital images. The model should enable agricultural researchers around the world to assess wheat density, health and maturity more effectively and should take into consideration different varieties of wheat, planting densities, patterns, and field conditions.

The dataset provided by the organizers has a quite limited number of files, around 3.500 images. This amount is very small for the computer vision-based deep learning models we are going to use.

We demonstrate in this work that the amount of input data plays a significant role in building high accuracy deep learning models.

We also demonstrate how to circumvent this problem through deploying image augmentation techniques.

We demonstrate that how image augmentation technique can significantly increase the size of our dataset and has direct impact in the predictive quality of our model.

ANALYTICAL SCHEMA

This work has as a final objective to create a model to detect the wheat heads in the images indicating how many and where those wheat heads are located. Therefore our objective is to create a tool capable of executing two processes:

* Object verification: There are wheat heads in the image ? How many ?
* Object detection: Where are wheat heads in the image ?

To achieve our objective we are going to use a deep neural network. A deep neural networks allows a single model to execute both processes (feature verification and detection).

We are going to build a basic model, which we are going to call “Baseline” and try to build upon it working, several parameters. This work is basically divided between the general description and evaluation of this baseline model and the description of the changes made on it and the evaluation of the results.

We worked the changes in the baseline model following three types of strategies:

* Hyper parameters: Changing the hyper parameters we evaluate how the parameters impact the quality of our predictions.
* Filtering the training results through specific training errors.
* Data augmentation: Using data augmentation techniques we evaluate the impact of the increase of the number of samples in our training base in the quality of our predictions.

We could had worked with the BACKBONE model itself. However, in this work we focus our efforts in the three previously mentioned factors (How the type of BACKBONE, the number of layers and the weight treatment impact the quality of our predictions).

Of course, even just for these three factors, due time constrains data processing availability, it wasn´t feasible to test every possible combination of Hyper-parameters, training filters and data augmentation (different techniques).

We are presenting here the combinations which seemed more promising and we know that with enough time and data processing capability, best results can be achieved.

DATA EXPLORATION

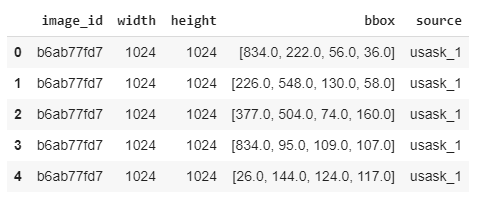
As already mentioned, our basic source of data is the databases provided by Kaggle in the Global Wheat Detection project – sponsored by the University of Saskatchewan. The data consists of images of wheat fields, with bounding boxes for each identified wheat head. Not all images include wheat heads / bounding boxes. The images were recorded in many locations around the world. There are 3.422 images, 47 of them without any wheat head. All have a format of 1024 x 1024 pixels.

The wheat heads presented in these pictures are labeled through boxes defined in a CSV file also provided (train.csv). There are 147.761 wheat heads identified in these pictures. The Wheat heads are spotted through a coordinate identification x,y,w,h which define the two extremes of a rectangle in the following manner:

X1 = x and Y1 = y

X2 = (x+w) and Y2 = (y+h)

The CSV file layout is as follows:

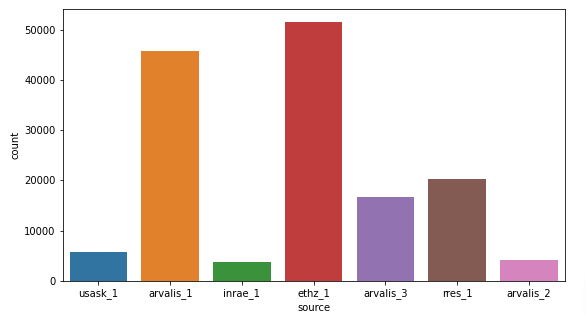


The image ID matches up with the filename of a given image (jpg format), and the width and height of the image are included, along with a bounding box. There is a row in train.csv for each bounding box. As mentioned, not all images have bounding boxes (Or associated wheat heads).



The image above has a name of “b6ab77fd7.jpg” and as we can see the first line of the previous spreadsheet correspond to the wheat head shown in the box (bbox). Of course there is no actual boxes shown in the provided images. This example is just to illustrate to the reader how the correspondence between the images and the labeling file is done (train.csv). Note also that the X and Y axis go from 0 to 1024.

The labels come from seven different sources (We presume the images also) as follows:



We also had 10 images not labelled to be used as testset.

