NEURAL NETWORKS' PROJECT REPORT

"Re-Implementation of YOLOv3 Paper using Tensorflow 2.1.x"

Romani Gianfranco - 1814407

George Ciprian Telinoiu - 1796513

2020/21

INTRODUCTION

Our work is based on the Paper published by Joseph Redmon and Ali Farhadi from the University of Washington back in April 2018 and presents refinement number 3 of their image detection algorithm, YOLO *(You Only Look Once)*.

The main differences with previous versions of YOLO come from the improvement on the Sub-Model that deals with feature extraction called *Darknet* that with YOLOv3 becomes a 53-layered Convolutional Model compared to the 19 that it had in the previous version.

Also, YOLOv3 introduces prediction across scales, specifically it predicts boxes at 3 different scales. This method enables getting more meaningful semantic information from the upsampled features and finer-grained information from the feature map produced by Darknet. Thus the predictions for the 3rd scale benefits from all the prior computation and there is a significant improvement in the detection accuracy of smaller objects.

Our goal, therefore, has been to reproduce from scratch the Neural Network used to make predictions on images from the COCO Dataset using the Tensorflow library by leveraging a Pre-Trained weight for the Model available from the creators' website.

The paper will briefly describe the key theoretical aspects behind the YOLO algorithm and proceed to thoroughly describe the code used in order to re-implement the detection process, starting from the model creation up to the final detection phase.

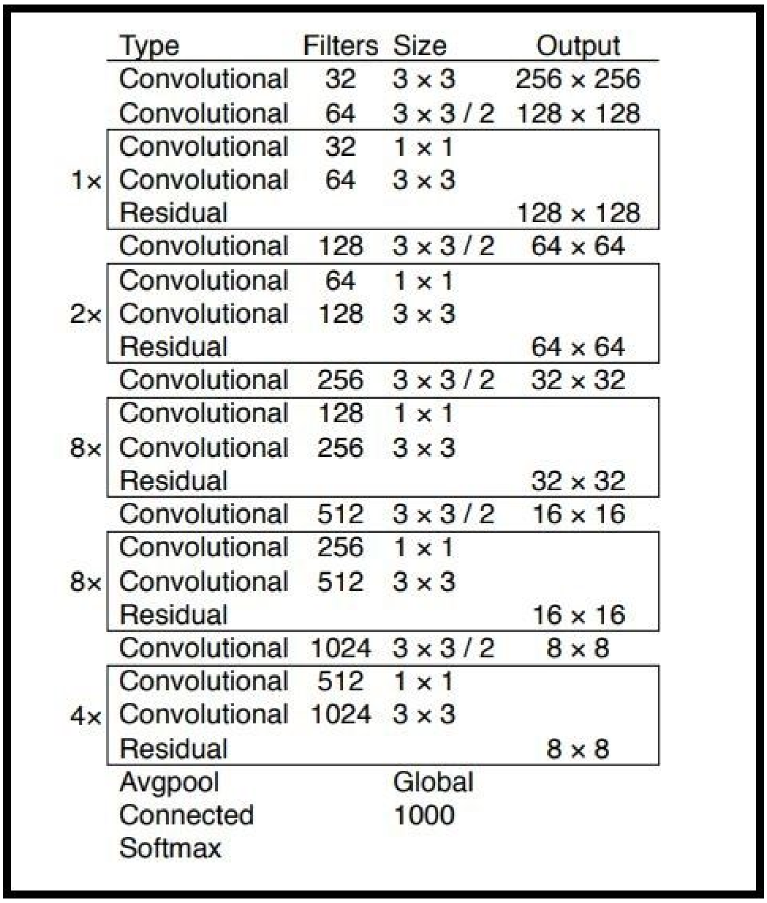
THEORY

Before YOLO, object detection models had to process images multiple times to be able to detect all objects. This approach is obviously slower compared to a model that needs a single forward pass to the image to predict bounding boxes and class probabilities, as YOLO does. The main idea is to divide the image into a grid and if the center of the detection falls into the grid, then this one is responsible for detecting. Each grid will eventually output a bounding box, a confidence score, that tells us whether there is an object or not, and the class probability. Bounding boxes contain four values: two for the coordinates of the center of the box and the others for the width and the height.

YOLOv3 improves precision over the previous version, but it is also a little slower. The changes that were implemented that have effects on the robustness of predictions regard the network architecture. We can divide it into two main parts that we are going to analyze distinctly: the feature extractor and the feature detector.

