Banana Detection with Deep Learning

University of Padua

Computer Vision Report Lab 5
Alberto Chimenti

04/06/21

Abstract

This report discusses a naive implementation of a deep learning architecture to recognize and detect the location of different bananas superimposed to images.

1 Code Development

The following section discusses the different modules used to develop this project. The implementation is mostly done in Python, however a simple C++ script has also been written for data augmentation of the training set. The chosen approach is based on training a classifier to recognize the object in smaller patches of the original image. By doing so, successively one can feed a new image splitted into a regular grid to such clasifier and get an estimation of the location of the banana in the image. Finally some location refinement improvements are discussed.

1.1 Data Augmentation

As just mentioned, the starting point was to build the training set. The provided datased includes 1000 images with the bananas superimposed in random locations, plus a *label.csv* file storing information about the original bounding box coordinates of the location of the object for each image. In order to obtain our training set, the *augmentation.cpp* takes care of splitting each image in patches, given the window size (for a square window), the stride, and the overlap threshold. It also generates a new header file in *.csv* format containing several information: file_name, a categorical label, a continuous label and the coordinates of the extracted patch in the original image. Details on how the categorical and continuous label are extracted and their use will be better explained in the following sections.

1.1.1 Structure

To simplify the processing, the main program makes use of two custom classes implemented in the *banana_utils.cpp* file: one for managing the header file and the other for the matching patch extraction.

The first one simply includes a custom method to read the *.csv* file and store its whole content in the class objects for easy later access and another to easily access the bounding box coordinates with one single line of code, given the index of the image.

The second class takes care of processing the image and it also implements two methods: *NetInput::getgrid(int ws, int stride)* used to extract the grid coordinates as a vector of integers, and *NetInput::GetPatch(std::vector<int> bbox)* to extract the patch as subimage given the bounding box coordinates taken from the grid vector.

```
std::vector<std::vector<int>> NetInput::getgrid(int ws, int stride) {
    std::vector<std::vector<int>> result;
    for (int y = 0; y <= in_img.cols - ws; y += stride) {
        for (int x = 0; x <= in_img.rows - ws; x += stride) {
            result.push_back(std::vector<int> { x, y, x + ws, y + ws });}}
    return result;};

cv::Mat NetInput::GetPatch(std::vector<int> bbox) {
    cv::Rect rect(cv::Point(bbox[0], bbox[1]), cv::Point(bbox[2], bbox[3]));
    patch = in_img(rect);
    return patch;};
```

1.1.2 Labels

The label extraction is handled by the function:

```
int findlabel(std::vector<int> orig, std::vector<int> curr, double& overlap,
            double thr_overlap) {

    double x_intersection = std::max(0,
            std::min(orig[2], curr[2]) - std::max(orig[0], curr[0]));
    double y_intersection = std::max(0,
            std::min(orig[3], curr[3]) - std::max(orig[1], curr[1]));
    double intersection_area = x_intersection * y_intersection;
    double orig_area = (orig[2] - orig[0]) * (orig[3] - orig[1]);
    double curr_area = (curr[2] - curr[0]) * (curr[3] - curr[1]);
    double union_area = orig_area + curr_area - intersection_area;
    overlap = intersection_area / orig_area;
    if (intersection_area / union_area > thr_overlap) return 1;
    else return 0;};
```

Given the original bounding box coordinates and the current patch ones, it computes the intersection and union area of the two. The categorical label (output of the function) is set to 1 if the ratio between the intersection area and the union area is greater that the threshold and to 0 otherwise. While the continuous label (overlap variable) is set equal to the ratio between the intersection area and the area of the original box, representing the percentage of the original bounding box included in the patch.

While the categorical label serves an obvious role for our classifier training, the continuous one will be used for box localization refinement which will be explained in the last sections of the report.

1.1.3 Additional Feature

Lastly, one important aspect to treat is the imbalance of the generated dataset labels. Since the superimposed objects are small with respect to the images, a simple fixed grid patch extraction would lead to very few samples actually containing the object and a lot more "empty" ones. To balance such difference, the main script takes two optional inputs: *train_mode* and *additional*. The first one is a boolean which is set to true if the word "train" is passed to the program. Such boolean controls the acquisition of additional samples including one sample centered on the original box (which otherwise would be difficult to obtain with the fixed grid) and a given number, set by the *additional* input variable, of supplementary samples with center coordinates randomly extracted from the original object box.

This procedure was very important to ensure heterogeneity in the training samples and improve the generalization capabilities of the classifier.

1.2 Deep Neural Network

The implementation of the Deep Neural Network has been performed in *Keras* using *Tensorflow Dataset* object. This section will explain in better detail the data loader approach, the architecture of the classifier and the selection method for correct predictions.

All of the classes and methods discussed below are contained inside the *model* directory in *dataset_class.py* and *tf_model.py* scripts.