BASELINE MODEL

The first step is to develop a baseline model. In our specific case we are going to use as our BACKBONE basis RCNN (Convolutional Neural Networks) and use a pre-trained model based on the Architectural Design Residual Network – with 101 layers : ResNet101.

RCNN allows training directly on raw pixel values for image verification which is an important simplifier. We are going to use the already trained weights in the resnet101 model add to this existing training our wheat head images, transfer learning.

Transfer learning allows us to train our specific model much quicker and using a lot less image samples. This process is the used in this work and we are describing it following the sequence:

* Preparing the model
* Training the model
* Evaluating the model
* Making predictions.

We are going to separate the database into training and test in a proportion of (90% training and 10% test). We created basically three scenarios:

1. Model trained with the provided images
2. Model trained with data augmentation (15 types) based on imgaug (Augmented images created during the training – on the fly)
3. Model trained based on images actually created based on techniques where we compose new images (ex cutmix)

The first two training, using just the original database and (using the augmented images will be done using colabs with extended GPU and memory (25M) the third one using GCP (Google Cloud Platform). We created three Jupiter notebooks with the code to execute this training:

1. Preparing and training the model
2. Evaluating the model
3. Making predictions

The code preparing and training the model has three versions:

* 1. Using only the original pictures
  2. Using data augmentation using imgaug (on the fly with 15 types of augmentation)
  3. Using other strategies to generate actual new images

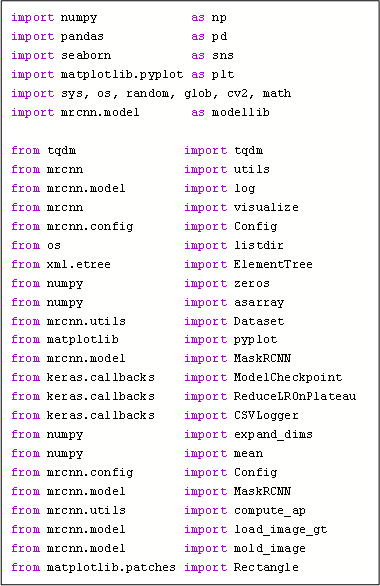


Step -1 – Preparing and training the model



Step 1.2 - Loading packages

After we prepared the environment (see item Environment in the end of this paper) we have to define the packages we are going to need to run our code:



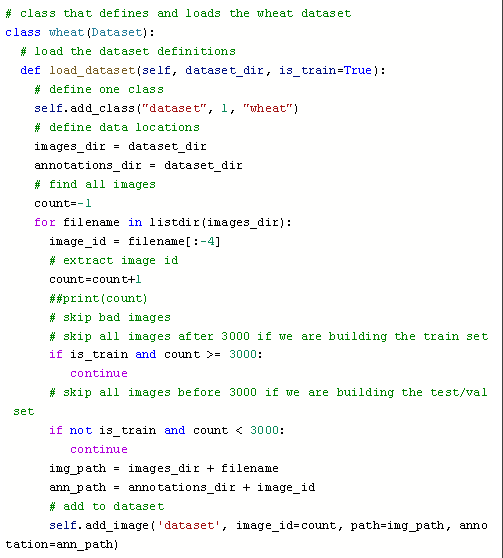
Step 1.3 - Defining the subroutines associated to loading the dataset - wheat class

At this point we have to prepare the routines load our data, that implies basically four aspects images, bounding boxes, masks and image reference.

The idea here is to produce a list where by the number we can get the image file, the bounding boxes associated to the image, the masks associated to the image.

Here it is also important to notice that the input of some of these subroutines are the output of the others. All four sub-routines are grouped into a class called “Wheat”:

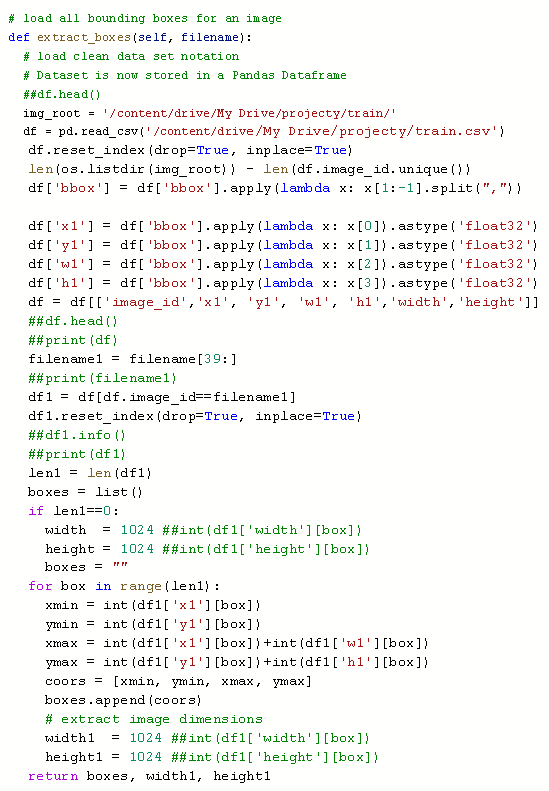
1. First subroutine (load\_dataset)



