Using wildlife images from Citizen Science databases

BIO8068 Data visualisation in Ecology

# Introduction

You may have noticed from looking at some of the Citizen Science databases we have explored earlier that records often have photographs associated with them. Go back and look at the National Biodiversity Network (NBN Atlas), Global Biodiversity Information Facility (GBIF) and xeno-canto birdsong databases and you’ll see that some, but not all, records have photographs. The quality of photographs is extremely variable, and it is usually the more recently submitted records that are likely to have photographs.

The international iNaturalist website <https://www.inaturalist.org/> and UK-based iRecord website <https://www.brc.ac.uk/irecord/> both provide access to high quality Citizen Science datasets. Both databases have mobile phone apps through which users can submit records, and as a result a high proportion of records include photographs. The UK iRecord site is more restricted in that it does not allow general downloading of images (although they can be viewed on the website).

# The “data bottleneck” in deep learning

You have already created a simple Convolutional Neural Network (CNN) to identify different species, and this required you to have lots of photographs that had already been labelled, that you could then put into your training or validation datasets to create the models. One problem is obtaining sufficient data for machine learning models that has already been labelled correctly. Often this process has to be done by hand, and is time consuming.

# Aims and objectives

In this workshop you’ll learn how to quickly obtain images from the iNaturalist Citizen Science database that can then be used in deep learning models. It will give you an understanding of the problems of data quality that can arise, as well as better technical insights. We shall pick several contrasting butterfly species as an example to work with.

# Bulk download brimstone butterfly images

Begin by loading the rinat package with library(rinat) which provides useful utility functions such as get\_inat\_obs that you have already used to retrieve records. You should also load the sf package with library(sf) as iNaturalist is an international biodiversity website, and for the purposes of this practical exercise we are going to restrict our search to Great Britain. Actually downloading the images takes a bit of R coding, and requires some additional packages to be installed. I have prepared a script, which will setup the additional packages, in particular RCurl for you, and also create a function called download\_images() which does the hard work of downloading and renaming images. Finally, we shall define a bounding box for Great Britain based on the gb\_simple.RDS file you have already used.

library(rinat)  
library(sf)  
  
# Both download\_images.R and gb\_simple.RDS available on Canvas  
source("download\_images.R")   
gb\_ll <- readRDS("gb\_simple.RDS")

Once you have setup your working environment, we can search for records with the get\_inat\_obs() function. Look at the help for this function and you will see that it is quite flexible, in that you can specify a particular month or year. We will restrict our search to a maximum of 600 records, and specify they must be of “research” quality. Again, I must stress that this is a very small dataset, and usually you would have far more to work with. After downloading, we will split each set of images into 3 groups:

* training dataset: this is used by the machine learning model to develop its weights
* validation dataset: during the training process, the outputs are continually compared with the validation dataset, and the weights updated to improve the fit
* test dataset: completely independent, to check model accuracy

Typical splits of the data vary, common ones being 70% train, 15% validation, 15% test, and also 80:10:10 and 60:20:20 are also common. To keep the numbers simple, we’ll use 400 images for training, 100 for validation, and 100 for testing. Admittedly this is 67:17:17 ratio which is not standard! I could ask you to download 1000 images (800:100:100) but suspect the download times might make you bored.

After you have downloaded the images, you’ll need to put them into two sets of folders:

* An images folder, with subfolders for each species. Images will be downloaded into this folder from iNaturalist automatically split into the training and validation sets from here
* A test folder, with subfolders for each species. You will have to move the last 100 images for each species into here manually.

We’ll begin by searching for records of the brimstone butterfly, *Gonepteryx rhamni* <https://butterfly-conservation.org/butterflies/brimstone> You could do a general searh for “brimstone” but this would also return records for the completely different brimstone moth *Opisthograptis luteolata*. They are both yellow in colour, but common names can be misleading, so stick to the Latin names:

brimstone\_recs <- get\_inat\_obs(taxon\_name = "Gonepteryx rhamni",  
 bounds = gb\_ll,  
 quality = "research",  
 # month=6, # Month can be set.  
 # year=2018, # Year can be set.  
 maxresults = 600)

If you look at the records, you’ll see that most of them have a URL to the actual image. If you copy a URL into your web browser you can have a look at one of them. If you wanted, you could filter your results, remember to library(dplyr), at this stage for example to remove any records tagged as larvae rather than adults. Unfortunately, in general this sort of information is missing for most records, so is hard to automate. However, in the UK the larvae are most active May-July, so you could always filter out records for those months if you find too many caterpillar photographs.