1.2.1 Dataset Class and Pipeline

In order to reduce the amount of memory occupied by data during training, the *BananaDataGenerator* class was written to implement on-demand loading of the input images through a generator object. The only data that is fully loaded in the actual memory is what is contained in the header *.csv* file.

The function *load_image* is a modular component responsible for loading a certain image given its filename. It is worth noting that in case one would want to perform any kind of preprocessing operation on such image (like resizing or changing the color space to a standard one) this could be easily achieved just by adding the few lines of code performing such tasks in this class method.

The second layer component of the pipeline is the *generator* module which creates an iterable object used to load the images in memory in an on-demand fashion.

Finally the *pipe* method is the higher level module which creates the actual Tensorflow dataset class object and, if requested, performs operations like: repeat, batching and caching.

In the following we see the simplest implementation of such structure:

```
def load_image(self, fname : str) -> np.ndarray:
        [...]
        img = img_to_array(plt.imread(os.path.join(self.imgfolder, fname)), dtype=np.float32)
        ### Possible addition of color space change or resizing
        return img

def generator(self):
        for i, name in enumerate(self.fnames):
            yield self.load_image(name), np.array(self.labels[i])

def pipe(self, batch_size : int = None, cache_file : str = None, repeat : bool = True):
        dataset = Dataset.from_generator(lambda: self.generator(),
                                         output_types=(float32, float32),
                                         output_shapes=((self.imshape, ())))
        dataset = dataset.cache(cache_file)
        dataset = dataset.repeat()
        dataset = dataset.batch(batch_size)
        dataset = dataset.prefetch(buffer_size=1)
        return dataset
```

1.2.2 Test Set Prediction Pipeline

In order to try implementing an alternative way of splitting the given image in smaller patches, without having to load an unnecessary large number of small images, a supplementary pipeline has been created in the dataset class object called *split_image_pipe*. It exploits the same structure as before but in this case the *split_image_generator* yields just a patch of the image at each iteration. At the same time a list is created containing the coordinates of the matching patch box. In this way, the association of the single patches with the full image is easily maintained and used to find the correct boxes corresponding to positive detection of the classifier.

1.2.3 Classifier Model

The classifier model is constructed using *Keras* modules exploiting the flexibility of the *Functional API*.

The model consists of three *Convolutional* hidden layer modules comprising a *Batch normalization* layer (on the color channel axis) and a *Max pooling* one. The kernel size for the convolutional layer and the max pooling are fixed across all of the three and are respectively $3x3$ (with $stride = 1$ and padding) and $2x2$. The selected activation function is the *ReLU*, while the number of kernels of the three layers are $8, 16, 32$. Also a *Dropout* layer is inserted between the second and the third layer with probability of 0.2 to reduce the chances of overfitting the training data.

Successively, the output of the CNN is flattened, then passed through a 0.2 *Dropout* layer and finally fed to two *Dense* layers with a single neuron output layer with *ReLU* and *sigmoid* activation functions.

For what concerns the optimizer algorithm, *Adam* was chosen with initial learning rate of 0.001 and a custom scheduler callback to superimpose an exponential decay.

Finally as we are treating a classification problem, the *binary Cross Entropy* loss function was used.

1.2.4 Network Prediction Selection

Since our model's output comes out of a sigmoid function, the prediction label associated with each patch of the image will be a continuous value between 0 and 1. This, leaves to the classifier room to also output imprecise detections and still obtain very good results for our purpose. We can see in *Figure(1)* that the chosen detections are the ones with prediction probability over 90%. However, when detections above such threshold are not found, the *threshold()* method (part of the *PredictionLabels* class) responsible for filtering the results, recursively calls itself reducing the initial threshold value by 10% until at least one detection is found or the value 0.5 is overcome.

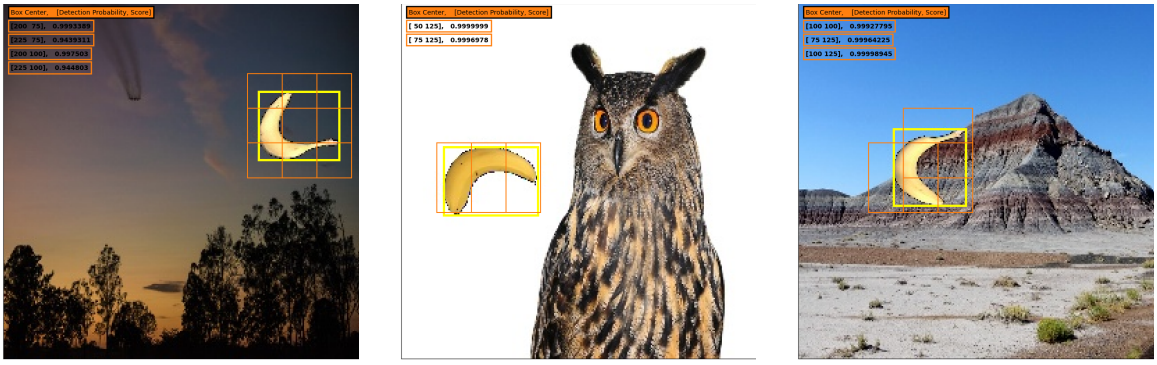

Figure 1: Example of classifier model detection. ([yellow]: original bounding box, [orange]: patches with positive detection)