Note that here we define manually the number of samples to be placed as train and as test. We defined here 3.000 as train and above that test (373). The ann\_path is just to register the name of the file but has no real utility; it will be actually filled by the subsequent routine “extract\_boxes”.

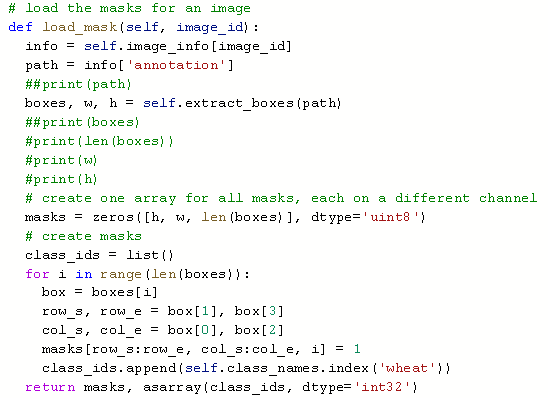
1. Loading the bounding boxes – “extract\_boxes”

This sub-routine is part of the class “wheat” and has as objective load the file train.csv, separate the bounding boxes associated to a specific image and join these boxes in the add\_image in the field “annotation”. Note that we load the csv file in a dataset called df and slice df with the notations associated to the image storing it into the file df1.



Note also that the variables width1 and height1 have a defined value of 1024, it just happens because all our training data has this format, but it could be different. In case we had a heterogeneous we would need to have a code to standardize the images. It wasn´t necessary here, but usually it is.

1. Loading the masks



1. Loading an image reference



1.4 Actually loading the data (separating train and test)

Once we had wrote the sub-routines to load the data we have to actually load it:



Doing this we manage to create two sets of data, indexed by a number. Through this number we can get the images, the bounding boxes and the masks. (train\_set and test\_set).

Step 1.5 - Define the training configuration for the model

When defining the training configuration we have to define if we are going to build a BACKBONE (sequence of Layers) and train it from scratch or use a pre-existing BACKBONE. We may use just part of a pre-existing BACBONE and remove or add layers to it.

If we may decide to use a pre-existing one, we have to decide if we are going to train it again from scratch or use the trained weights total or partially. This is known as transfer learning. In our case we are going to use the RSNET101, not modify it and load the pre-trained weights.

Once we have defined these aspects, we have to train our model. That requires a number of parameters to be specified and tailored to the training of the model.

Note that, even though we are not training our model from scratch (We are using a pre-trained model – transfer learning) we still have to define these parameters, which will be used in the training process using the new data (The wheat heads images). These parameters are consolidate in a class called WheatConfig().

We are going to describe each parameter indicating the value we defined and why.

# NAME

Name of the configuration

NAME wheat\_cfg

# GPU\_COUNT

Set the number of GPUs to use along with the number of images

GPU\_COUNT = 1

IMAGES\_PER\_GPU = 2

# BACKBONE

The BACKBONE being used

BACKBONE = 'resnet101' (we considered using resnet50)

#NUM\_CLASSES

Number of classes (we would normally add +1 for the background) # BG + Wheat

NUM\_CLASSES = 2

# IMAGE\_XXX\_DIM

The default settings resize images to squares of size 1024x1024. If you can use smaller images then you'd reduce memory requirements and cut training and inference time as well, but we decide to keep it as it was in the source database.

IMAGE\_MIN\_DIM = 1024

IMAGE\_MAX\_DIM = 1024

# IMAGE\_RESIZE\_MODE

Force images to be a square (just a precaution all images are squares.

IMAGE\_RESIZE\_MODE = "square"

#STEPS\_PER\_EPOCH

The backpropagation algorithm requires that the network be trained for a specified number of exposures to the training dataset (epochs). Each epoch can be partitioned into groups of input-output pattern pairs called batches.