Darknet-53

The feature extractor for the first versions of YOLO was based on Darknet-19, a neural network designed on purpose for this model. As the name could suggest, this neural network originally had 19 layers, which were increased to 30 in YOLOv2. Since the algorithm had problems detecting small objects because of the downsampling of the input, in this version of YOLO the architecture passed to have 53 layers (3x3 and 1x1 convolutional layers) with (short) skip connections. Skip connections were introduced by ResNet to tackle the problem of vanishing gradient, so the fact that the gradient can become very small for the earliest layers of the model. They consist of non-sequential layers that feed with their output not only the successive layer but also the next ones.



	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

The backbone model has three branches (or routes as we call them in our implementation) that are used to obtain three feature maps of different sizes, 13x13x255, 26x26x255 and 52x52x255. Downsampling the input with three ratios helps to focus on the detection of big, medium and small objects (in older versions of YOLO small objects were troublesome to detect).

Multi-scale Detector

The final part of the architecture is the multi-scale detector that applies 1x1 kernels and some concatenations on the three feature maps computed by the previous part of the algorithm to predict $B * (5 + C)$ values, where B stands for the number of anchor boxes and C is the total number of classes. The concatenation is made possible thanks to upsampling using nearest-neighbor interpolation.
Anchor boxes are pre-defined boxes with a certain aspect ratio set (aspect ratio is computed before training using k-means clustering on the entire dataset, but we used the standard values for COCO dataset). In YOLOv3 there are 3 anchor boxes for every detection scale. After the output of the three branches is computed, we need to decode them to adapt the anchor boxes to the predicted boxes. Given x and y, the offset of the center of an object from the upper left corner of the grid, p_w and p_h length and width of the anchor and c_x and c_y, coordinates of the upper left corner of the grid, the values for the predicted box are computed as $b_x = \sigma(x) + c_x$, $b_y = \sigma(y) + c_y$, $b_w = p_w e^x$ and $b_h = p_h e^y$, where σ stands for the sigmoid function *(see picture for explicit formulas)*. Sigmoid is used to squash values between 0 and 1 so that the relative coordinates fall in the considered cell and not outside. Centroid locations computed in this way, as stated by the authors of the paper, is not enough anyway, it is necessary to normalize the two values dividing by the *strides* value, computed as dimensions of the image divided by the *grid_shape*. Exponential instead is applied because x and y could be negative values while width and height can't.

$$b_x = \sigma(t_x) + c_x$$

$$b_y = \sigma(t_y) + c_y$$

$$b_w = p_w e^{t_w}$$

$$b_h = p_h e^{t_h}$$

IoU & NMS

The last operation to do is to discard the boxes with low confidence scores and the ones with high overlap. For such tasks, Non-max suppression is used.
This algorithm uses the concept of IoU *(Intersection over Union)* that measures the overlap between two proposed bounding boxes, the steps are:

1. Select the proposal with the highest confidence score, remove it from the list of proposed boxes B and add it to the final proposal list R. *(Initially R is empty)*;
2. Compare this proposal with all the proposals, which means the IOU of this proposal with every other proposal has to be computed.
   If the IOU is greater than the threshold N, remove that proposal from B;
3. These two steps are repeated until there are no more proposals left in B.

DEVELOPMENT

Tools

Development has been done in cooperation utilizing a common git repository within a Ubuntu 18.04 environment. *(Repo's Link)*
As IDE we opted for Visual Studio Code and our main support libraries have been:

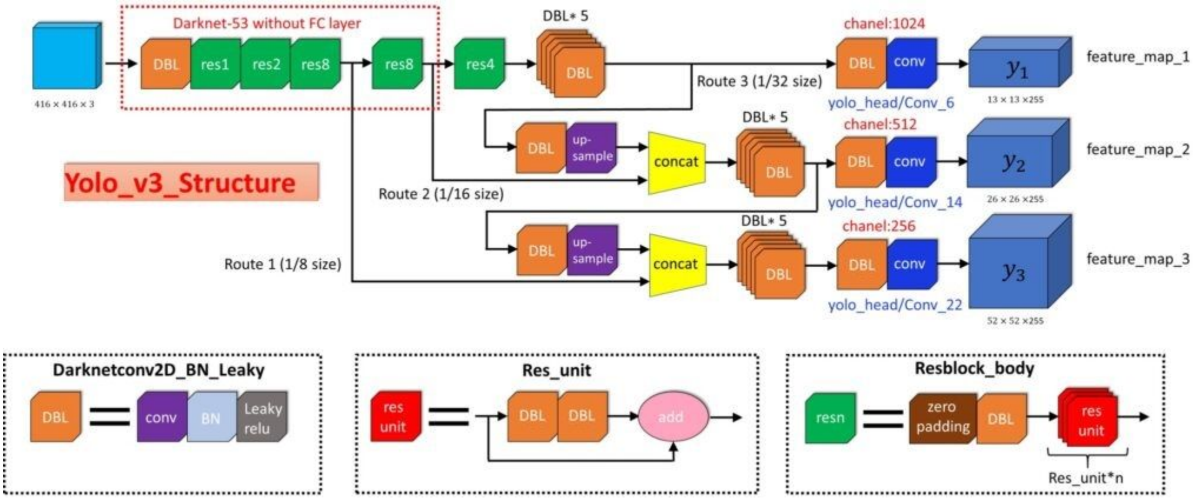
- Tensorflow
- Numpy
- Open-CV

The project's repository had been planned as follows:

- [DATASET] (ref COCO)
    - coco.names
    - dog.jpg | horses.jpg | kite.jpg | scream.jpg
- [LAYERS]
    - common_layers.py
        - batch_norm
        - conv_2d_padding
        - convolutional_block
        - darknet_residual
        - yolo_convolution_block
        - yolo_layer
    - darknet53.py
        - darknet53
    - yolov3_model.py
        - upsample
        - yolov3
- [UTILS]
    - boxes.py
        - read_class_names
        - image_preprocess
        - postprocess_boxes
        - non_max_suppression
        - draw_boxes
    - load_weights.py
        - load_yolo_weights
- detection.py
    - detect_image
    - main
- yolov3.weights

Therefore our roadmap had been to first design the Neural Network, load the weights on it and proceed to implement the detection phase on a given input test image.

Model Creation



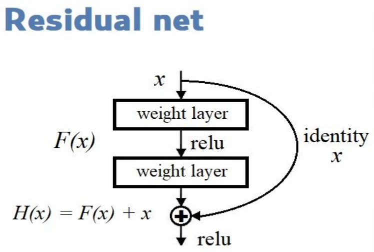
[common_layers.py]
Our work started within the *LAYERS* directory with the *common_layers* module where we stored all building blocks necessary to construct the models that we needed.

The first 2 functions that we needed were *batch_norm* and *conv2d*, for which we used *BatchNormalization* and *Conv2D* methods the *tf.keras.layers* library's module adding some modularity for the convolutional layer, specifically we introduced as inputs of the method the *kernel_size* and *strides* in order to be able to reutilise the same function in all our project.

Once we had those 2 primitives, we were able to finalize a true building block of YOLO, specifically the *convolutional_block* composed by a convolutional layer, followed by a batch_normalization layer and a final *LeakyReLU* activation layer.

Having done this, we moved to the *yolo_convolutional_block* that returned 5 stacked *convolutional_block* layers and a single *convolutional_block* used by the model before moving into detection for all 3 scales.

On the side, we also created a *darknet_residual* layer that it's just an adaptation of the classical ResNet Residual Block used within the DarkNet-53 Model. *(image on the side).*