1.3 Location Refinement: Regressive Approach

As one can see from the examples in *Figure(1)*, all of the patches detections are correctly placed, but the overall localization of the box is limited by the predefined gridding. In order to smooth such limitation and obtain also sub-grid localizations a regressive approach has been proposed. Together with the classifier output from the network the idea was to also obtain a score which says how important is the detected patch with respect to the others. Here comes into play the *continuous label* previously mentioned. Indeed, such quantity has been used, together with the categorical one, to train a multioutput model with two identical branches: one trained as a classifier to predict the categorical label, and one trained to solve a regression problem to predict the continuous label. The architecture diagram is reported in the Appendix.

This approach needed some customization of the vanilla *Keras* model structure and the flexibility of the Functional API came to aid.

```
def Classifier_with_regression(self, input_shape):
        image_input = keras.Input(input_shape,name='image')

        class_branch =self.Classifier_branch(image_input)
        reg_branch =self.Regression_branch(image_input)

        model = keras.Model(inputs =image_input,
                                    outputs = [class_branch, reg_branch],
                                    name = 'Banana_classregressor')
        return model
```

By creating two separate branches in the model one is able to feed a multilabel input in form of a python dictionary with keys corresponding to the branch names, predefined as separate modules.

1.3.1 Prediction Smoothing

The scoring associated with each patch is then used as weight factor to compute the centroid of the detected patches central points. Finally, the input parameter of the *detector.py* script *smoothing_factor*, is responsible of deciding how much each of the box centers has to move in the direction of the centroid. By doing so we obtain a new, more precise, sub-patch localization of the detected boxes and the extreme box corners are taken as reference to construct a big box which is chosen as final prediction.

Figure 2: Example of classifier model detection with regressive smothing approach. Smoothing factor set to 0.6. ([yellow]: original bounding box, [orange]: patches with positive detection, [red]: final box prediction)

2 Example Results & Performance

Here are underlined limitations and possible improvements to the just explained approach.

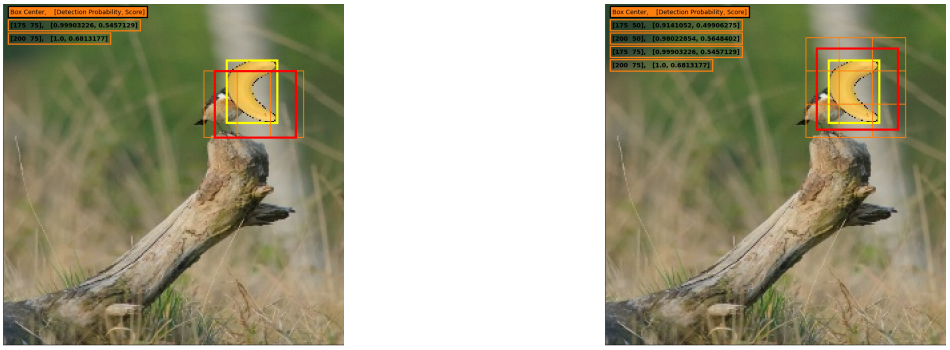

Figure 5: Example of misclassified boxes. Left: threshold=0.99; Right: threshold=0.9 ([yellow]: original bounding box, [orange]: patches with positive detection, [red]: final box prediction)

In *Figure(5)* we can clearly see that the threshold is crucial to obtain the correct output. These examples show the limitations of the model in generalizing the results. This might be due to imbalances in the presented training samples and, in case of the first example, also the asymmetry of the object to detect (see the lack of localizations in the upper-left zone).

Moreover, notice in the first example that the scoring predicted from the regressive branch does not seem to prefer the more centered box to the shifted ones. But to a certain extent in the second example such condition does not hold. Here in the right image the much shifted box is associated with a score which is 1/3 times the others. Indeed even if the detection in the left image is much better, the deviation of the final prediction box in the right one is not as big as we would expect.

Something else worth mentioning is that the proposed smoothing solution yields better results but probably needs some refinements in the data heterogeneity provided during training. Moreover, the smoothing factor is unfortunately an arbitrary value which has to be fine-tuned, depending on the case, to output the best final possible result.

