Supplementary Information for

A comparison of deep learning and citizen science techniques for counting wildlife in aerial survey images

Colin J. Torney, David J. Lloyd-Jones, Mark Chevallier, David C. Moyer, Honori T. Maliti, Machoke Mwita, Edward M. Kohi, J. Grant C. Hopcraft

E-mail: colin.torney@glasgow.ac.uk

Guidelines for using the code

All source code used in this project is available at https://github.com/ctorney/deepWildCount. The code is largely based on the original YOLO v3 code available here https://github.com/pjreddie/darknet and a keras implementation available here https://github.com/experiencor/keras-yolo3. We present here an overview of the code and guidelines for applying the methods for the automatic detection of other species. The repository has the following directories,

- `models`: the implementation in keras of the YOLO object detection network
- `train`: the code used to generate training data and train the neural network
- `predict`: the detection and counting of wildebeest from images

Weights files for the neural networks are stored in the `weights` directory.

Models

Implementation of the YOLO models is coded in the file `models/yolo_models.py`. There are two versions of the architecture, the first is an implementation of the original YOLO v3 model trained to detect 80 different classes using the COCO data set (the weights file for this model is available here https://pjreddie.com/media/files/yolov3.weights and a utility to convert to keras network weights is in the file `utils/convertKeras.py`). The second model is a version able to detect only a single class that removes the multiscale output of the original YOLO and only detects similar sized objects. The first model can be created by calling the function `get_yolo_coco(Nx, Ny)`, while the second model can be created by calling `get_yolo(Nx, Ny)`, where `Nx` and `Ny` are the image width and height respectively. Both functions return an instance of a keras model.

Training

There are two stages to creating and training a deep convolutional neural network for object detection. The first stage is to generate a sufficiently large training data set consisting of images and a list of the locations of objects within the images. The second stage is then to use these images and labels to train the network.

Creation of a training data set. Within the directory `train/prepare_samples/` there are several utilities to assist in the creation of training data. There are many alternative applications for creating training data and these other applications could be used in place of these utilities (e.g. https://github.com/Microsoft/VoTT). For training our network all that is required is a file containing a python list of image filenames and the location of objects to be detected within each image. The python list contains a series of python dictionaries that define image properties and object locations. An example of a single entry of this list is shown here,

```
{'object': [{'name': 'wildebeest', 'xmin': 416, 'ymin': 423, 'xmax': 461, 'ymax': 467},
  {'name': 'wildebeest', 'xmin': 0, 'ymin': 217, 'xmax': 34, 'ymax': 239},
  {'name': 'wildebeest', 'xmin': 382, 'ymin': 473, 'xmax': 414, 'ymax': 509}],
 'filename': '/home/staff1/ctorney/workspace/deepWildCount/yolo_v3/train/train_images/SWC1247-22.JPG',
 'width': 864,
 'height': 864}
```

The dictionary contains keys for the image filename, the size of the image and a key for all the objects. The objects key is a nested list of dictionaries that has keys for the properties (type and bounding box coordinates) of each object in the image.

To generate this file we employ the following approach. Firstly we use the file `train/prepare_samples/processImages.py` to iterate over every image in the training data set and run the pre-trained YOLO object detector on these images. This step gives the coordinates of objects that are recognised by the YOLO network trained on the COCO data set. We then filter these objects by comparing to the zooniverse data. If no zooniverse volunteers have indicated the object to be a wildebeest it is ignored. This step removes all other objects that are in the images (trees, cars, bushes etc.) that the non-specialised YOLO network has detected. We then manually check each image using the jupyter notebook `train/prepare_samples/manualOverride.ipynb`. This notebook allows the user to specify the image annotations file then provides an interface which displays each image in turn and enables the user to correct bounding boxes.

Training the network. Once the training data set has been prepared we next train the network using keras. The file `train/train.py` defines the loss function we use and runs the training. This file has the option to specify whether we are fine tuning the network or not. If we are not performing fine tuning we employ transfer learning and load the YOLO model with pretrained weights downloaded from https://pjreddie.com/media/files/yolov3.weights). We freeze all but the final 7 layers of the network and train for 25 epochs (this trains the final layers of the network). Once we have trained the top layers of the network we perform 20 epochs of fine tuning, in which all parameters of the network are modified. The file `train/generator.py` provides utilities for loading training images and implementing data augmentation.

Prediction

Once the network has been trained the prediction step is relatively straightforward. The code that implements this is contained in the file `predict/count.py`. This code implements the following steps:

1. It loops through each of the 1000 images in the test data set
2. Each 4928x7360 pixel image is tiled into 4 subimages of 2464x3680 pixels (due to memory limitations and the high resolution of the images we are unable to run the full size image through the network in a single pass).
3. Each of the 4 images are then passed into the object detector and the output is decoded (output has to be converted to pixel coordinate system as described in (1)).
4. Finally we filter the detections by removing objects below a certain confidence threshold and applying non-maximum suppression to remove overlapping detections. We use threshold levels that are optimal for the training data.

As this code is run it will print the count for each image to the console.

References

1. Redmon J, Divvala S, Girshick R, Farhadi A (2016) You only look once: Unified, real-time object detection in *Proceedings of the IEEE conference on computer vision and pattern recognition*. pp. 779–788.