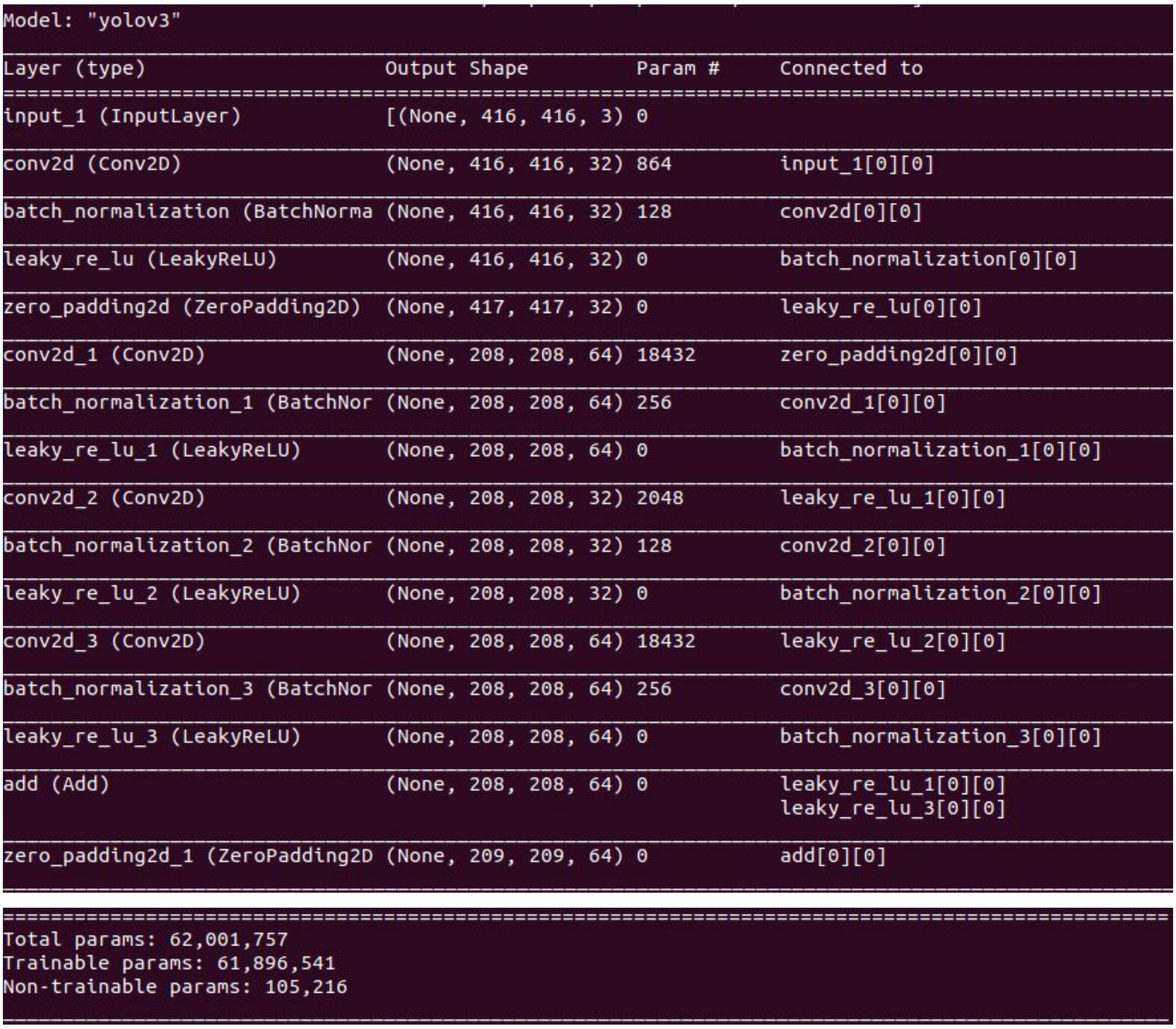


Finally, we have *yolo_layer* that handles the identification of the bounding boxes on the provided input tensor for each of the 3 scales of interest. The input gets stacked to a final 1*1 Convolution with *n_anchors * (5 + n_classes)* filters from which we will compute the resultant bounding boxes. Before we start working on it we need to re-shape it into a 3-D Tensor whose last axis is composed of the 2 coordinates of the centre, the width and height of the box, the objectness score and the 80 classes_scores.

We then proceed to pre-compute the x_y_offset for the box, squash the values of the box_centers, objectness score and class_scores between 0 and 1 using the sigmoid function and finally putting it all together into a new tensor called *boxes*.

[darknet53.py]
This method returns the 3 output tensors of the feature extractor DarkNet by stacking the previously defined layers, following the provided structure of the authors.
We decided not to return a Model at this point since we would have needed an extra *Placeholder Tensor* aside from the one that we use within the actual construction of the final model in *yolov3_model.py* and through various iterations, considering all the other functions and specifically the load_yolo_weights function, we decided to proceed this way.

[yolov3_model.py]
Now that we had all the blocks necessary to build the YOLO Model we defined the *yolov3* method that starting from a Placeholder Tensor, returned a *tensorflow.keras Mode*l whose last layer is the concatenation of the detections for each of the 3 scales.
Below we show the the beginning layers and the total parameters of the final model:



```
Model: "yolov3"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 416, 416, 3) 0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 416, 416, 32) 864         input_1[0][0]
__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 416, 416, 32) 128         conv2d[0][0]
__________________________________________________________________________________________________
leaky_re_lu (LeakyReLU)         (None, 416, 416, 32) 0           batch_normalization[0][0]
__________________________________________________________________________________________________
zero_padding2d (ZeroPadding2D)  (None, 417, 417, 32) 0           leaky_re_lu[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 208, 208, 64) 18432       zero_padding2d[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 208, 208, 64) 256         conv2d_1[0][0]
__________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 208, 208, 64) 0           batch_normalization_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 208, 208, 32) 2048        leaky_re_lu_1[0][0]
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 208, 208, 32) 128         conv2d_2[0][0]
__________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)       (None, 208, 208, 32) 0           batch_normalization_2[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 208, 208, 64) 18432       leaky_re_lu_2[0][0]
__________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 208, 208, 64) 256         conv2d_3[0][0]
__________________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)       (None, 208, 208, 64) 0           batch_normalization_3[0][0]
__________________________________________________________________________________________________
add (Add)                       (None, 208, 208, 64) 0           leaky_re_lu_1[0][0]
                                                                 leaky_re_lu_3[0][0]
__________________________________________________________________________________________________
zero_padding2d_1 (ZeroPadding2D (None, 209, 209, 64) 0           add[0][0]
__________________________________________________________________________________________________

==================================================================================================
Total params: 62,001,757
Trainable params: 61,896,541
Non-trainable params: 105,216
__________________________________________________________________________________________________
```