Now use the download\_images() function to retrieve the images. This has the following arguments:

* spp\_recs The list of records from iNaturalist, possibly after a filter to improve quality (required)
* spp\_folder The name of the folder into which the image files will be saved (required)
* image\_folder By default this is a subfolder within your RStudio Project called images and will be created for you if it does not exist (optional)

Depending on the speed of your internet connection it may take 20 to 30 minutes or more to download a large number of images. A popup screen will indicate progress, so you will have an understanding of how long it is likely to take:

download\_images(spp\_recs = brimstone\_recs, spp\_folder = "brimstone")

As the files download, you’ll notice that they all have exactly the same name of medium.jpg so the download\_images() function renames them spp\_1.jpg, spp\_2.jpg, spp\_3.jpg etc. to avoid over-writing. They will be stored in a subfolder called brimstone within your images folder. Each individual file is fairly small, but collectively your downloaded images will probably be over 50 Mb. Go to the brimstone subfolder in File Explorer (Windows) or Finder (Mac) and look at some of them. Most of mine are of adult butterflies, but there are a small number of images with larvae, or the butterfly is difficult to see. There are also quite a lot of images where the main object in the photograph is actually a flower from which the butterflies are collecting nectar, typically purple or pink-coloured flowers. Be alert that if you train a model on other species of butterflies that have similar feeding preferences the model may learn to identify the flowers, rather than the butterflies!

# Bulk download holly blue and orange tip butterfly records and images

Now we’ll repeat the process for two other species of common butterflies in a similar way, the holly blue <https://butterfly-conservation.org/butterflies/holly-blue> and the orange tip <https://butterfly-conservation.org/butterflies/orange-tip>:

# Holly blue; Celastrina argiolus  
hollyblue\_recs <- get\_inat\_obs(taxon\_name = "Celastrina argiolus",  
 bounds = gb\_ll,  
 quality = "research",  
 maxresults = 600)  
  
  
# Orange tip; Anthocharis cardamines  
orangetip\_recs <- get\_inat\_obs(taxon\_name = "Anthocharis cardamines",  
 bounds = gb\_ll,  
 quality = "research",  
 maxresults = 600)

One thing to be alert to with both these species is that the males are much more brightly coloured than the females. The overwhelming majority of records of butterflies on iNaturalist do not unfortunately distinguish between male and female photographs, and this will decrease the accuracy of any deep learning models. After you have downloaded the records, again explore them using View(hollyblue\_recs) or View(orangetip\_recs). Nearly all the records include a link to a photograph.

Next download the images, which will be stored in two subfolders, named accordingly, within your images folder. You may want to make a cup of tea or coffee whilst the downloads take place…

download\_images(spp\_recs = hollyblue\_recs, spp\_folder = "hollyblue")  
download\_images(spp\_recs = orangetip\_recs, spp\_folder = "orangetip")

If you look at the photographs in the hollyblue and orangetip subfolders, you’ll notice that the imagery for the latter is not as well-defined. This is likely to give poorer performance in any deep-learning model.

## Put test images into separate folder

Ideally we would have several thousand images for each of our three species, indeed many deep learning approaches uses tens of thousands of images. However I did not want you to have to wait too long to download the images, or do the training, but keep this weakness in mind when you analyse the data. We will the first 500 for training and validation, with 80% (400 per species) for training, and 20% (100 per species) for validation. This 80:20 split can be done automatically by Keras.