This defines the number of images that the network is exposed to before the weights are updated within an epoch. It is also an efficiency optimization, ensuring that not too many input patterns are loaded into memory at a time. We changed this parameters several times during the training, reflecting the number of images available.

#RPN\_ANCHOR\_SCALES

Anchors are a set of boxes with predefined locations and scales relative to images. Ground-truth classes and bounding boxes are assigned to individual anchors according to some IoU value. As anchors with different scales bind to different levels of feature map, RPN uses these anchors to figure out where of the feature map ‘should’ get an object and what size of its bounding box is.

The output of a region proposal network (RPN) is a group of boxes/proposals that will be examined by a classifier to eventually check the occurrence of objects. The anchor divide the image in small boxes, therefore the scales should me multiples of the image size. In our specific case we have images 1024 x 1024 that means the anchors are fractions of 1024. We used different size anchors because our target objects are multi-scale (wheats are some too big, some too small).

RPN\_ANCHOR\_SCALES = (16, 32, 64, 128, 256)

# LEARNING\_RATE

The amount that the weights are updated during training is referred to as the step size or the “learning rate”. We worked the learning rate parameter reducing it when the val\_mrcnn\_bbox\_loss gets to a plateau during the training process.

LEARNING\_RATE = 0.005 (starts and from this value goes down)

# WEIGHT\_DECAY

The weight\_decay parameter adds a L2 penalty to the cost which can effectively lead to smaller model weights.

WEIGHT\_DECAY = 0.0005

# TRAIN\_ROIS\_PER\_IMAGE

Maximum number of regions of interest (ROI) is the maximum number of regions of interest per image. In our case we have images with several wheat heads, we selected a number above the worst case scenario (The image with most wheat heads).

TRAIN\_ROIS\_PER\_IMAGE = 350

# DETECTION\_MIN\_CONFIDENCE

This parameters indicate to the model to skip a prediction where it has less than a given percentage of confidence that there is a object there.

DETECTION\_MIN\_CONFIDENCE = 0.60

#VALIDATION\_STEPS

This is basically the batch size of the validation in each epoch, we are using half of the number of the batch size (120).

VALIDATION\_STEPS = 60

# MAX\_GT\_INSTANCES

Maximum number of instances that can be detected in one image.

MAX\_GT\_INSTANCES = 500 # 200

# LOSS\_WEIGHTS

Loss weights for more precise optimization. We considred increasing the weight of “mrcnn\_bbbox\_loss”, but time constrains limited our possibility of simulating the training using this strategies.

LOSS\_WEIGHTS = {

"rpn\_class\_loss": 1.0,

"rpn\_bbox\_loss": 1.0,

"mrcnn\_class\_loss": 1.0,

"mrcnn\_bbox\_loss": 1.0,

"mrcnn\_mask\_loss": 1.0

}

# LEARNING\_MOMENTUM

Moving’ average which would ‘denoise’ the data and bring it closer to the original function. The objective is reduce the impact of the outliers in. The momentum parameter must be chosen. It ranges between 0.0 and 1.0 in our specific case we put it near 1 which removes almost all outliers.

LEARNING\_MOMENTUM=0.9

#LOSS

In a neural networks we are looking for minimizing the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply “loss.” In calculating the error of the model during the optimization process, a loss function must be chosen. In our specific case we are using “mean\_squared\_error”. We have a set of six losses calculated. These losses are calculated from two perspectives: The model against the training:

* Loss = Summary of the five types
* rpn\_class\_loss = RPN (region proposal network) anchor classifier loss – We have just two classes (background and wheat)
* rpn\_bbox\_loss = RPN bounding box loss graph
* mrcnn\_class\_loss = loss for the classifier head of Mask R-CNN
* mrcnn\_bbox\_loss = loss for Mask R-CNN bounding box refinement
* mrcnn\_mask\_loss = mask binary cross-entropy loss for the masks head

The model against the validation:

* val\_loss = Summary of the five types
* val\_rpn\_class\_loss = RPN (region proposal network) anchor classifier loss – We have just two classes (background and wheat)
* val\_rpn\_bbox\_loss = RPN bounding box loss graph
* val\_mrcnn\_class\_loss = loss for the classifier head of Mask R-CNN
* val\_mrcnn\_bbox\_loss = loss for Mask R-CNN bounding box refinement
* val\_mrcnn\_mask\_loss = mask binary cross-entropy loss for the masks head

Each of these loss metrics is the sum of all the loss values calculated individually for each of the regions of interest. The general loss metric given in the log is the sum of the other five losses.