Loading Pre-Trained Model

[load_weights.py]
To test the detector we decided to use a pre-trained model, so we downloaded Darknet-53 weights (from here) obtained from training on the COCO dataset. These weights had to be converted to a TensorFlow model to be used by our implementation.
This is done by iterating on the darknet-53 weights' binary file to retrieve parameters of interest for convolutions and batch normalization (*get_layer* method from *tensorflow.keras.model*) and shapes of the model (height, width, in_dim, out_dim). Batch normalization is not considered for the layers that define the three scales for detection. At each iteration, we set the parameters of our model using *set.weights* method.

Prediction

[detection.py]
Within the *main()* function of our program we finally proceed to execute the detection on our input image. The first necessary step is initializing the *tf.keras.Model* by calling the *yolov3* method that immediately gets used by the *load_yolo_weights* method in order to have a ready and functional Model at our disposal.

At this point we have the *detect_image* method that will return our input image with the proper labelled bounding boxes.

[boxes.py]
The process starts from loading the image as a *numpy* array through the *cv2* library that will get padded and reshaped in order to handle all possible input shapes. Afterwards we need to add 1 dimensionality to be coherent with the input required by the *tf.keras.predict* that will return all the predicted bounding boxes. The last step is isolating the best bounding boxes, which is handled by our 2 methods *postprocess_boxes* and *non_max_suppression*.
The first one rescales the coordinates of the bounding boxes considering the original image and handles edge case scenarios such as out-of-range boxes, invalid scale boxes and low score_threshold boxes.
The second one uses the *tf.image.non_max_suppression* method that identifies the best ones among the previously selected boxes, mainly to avoid the problem of overlapping boxes for each class detection considering the 3 scale process.
The result is a 2D tensor containing for each row the final detections with the coordinates of the bounding box, the precision score and finally the class identifier.

Finally using the cv2 library we draw on our input image the final bounding boxes, since we now only have unique detections within our input tensor.

[detection.py]
The final image gets prompted to the end user and before exiting gets also saved in the current directory.

VALIDATION

In this section we will briefly compare 4 detections from our detector with the original one proposed by the authors that leverages their custom Neural Network framework called Darknet ([Repo's Link](#))

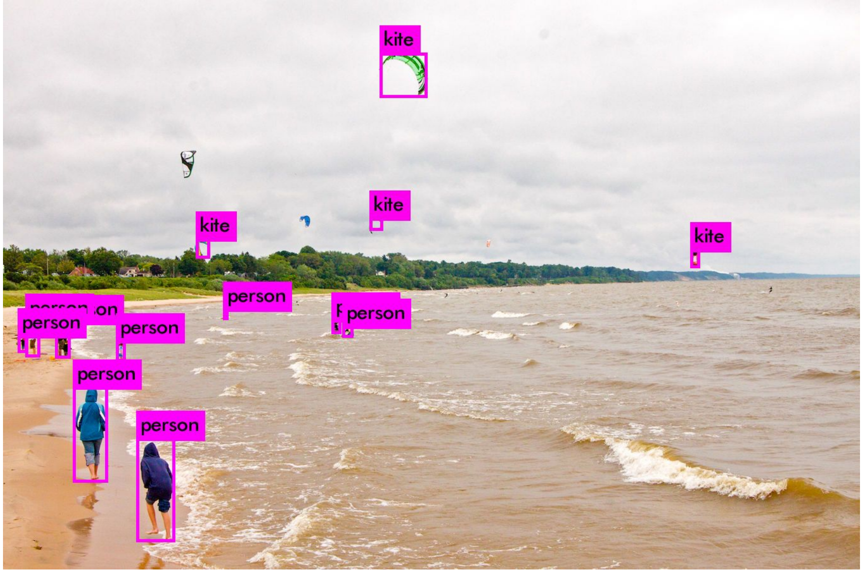
Firstly we will have the darknet detection and then ours.

Kite.jpg

Darknet *(original)*

data/kite.jpg: Predicted in 12.058297 seconds.