However, we need to manually have some totally independent images (100 per species) for testing. We’ll use the files spp\_501.jpg to spp\_600.jpg for each species for this. We can easily move them into a separate folder with the right structure:

image\_files\_path <- "images" # path to folder with photos  
  
# list of spp to model; these names must match folder names  
spp\_list <- dir(image\_files\_path) # Automatically pick up names  
#spp\_list <- c("brimstone", "hollyblue", "orangetip") # manual entry  
  
# number of spp classes (i.e. 3 species in this example)  
output\_n <- length(spp\_list)  
  
# Create test, and species sub-folders  
for(folder in 1:output\_n){  
 dir.create(paste("test", spp\_list[folder], sep="/"), recursive=TRUE)  
}  
  
# Now copy over spp\_501.jpg to spp\_600.jpg using two loops, deleting the photos  
# from the original images folder after the copy  
for(folder in 1:output\_n){  
 for(image in 501:600){  
 src\_image <- paste0("images/", spp\_list[folder], "/spp\_", image, ".jpg")  
 dest\_image <- paste0("test/" , spp\_list[folder], "/spp\_", image, ".jpg")  
 file.copy(src\_image, dest\_image)  
 file.remove(src\_image)  
 }  
}

## Train up your deep learning model

### Initial setup

Now you have downloaded your image data from iNaturalist and sorted it at random into training and validation sets, you can create a deep learning model using a convolutional neural network, similar to your earlier example. The following code is almost identical to the one you used earlier, with only some minor changes. Remember to load the keras package using library(keras) before you continue:

# image size to scale down to (original images vary but about 400 x 500 px)  
img\_width <- 150  
img\_height <- 150  
target\_size <- c(img\_width, img\_height)  
  
# Full-colour Red Green Blue = 3 channels  
channels <- 3

Next rescale your images (255 is max colour hue) and define the proportion (20%) that will be used for validation

# Rescale from 255 to between zero and 1  
train\_data\_gen = image\_data\_generator(  
 rescale = 1/255,  
 validation\_split = 0.2  
)

### Reading all the images from a folder

The flow\_images\_from\_directory() function batches-processes the photographs according to the image\_data\_generator() defined above. We call it twice, once to define the images for training, and once for the validation. To ensure reproducibility, so you get the same results each time you run it, I’ve included a random number generator seed, although note that you will get slightly different results to me. The subset option defines whether the images will be assigned to the training or validation sets.

**Note** You could manually put your images into separate training and validation sub-folders, and define separate image\_files\_paths and omit the subset option, but this takes longer to configure.

# training images  
train\_image\_array\_gen <- flow\_images\_from\_directory(image\_files\_path,   
 train\_data\_gen,  
 target\_size = target\_size,  
 class\_mode = "categorical",  
 classes = spp\_list,  
 subset = "training",  
 seed = 42)  
  
# validation images  
valid\_image\_array\_gen <- flow\_images\_from\_directory(image\_files\_path,   
 train\_data\_gen,  
 target\_size = target\_size,  
 class\_mode = "categorical",  
 classes = spp\_list,  
 subset = "validation",  
 seed = 42)

When you run the two commands above, it should confirm that it has correctly identified different sets of images for the model. Next, check that we seem to have the right number off classes and images:

# Check that things seem to have been read in OK  
cat("Number of images per class:")

## Number of images per class:

table(factor(train\_image\_array\_gen$classes))

##   
## 0 1 2   
## 400 400 400

cat("Class labels vs index mapping")

## Class labels vs index mapping

train\_image\_array\_gen$class\_indices

## $brimstone  
## [1] 0  
##   
## $hollyblue  
## [1] 1  
##   
## $orangetip  
## [1] 2

To look at one of your images you can display it via the as.raster() function (note, you may get a different image displayed):

plot(as.raster(train\_image\_array\_gen[[1]][[1]][8,,,]))



You might be puzzling about all the sets of double-square brackets etc. This is because the train\_image\_array\_gen is actually quite complex, containing information in an R list structure, that will be passed to Python. If you check the size of the first element with dim(train\_image\_array\_gen[[1]][[1]]) you will see it returns 32 150 150 3. This is because by default Keras sub-divides your images into “batches” of 32 images for ease of processing, although you can over-ride this. The 150 x 150 is the size of your rescaled image that you defined earlier. The 3 represents RGB for red, green and blue full-colour images.