The classification loss values are basically dependent on the confidence score of the true class, hence the classification losses reflect how confident the model is when predicting the class labels, or in other words, how close the model is to predicting the correct class. In the case of “mrcnn\_class\_loss”, all the object classes are covered, whereas in the case of “rpn\_class\_loss” the only classification that is done is labelling the anchor boxes as foreground or background (which is the reason why this loss tends to have lower values, (in our specific case where there are only 'two classes' than can be predicted).

In our specific problem the real issue is to guarantee that the bounding boxes are correct. The bounding box loss values reflect the distance between the true box parameters -that is, the (x,y) coordinates of the box location, its width and its height- and the predicted ones. It is by its nature a regression loss, and it penalizes larger absolute differences (in an approximately exponential manner for lower differences, and linearly for larger differences -). Hence, it ultimately shows how good the model is at locating objects within the image, in the case of “rpn\_bbox\_loss”; and how good the model is at precisely predicting the area(s) within an image corresponding to the different objects that are present, in the case of “mrcnn\_bbox\_loss”.

The mask loss, similarly to the classification loss, penalizes wrong per-pixel binary classifications (foreground/background, in respect to the true class label). It is calculated differently for each of the regions of interest: Mask R-CNN encodes a binary mask per class for each of the RoIs, and the mask loss for a specific RoI is calculated based only on the mask corresponding to its true class, which prevents the mask loss from being affected by class predictions.

As mentioned before, these loss metrics are indeed training losses, and the ones with the val\_ prefix are the validation losses.

Fluctuations in the validation loss can occur for several different reasons, we initially trained our model searching for the smaller val\_loss (two first strategies – 1) only original pictures and 2) augmenting on the fly using imgaug). But during the training process we realized that val\_mrcnn\_bbox\_loss would be much more effective considering our specific problem. We adjusted it in the callback function.

# BACKBONE\_STRIDES

The strides of each layer of the FPN Pyramid. These values are based on a Resnet101 backbone.

BACKBONE\_STRIDES = [4, 8, 16, 32, 64]

# BBOX\_STD\_DEV

Standard deviation admissible in the bounding boxes

BBOX\_STD\_DEV [0.1 0.1 0.2 0.2]

# DETECTION\_NMS\_THRESHOLD

Non-maximum suppression threshold for detection

DETECTION\_NMS\_THRESHOLD 0.3

# MASK\_SHAPE

Shape of output mask, to change this you also need to change the neural network mask branch. We didn´t do that (we kept the model as it was.

# TRAIN\_BN

Train or freeze batch normalization layers None: Train BN layers. This is the normal mode False: Freeze BN layers. Good when using a small batch size True:Set layer in training mode even when predicting

TRAIN\_BN = False # Defaulting to False since our batch size is small

# USE\_RPN\_ROIS

Use RPN ROIs or externally generated ROIs for training, We are keeping this True for most situations. Set to False if you want to train the wheat heads on ROI generated by code rather than the ROIs from the RPN. For example, to debug the classifier head without having to train the RPN.

Step 1.6 - train the model

We divided this process into two, preparing a callback routine and a routine to actually executing the training. The callback function executes some process during the training given a set of requirements. We are requesting that a file with the training variables to be saved every time the variable “val\_mrcnn\_bbox\_loss” gets smaller. We are also saving all variables in the end of each epoch.

1. Call back sub-routine

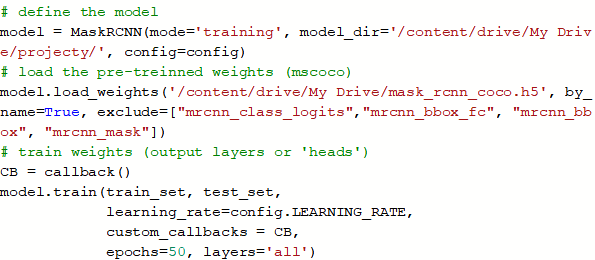
This function defines that we save the trained file of the epoch with the minimum “val\_mrcnn\_bbox\_loss”. We store this data in a file called ‘wheat\_wg.h5” . Basically this file represents the best scenario identified in the training during all epohs.

Here also we save a file called wheat\_history.csv with all parameters of each epoch.



Note that we also change the value of the “learning rate” when during 5 epochs the value of “val\_mrcnn\_bbox\_loss” doesn´t get smaller. We change this value by a factor of 0.3 and limit it to 0.00001.

1. Training



Note that we set the model in the mode=”training”, loaded the pre-trained weighs “mask\_rcnn\_coco.h5”. The type of BACKBONE is defined in the file “config”.