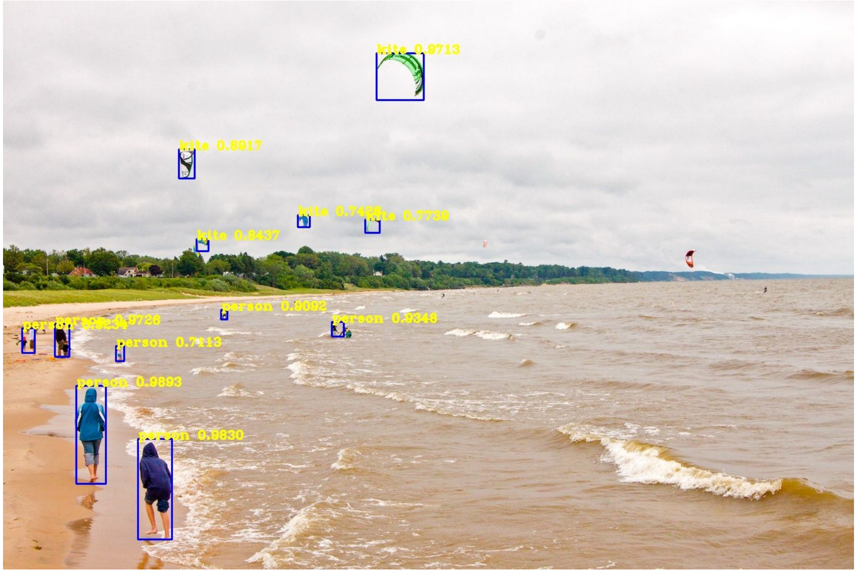
kite: 99% | kite: 84%
kite: 80% | kite: 73%
person: 100%
person: 100%
person: 97%
person: 96%
person: 95%
person: 91%
person: 88%
person: 85%
person: 52%



Re-Implementation *(custom)*

data/kite.jpg: Predicted in 0.882436 seconds.

kite: 97 % | kite: 89 %
kite: 84 % | kite: 77 %
kite: 0.74 %
person: 0.99 %
person: 0.98 %
person: 0.97 %
person: 0.93 %
person: 0.92 %
person: 0.91 %
person: 0.71 %

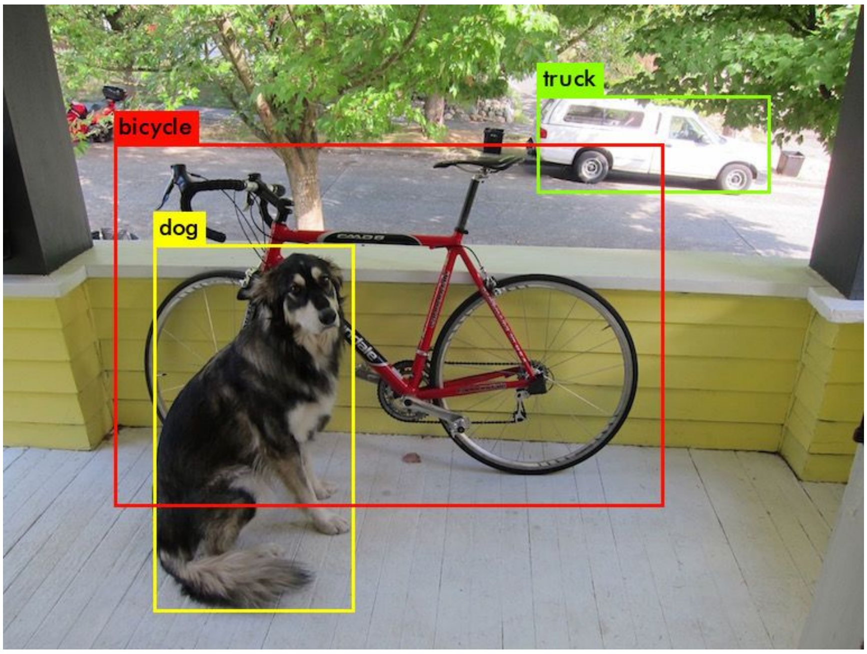


Dog.jpg

Darknet *(original)*

data/dog.jpg:
Predicted in 10.588475 seconds.