### Define additional parameters and configure model

Next define some ‘hyper-parameters’ such as batch size for numbers of images flowing through the system with each ‘epoch’. I’ll define

# number of training samples  
train\_samples <- train\_image\_array\_gen$n  
# number of validation samples  
valid\_samples <- valid\_image\_array\_gen$n  
  
# define batch size and number of epochs  
batch\_size <- 32 # Useful to define explicitly as we'll use it later  
epochs <- 10 # How long to keep training going for

Now you define how your CNN is structured.

# initialise model  
model <- keras\_model\_sequential()  
  
# add layers  
model %>%  
 layer\_conv\_2d(filter = 32, kernel\_size = c(3,3), input\_shape = c(img\_width, img\_height, channels), activation = "relu") %>%  
  
 # Second hidden layer  
 layer\_conv\_2d(filter = 16, kernel\_size = c(3,3), activation = "relu") %>%  
  
 # Use max pooling  
 layer\_max\_pooling\_2d(pool\_size = c(2,2)) %>%  
 layer\_dropout(0.25) %>%  
   
 # Flatten max filtered output into feature vector   
 # and feed into dense layer  
 layer\_flatten() %>%  
 layer\_dense(100, activation = "relu") %>%  
 layer\_dropout(0.5) %>%  
   
 # Outputs from dense layer are projected onto output layer  
 layer\_dense(output\_n, activation = "softmax")

Remember to check the CNN structure before compiling and running it

print(model)

Define the error terms and accuracy measures. Use categorical\_crossentropy as we have more than two species:

# Compile the model  
model %>% compile(  
 loss = "categorical\_crossentropy",  
 optimizer = optimizer\_rmsprop(lr = 0.0001, decay = 1e-6),  
 metrics = "accuracy"  
)

### Set the deep-learning model off (slow!)

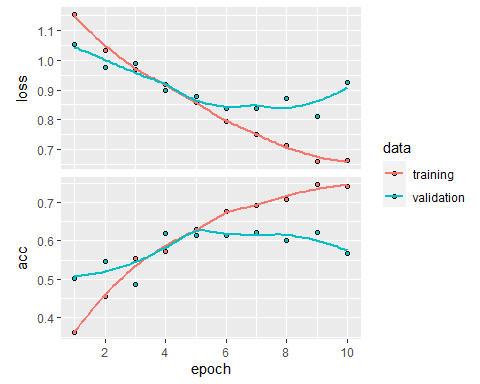
Finally, train up the model. **This is slow**. Depending on your PC, it may take more than 30 minutes to complete. On my PC it takes about 10 minutes, but runs at 100% CPU and the cooling fan comes on. I would also recommend closing any other applications such as Word, Chrom etc. you have running. Occasionally RStudio will crash when running a deep learning model, so **save all your R scripts** before running the next commands.

# Train the model with fit\_generator  
history <- model %>% fit\_generator(  
 # training data  
 train\_image\_array\_gen,  
   
 # epochs  
 steps\_per\_epoch = as.integer(train\_samples / batch\_size),   
 epochs = epochs,   
   
 # validation data  
 validation\_data = valid\_image\_array\_gen,  
 validation\_steps = as.integer(valid\_samples / batch\_size),  
   
 # print progress  
 verbose = 2  
)

### Assessing the accuracy and loss

The results of the model training are stored in the history object, which can be plotted separately (note that your graph and results will differ):

plot(history)



For me there is evidence of overtraining after about 8 epochs, in that the training loss declines, but the validation loss flattens out, or even increases slightly. The accuracy graph shows the revere pattern, in that the training accuracy increases to over 90%, but the validation accuracy is stuck at around 55 to 60%. Whilst 60% is fairly low, keep in mind that:

* you are working on a very small dataset, with only 400 photos per species to build the model
* the quality of the images is quite variable. Some of the photos I have used include butterfly eggs, larvae or pupae, rather than adults.