We also define the number of epochs (although the size of the batch is defined in the config file).

There is another parameter defined here “layers”. To understand that parameter we need to understand that we may use the BACKBONE as it is or we may cut out some layers or even add new ones. In our case we are using the BACKBONE (resnet101) as it is. If for example removing the top layers of the BACKBONE the parameter “layers” would be “Top”.

A sequential model (BACKBONE) is a pipeline with the raw data fed in at the bottom and predictions coming out in the top.

In our specific case we are looking for only one specific type of object in the images (although the object may appear more than one time), therefore we are looking for only one class. (There are two classes of objects because one is the background).

Step – 2 Evaluating the model

Once we had trained the model, we created a code to evaluate it.

The first three parts of this code is pretty much like the training module and therefore we are not going to describe it again.



A loss function is used to optimize a machine learning algorithm. The loss is calculated on training and validation and its interpretation is based on how well the model is doing in these two sets. It is the sum of errors made for each example in training or validation sets. Loss value implies how poorly or well a model behaves after each iteration of optimization (epoch). In our case we evaluated two main losses vl\_loss which is the sum of all losses and bbox\_loss which is the basically the error when predicting bounding boxes. We concluded that val\_mrcnn\_bbox\_bbox is the key factor for this specific problem. This is the loss we have to minimize.

For problems like ours, it is important to define an accuracy metric to be used to measure the algorithm’s performance in an interpretable way. The accuracy of a model whose objective is to spot objects in image is usually verified through calculating the percentage of match between the area of the boxes predicted and the actual boxes (between 0 – 1). This is the measure of how accurate the model's prediction is compared to the true data. In other words how correct the bound boxes are predicted. This is the concept o mean absolute predication mAP.

Step 2.4 - subroutine to calculate mean absolute precision (mAP)

As mentioned, we are using the mean absolute precision, or mAP to evaluate how good our model is performing. We need to predict bounding boxes so we can determine whether a bounding box prediction is good or not based on how well the predicted and actual bounding boxes overlap.

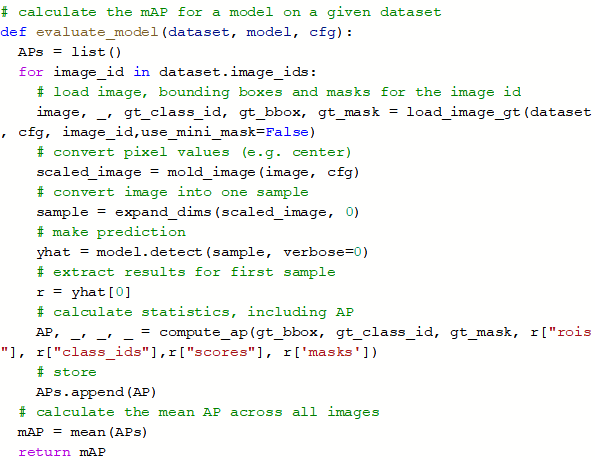
This is calculated by dividing the area of the overlap by the total area of both bounding boxes (real and predicted), or the intersection divided by the union, referred to as intersection over union, or IoU. A perfect bounding box prediction will have an IoU of 1. It is standard to assume a positive prediction of a bounding box if the IoU is greater than 0.5, e.g. they overlap by 50% or more.

Precision in this context refers to the percentage of the correctly predicted bounding boxes (IoU >0.5) out of all bounding boxes predicted.

And Recall is the percentage of the correctly predicted bounding boxes (IoU > 0.5) out of all objects in the image.

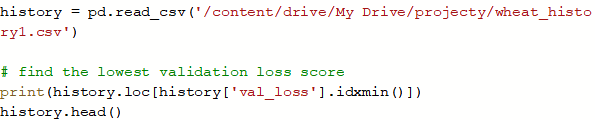
The average or mean of the average precision (AP) across all of the images in a dataset is called the mean average precision, or mAP.