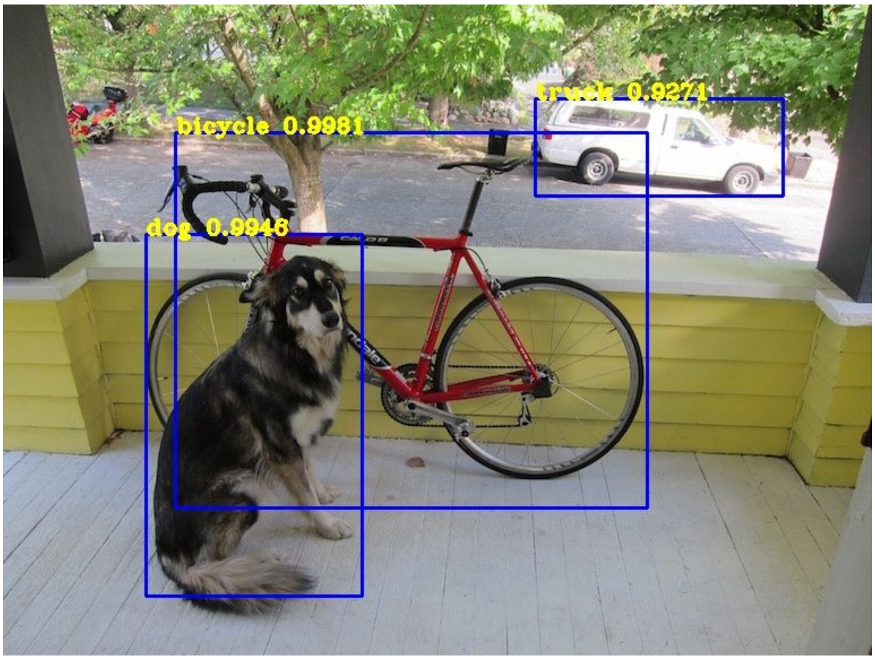
dog: 100%
truck: 92%
bicycle: 99%



Re-Implementation *(custom)*

data/dog.jpg:
0.855153 seconds.

bicycle: 100 %
truck: 93 %
dog: 99 %

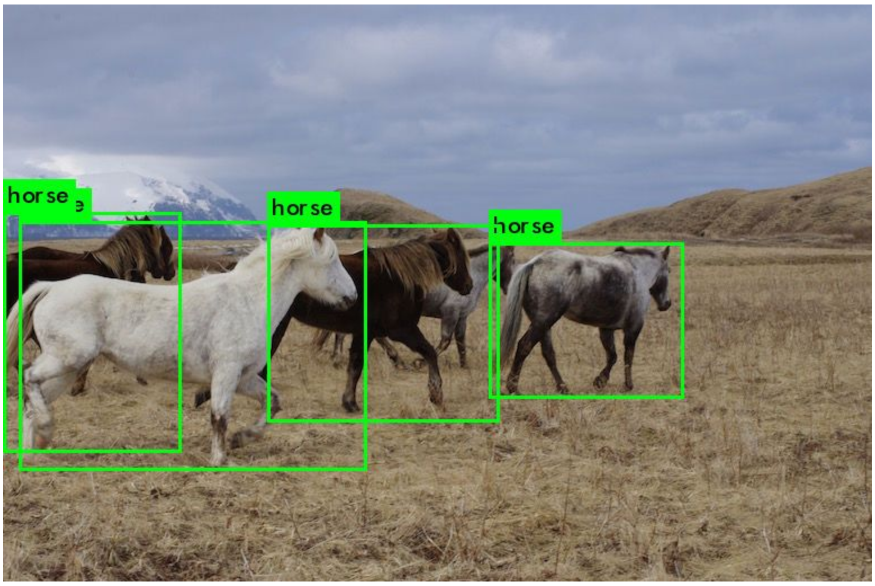


Horses.jpg

Darknet *(original)*

data/horses.jpg:
Predicted in 12.004588 seconds.

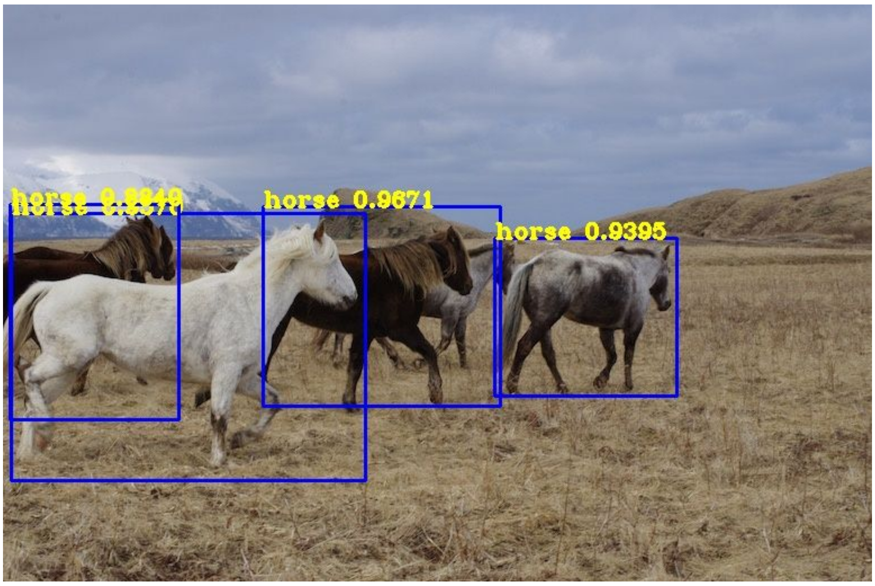
horse: 100%
horse: 100%
horse: 96%
horse: 95%