## Saving your model for future use

It is useful to save your model so that you can re-use it later, without having to go through the whole process of re-training it. For something small, like this example, you can simple save the R data space with the save.image command. For larger models, especially if you are fine-tuning them and want to compare outputs and predictions, it is better to use the dedicated Keras save\_model\_hdf5 which stores it in a special hdf5 format. You can retrieve a model using the load\_model\_hdf5 command.

# The imager package also has a save.image function, so unload it to  
# avoid any confusion  
detach("package:imager", unload = TRUE)  
  
# The save.image function saves your whole R workspace  
save.image("animals.RData")  
  
# Saves only the model, with all its weights and configuration, in a special  
# hdf5 file on its own. You can use load\_model\_hdf5 to get it back.  
#model %>% save\_model\_hdf5("animals\_simple.hdf5")

## Testing your model

We can now test how good the model is with a completely independent dataset, that hasn’t been seen by your model. In other words, the 100 photos per species in the test folder. We create an image\_data\_generator as before, and then “flow” all the images through. The one change is that we have set shuffle = FALSE. By default during training Keras shuffles the images to improve accuracy. However, you don’t want to do this when comparing observed and predicted. We can then push the model into the evaluate\_generator() function to display the loss and accuracy. These will always be **worse** than the ones during model training+validation, simply because the data are totally independent.

path\_test <- "test"  
  
test\_data\_gen <- image\_data\_generator(rescale = 1/255)  
  
test\_image\_array\_gen <- flow\_images\_from\_directory(path\_test,  
 test\_data\_gen,  
 target\_size = target\_size,  
 class\_mode = "categorical",  
 classes = spp\_list,  
 shuffle = FALSE, # do not shuffle the images around  
 batch\_size = 1, # Only 1 image at a time  
 seed = 123)  
  
# Takes about 3 minutes to run through all the images  
model %>% evaluate\_generator(test\_image\_array\_gen,   
 steps = test\_image\_array\_gen$n)

For my data, I ended up with an accuracy of 58.7% which is actually only marginally poorer than that for the training+validation dataset.

## A word of warning about unbalanced data (optional)

Sometimes you may have data where the numbers of species are very unequal, e.g 500 images for a common species, but only 75 for a rare one. Such unbalanced data can skew accuracy measures, making them appear much better than they really are. Another scenario is camera trap data, where 90% of the photos might be blank, and only 10% contain an animal. If you have a poor model, it is still likely to correctly predict a blank (simply because there are so many of them). What really matters is whether it can spot what is going on in the small number of true positives.

There are several methods of resolving this issue of misleading accuracy scores with unbalanced data. Whilst we don’t have unbalanced data here, it’s useful to look at one approach, which is to compare the observed vs predicted for each species. This can be displayed in a ‘confusion matrix’; ideally all the images fall on the main diagonal.

First of all, lets make predictions for our test photographs, but store the results in a data.frame:

predictions <- model %>%   
 predict\_generator(  
 generator = test\_image\_array\_gen,  
 steps = test\_image\_array\_gen$n  
 ) %>% as.data.frame  
colnames(predictions) <- spp\_list

Have a look at the contents of the predictions table. Each row represents one photograph, with the values being the probability of it belonging to each species, the sum of each row being 1.000. So, rows 1-100 are brimstone, 101-200 are holly blue and 201-300 orange tip. Scan through the table: which species has it done best on, and which worse?

The easiest way to confirm your suspicions is to create a confusion matrix. Simply locate the highest probability in each row, and assume that is the predicted class:

# Create 3 x 3 table to store data  
confusion <- data.frame(matrix(0, nrow=3, ncol=3), row.names=spp\_list)  
colnames(confusion) <- spp\_list  
  
obs\_values <- factor(c(rep(spp\_list[1],100),  
 rep(spp\_list[2], 100),  
 rep(spp\_list[3], 100)))  
pred\_values <- factor(colnames(predictions)[apply(predictions, 1, which.max)])  
  
library(caret)  
conf\_mat <- confusionMatrix(data = pred\_values, reference = obs\_values)  
conf\_mat