In the sequence we define the subroutine to calculate the mAP

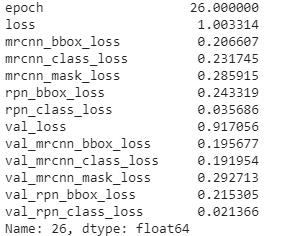


Step 2.7 - Generating tables of training history

In the sequence we create a code to generate the tables showing the training history, demonstrating how the loss changed as the training process evolved:



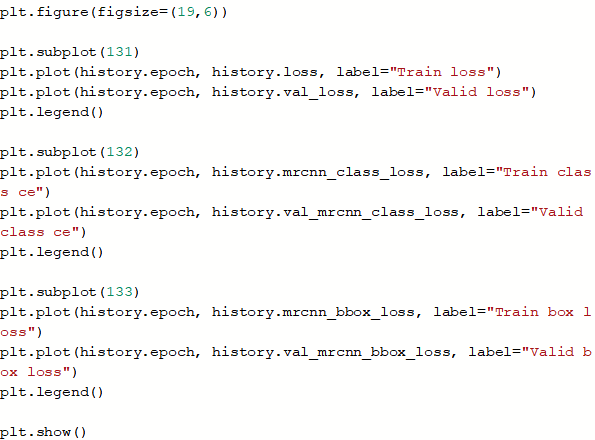
This code generates tables as decribed:



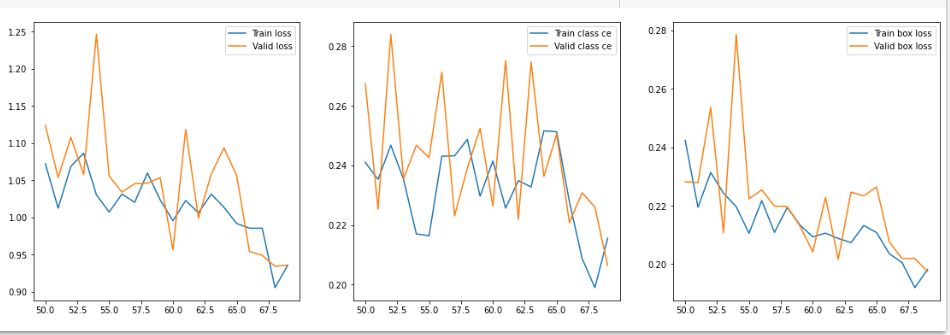
And



Step 2.8 - generating graphics of training history



This code produces graphics comparing the loss in the training set compared with the loss in the test\_set (here called validation).



SCENARIOS

Now we have our code written we are going to experiment trying to achieve the best possible results. In summary a model capable to identify wheat with the best precision and accuracy in every image we show it. Initially we played only with the value of the batches and the number of epochs.

Without data augmentation three scenarios



In sequence we worked with eight more scenarios using data augmentation based on **imgaug** and 15 augmentation techniques:



Note that the first three scenarios we had a number of exposures but with repetition (the actual images are only 3.000). In the second group of scenarios with data augmentation the exposures are of augmented images which may or may not be repeated but most likely are not.



In the sequence we crated two scenarios where we had 3.000 generated images using cutmix.

In the sequence we combined the new 3.000 images with the 3.000 original ones:

Insert stan Scenarios

Step 3 - Making predictions

DATA GENERATION / AUGUMENTATION

The initial scenario shows that the model was trainable but we didn´t have enough labeled images to keep training it. The solution was to use data augmentation. Data augmentation is a technique where we distort a bit the original picture creating new pictures. Proceeding in this manner we manage to increase the training pool and consequently improve the predicative capacity of our model.

Global Wheat Head Dataset is a very limited dataset with just 3.422 images of the wheat head for the training stage. This number of images was reduced even further due using 10% of these pictures for validation. Therefore we had just 3.000 images for actual training.

In general there are several packages which can be used to produce these changes in the images and there are several types of changes which can be used. However in our particular case there are some problems which limits what we can use. We have a problem where we have to distort not only the images but also the bounding boxes.

Initially we manage to find a library for image augmentation in machine learning experiments called **imgaug**. It supports a wide range of augmentation techniques, allows to easily combine these and to execute them in random order or on multiple CPU cores not only treating the images but also the bounding boxes.

We managed to built a routine using 15 types of distortions (augmentation) strategies:

Rotation: Rotates the image using four different angles (we used four angles), note 0 grades corresponds to the original image:

* iaa.Affine(rotate=0),
* iaa.Affine(rotate=90),
* iaa.Affine(rotate=180),
* iaa.Affine(rotate=270),

Flip: We flip the picture left to right and up to down. We used 0.5 as parameter which means we horizontally flip 50% of the images, and 0.2 we vertically flip 20% of the images.