Re-Implementation *(custom)*

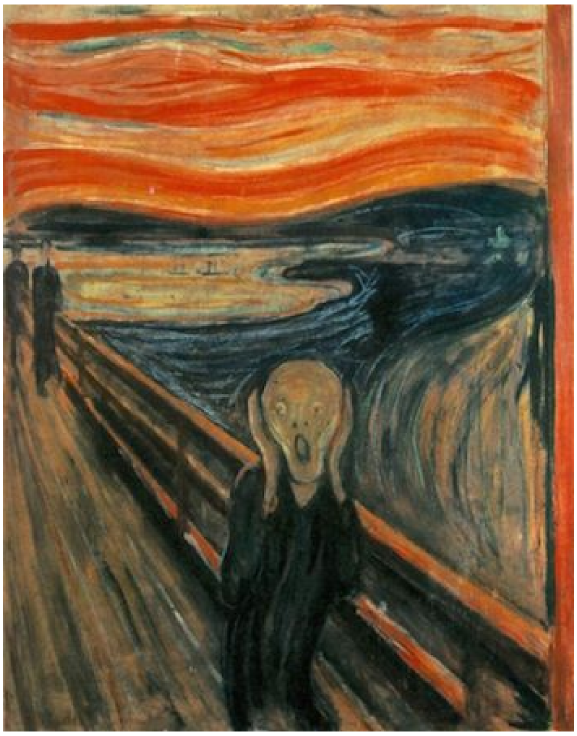
data/horses.jpg:
Predicted in 0.869875 seconds.

horse: 97 %
horse: 96 %
horse: 94 %
horse: 88 %

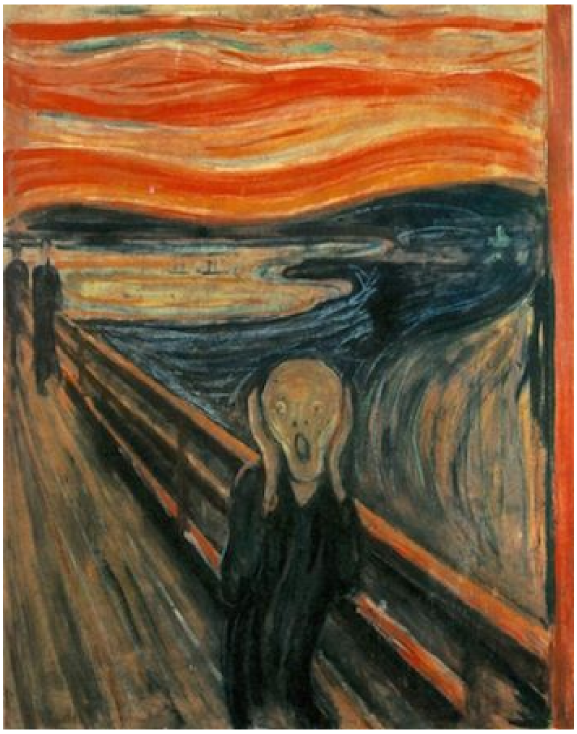


Scream.jpg

Re-Implementation *(custom)*



Darknet *(original)*



No detection case scenario

CONCLUSION

The results obtained are close to equal to the original YOLO algorithm for the COCO Dataset, and it's mainly related to the usage of the common pre-trained weights. Bounding boxes are a bit more refined on Darknet but this is linked to the proprietary methods utilized on the predicted bounding boxes. Nonetheless we outspeed by a good margin darknet, even though from their source code it is not precisely documented what specific processes are timed.

We are glad that we decided to reproduce from scratch the whole Model since this gave us a much deeper and complete understanding on how Neural Networks work, specifically on how to handle tensors and using them in order to compute the detection process. The task was definitely challenging considering the intricacies of the Model and our prior knowledge of the Tensorflow library, but by following a methodological approach we managed to successfully complete our task.

Finally, our implementation is not only restricted to the COCO Dataset since it can be used by anyone that has already computed weights for their specific Dataset, needing only to import labels and images, and tweak some hyperparameters such as number of classes and thresholds.

REFERENCES

1. https://arxiv.org/pdf/1506.02640.pdf
2. https://arxiv.org/pdf/1612.08242v1.pdf
3. https://arxiv.org/pdf/1804.02767v1.pdf