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction brimstone hollyblue orangetip  
## brimstone 31 0 1  
## hollyblue 2 51 5  
## orangetip 67 49 94  
##   
## Overall Statistics  
##   
## Accuracy : 0.5867   
## 95% CI : (0.5286, 0.643)  
## No Information Rate : 0.3333   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.38   
##   
## Mcnemar's Test P-Value : < 2.2e-16   
##   
## Statistics by Class:  
##   
## Class: brimstone Class: hollyblue Class: orangetip  
## Sensitivity 0.3100 0.5100 0.9400  
## Specificity 0.9950 0.9650 0.4200  
## Pos Pred Value 0.9687 0.8793 0.4476  
## Neg Pred Value 0.7425 0.7975 0.9333  
## Prevalence 0.3333 0.3333 0.3333  
## Detection Rate 0.1033 0.1700 0.3133  
## Detection Prevalence 0.1067 0.1933 0.7000  
## Balanced Accuracy 0.6525 0.7375 0.6800

The above produces a lot of output, but the caret package is quite useful in that it produces nearly all the relevant statistics that you need. You can see from the confusion matrix that the main problem is we heavily over-predict orange tip, especially for brimestone photographs. This might be because the latter are also relatively pale, and in some photos look light coloured. Overall, however the Kappa statistic is highly significant, showing a significant relationship between observed and predicted.

A couple of terms you may have heard on the news in relation to medical tests:

* Sensitivity is the proportion of positives that are correctly identified (true positive rate). So for me, whilst 94% of Orange Tips are correctly identified, only 31% and 51% of brimstone and holly blue are correct.
* Specificity is the proportion of negatives that are correctly identified (true negative rate). Here there are very few errors for brimstone or holly blue. i.e. when it was not a brimstone or holly blue (a negative) it was rarely incorrectly classed as one. In contrast, this was a problem with orange tips.

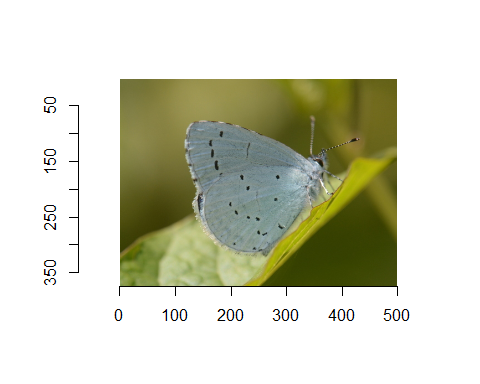
The relevant Wikipedia page gives a nice summary of these statistics, from a medical perspective, but it applies to ecology <https://en.wikipedia.org/wiki/Sensitivity_and_specificity>

## Making a prediction for a single image (optional)

Finally, let’s display a single image from our test dataset, and make a prediction for it. Remember the numbering of your images in the test dataset is 501-600.

We can display the original image in its correct colours using the imager package:

# Original image  
test\_image\_plt <- imager::load.image("test/hollyblue/spp\_508.jpg")  
plot(test\_image\_plt)



We have to import it using the image\_load function from Keras, and rescale, resize it etc., for use in a prediction.

# Need to import slightly differently resizing etc. for Keras  
test\_image <- image\_load("test/hollyblue/spp\_508.jpg",  
 target\_size = target\_size)  
  
test\_image <- image\_to\_array(test\_image)  
test\_image <- array\_reshape(test\_image, c(1, dim(test\_image)))  
test\_image <- test\_image/255

Now we can make a prediction. The next few lines calculates the predicted probability, then formats it into a nice table witht the species names.

# Now make the prediction, and print out nicely  
pred <- model %>% predict(test\_image)  
pred <- data.frame("Species" = spp\_list, "Probability" = t(pred))  
pred <- pred[order(pred$Probability, decreasing=T),][1:3,]  
pred$Probability <- paste(round(100\*pred$Probability,2),"%")  
pred

## Species Probability  
## 2 hollyblue 82.46 %  
## 3 orangetip 17.02 %  
## 1 brimstone 0.52 %