2.1 Possible Improvements

The biggest underlying limitation of this approach lays on the fact that the information processed from the network about the object is not scale invariant. Indeed, one can observe that smaller objects are better detected but with lesser precision on the contours while bigger ones can be cropped on the sides. Such problem could be resolved by analyzing the grids for different scales of the same starting image. This could give the algorithm much better localization capabilities.
However it is important to consider also the impact that such improvement would have on the performance of the detection. Currently the program is able to detect objects at $5FPS$ (0.2s).

Another possibility would be to use a ROI-based detection. Select some macro regions in the image as proposals and run the detection on those smaller subregions. However, the technique employed to extract such regions could also affect the performance significantly.

A Training Performance

Here we show the results of the training performances for the network with regression branch for the sake of completeness.

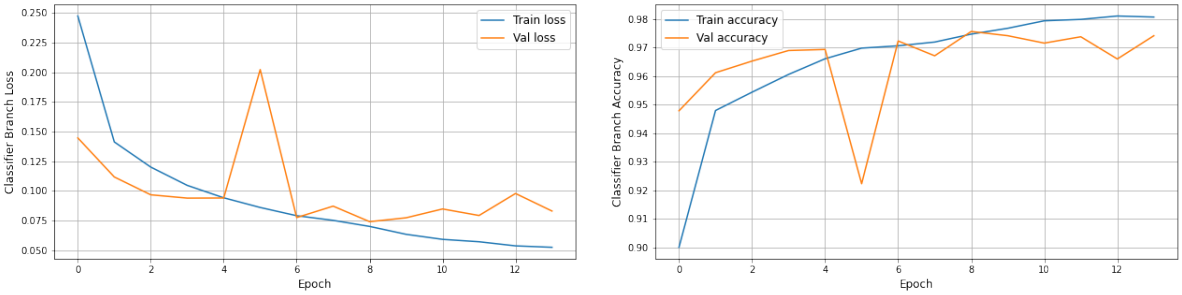

Figure 6: Classifier branch training performance. Left: Cross Entropy loss; Right: Accuracy

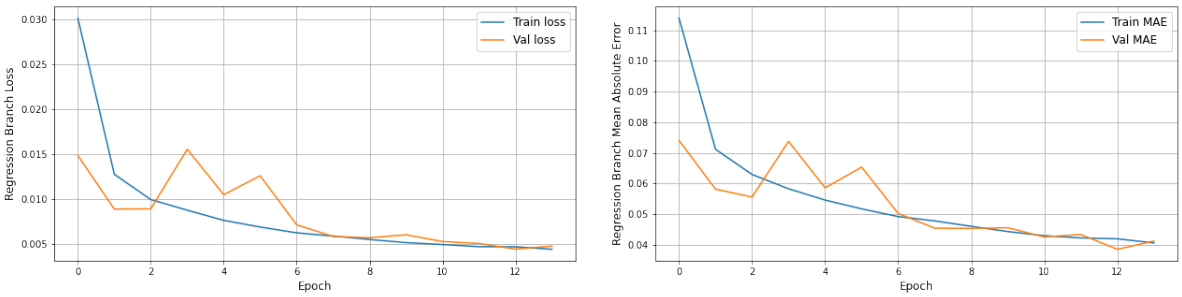

Figure 7: Regression branch training performance. Left: Mean Square Error loss; Right: Mean Absolute Error

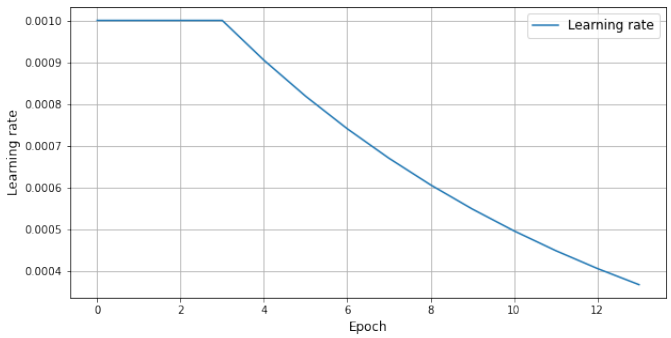

Figure 8: Learning Rate protocol

The training has been performed using augmentation script with additional sampling of 15 patches per image. The selected window size and stride are respectively 50 and 25. This yields a total number of 95938 samples among which 13578 correspond to positive detections with 0.3 overlap threshold. Then the dataset has been balanced making the number of the two sample types identical. Finally 10% of the samples were randomly selected as validation set.

A.1 Performance on the Test set

From the test set, 81000 samples of unbalanced patches have been extracted. The evaluation of the architecture performance yielded the following results:

Class_branch Loss	Class_branch Accuracy	Reg_branch Loss	Reg_branch MAE
0.0592	0.9821	0.0026	0.0206

Table 1: Evaluation performance on test set

B Network Architecture Graph