* iaa.Fliplr(0.5),
* iaa.Flipud(0.2),

Dropping rectangles: cutout is defined as an operation that drops exactly one rectangle from an image, while here CoarseDropout can drop multiple rectangles (with some correlation between the sizes of these rectangles):

* iaa.Cutout(fill\_mode="constant", cval=255),
* iaa.CoarseDropout((0.0, 0.05), size\_percent=(0.02, 0.25)

Weather augmentation: Augmenters that create weather effects, we had seven possibilities;

• FastSnowyLandscape

• CloudLayer

• Clouds

• Fog

• SnowflakesLayer

• Snowflakes

• RainLayer

• Rain

We are using just four:

* iaa.Snowflakes(flake\_size=(0.2, 0.4), speed=(0.01, 0.07)),
* iaa.Rain(),
* iaa.Fog(),
* iaa.Clouds(),

Brightness or contrast: Search for all pixels in the image with a lightness value in HLS colorspace of less than 128 or less than 200 (one of these values is picked per image) and multiply their lightness by a factor of X. Note, that we are decreasing and increasing the lightness of the original pixels. We also worked with the contrast adjusting the image contrast by scaling pixel values to 255\*((v/255)\*\*gamma).

* iaa.Multiply((0.8, 1.2)),
* iaa.contrast.LinearContrast((0.9, 1.1)),

Blur: Blur an image by computing simple means over neighbourhoods.

* GaussianBlur
* AverageBlur
* MedianBlur
* BilateralBlur
* MotionBlur
* MeanShiftBlur

We are using just one of them:

* iaa.GaussianBlur(sigma=(0.0, 0.1)),

Sharpen: Augmenters that are based on applying convolution kernels to images. List of possible augmenters:

• Convolve

• Sharpen

• Emboss

• EdgeDetect

• DirectedEdgeDetect

We are using just one of them:

* iaa.Sharpen(alpha=(0.0, 0.1)),

Note that we don’t actually create new images and new rows in the trains.csv file, when we setup augmentation=xxxx in the line of command for training the model it does it for us automatically (this is called on the fly). Each new image in is selected and transformed.

That means our original 3.000 images can become 45.000. That said we can use a combination between epochs and batches 375 epochs with 120 images per batch or 75 epochs with 600 images per batch. Using these parameters we have a fair expectation that we used one image (augmented) only one time. The process of selecting the type of augmented image is random therefore we cannot guarantee that one augmented image will appear only one time, but by the law of great numbers the process works fairly well.

Here there is another parameter to be considered, the number of samples to validate. We are using 60 in a batch of 120. That means we are validating our prediction using half of the number of samples used to train.

*Stan inserts his description*

*As mentioned, initially we used* ***imgaug****, but we also decided to use another library called cutmix which allows the creation of new images from a given training dataset by randomly cutting parts of images and combining them into new images. There is an example to illustrate the results of cutmix.*

A close up of a garden

Description automatically generatedExample 1. (Generated Image)

A green plant in a forest

Description automatically generated

Example 2. (Original Image)

EVALUATING THE RESULTS

Scenario - 1

Initially we trained the model using only the original images, first limiting ourselves to the number of images and in sequence increase the number of exposures of the images:



As we can see in the graphic the loss keeps getting smaller as the number of epochs (exposures) gets bigger.



Two aspects become clear:

* The performance of the model using just the 3.000 images for training was not good. We know that a good mAP has to be above 0.5.
* The performance of the model gets better as it is exposed to more images (the model is trainable)

This two facts together with the reality that we don´t have additional labeled pictures to do additional train pushed us towards the data augmentation.

Scenario 2 - In the sequence we used imgaug using 15 augmentation scenarios with the augmentation made “on the fly” during the training. That allowed us to create 45.000 images and train the model with it:



We trained the model until 24.000 exposures. It becomes clear that the loss wasn´t going down anymore. Note that the values of the mAP are still unsatisfactory. Although it improved 5 times.

In the sequence we used cutmix to created 3.000 new images (and its bounding boxes) and added it to the original pool of images.

Here we followed two strategies:

1. Scenario – 3: We loaded the last best model obtained in the previous phase (wheat\_cfg8.h5) and retrained the models using just the 3000 new images (using imgaug on the fly).
2. Scenario 4: We joined the 3.000 original images and the 3.000 new ones and retrained it loading the original weights (“mask\_rcnn\_coco.h5”).

Scenario 3 – we trained it with 100 epochs;



As we can see using just 3.000 images generated by Cutmix didn´t improve the model, in fact it made it worse. (It was somehow expected).

Stan describes his process and results

THE ENVIRONMENT

The processing of the code was executed in most part in Google colabs and the generation of new pictures and the training of these new pictures were executed using Google Cloud Platform as part of the York University partnership with Google.

The environment in which this work was developed was the following:

