Realtime Object Detection through Artificial Intelligence

By Pulkit Mehta

[https://www.linkedin.com/in/ipulkitmehta/ (https://www.linkedin.com/in/ipulkitmehta/)](https://www.linkedin.com/in/ipulkitmehta/)

Getting Started

This guide will help you to implement Object Detection through Deep-Transfer Learning. The object localization task has been done by YOLO algorithm which means 'You Only Look Once'. The code has been implemented with raw TensorFlow without any high level abstraction API in order to get better scalability and handling.

GitHub: [https://github.com/pulkitmehta/ (https://github.com/pulkitmehta/)](https://github.com/pulkitmehta/)

You can download the project from [https://github.com/pulkitmehta/Realtime-Object-Detection (https://github.com/pulkitmehta/Realtime-Object-Detection)](https://github.com/pulkitmehta/Realtime-Object-Detection)

I have used DarkNet53 model for this purpose.

LICENSE

Import necessary Libraries

Let us do a quick version check:

In [37]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import cv2
print(tf.__version__)
```

```
1.13.1
```

If your version is >= 2, then remove the (''') and run the following code

In [2]:
```
'''
## Remove the comments
## Run only when version >= 2
del tf    ## delete existing TensorFlow instance (version>=2)
import tensorflow.compat.v1 as tf  ## Enable Backward compatibility for v1 code
'''
print()
```

- Now we initialize important Hyperparameters.

In [3]:
```
_BATCH_NORM_DECAY = 0.9
_BATCH_NORM_EPSILON = 1e-05
_LEAKY_RELU = 0.1
_ANCHORS = [(10, 13), (16, 30), (33, 23),
            (30, 61), (62, 45), (59, 119),
            (116, 90), (156, 198), (373, 326)]
_MODEL_SIZE = (416, 416)
```

A helper function to read class name file

In [21]:
```
def load_class_names(file_name):
    """Returns a list of class names read from `file_name`."""
    with open(file_name, 'r') as f:
        class_names = f.read().splitlines()
    return class_names
```

In [25]:
```python
"""A Preview of 10/80 types of objects."""
load_class_names("./model_data/coco_classes.txt")[:10]
```

Out[25]:
```
['person',
 'bicycle',
 'car',
 'motorbike',
 'aeroplane',
 'bus',
 'train',
 'truck',
 'boat',
 'traffic light']
```

Transfer Learning

- I have built the model architecture in native TensorFlow.
- Then we assign the weights file to our untrained model.
- You can skip till Here but i would recommend you going through below section for better understanding.
- We would be using DarkNet Model architecture.

Important! You will not get weights file from this repository. Download it from here (https://pjreddie.com/media/files/yolov3.weights) and place the weight file in "./model_data/" directory.

Important Functions

Batch Normalization

- Apply Batch Normalization on inputs

In [4]:
```python
def batch_norm(inputs, training, data_format):
    return tf.layers.batch_normalization(
        inputs=inputs, axis=1 if data_format == 'channels_first' else 3,
        momentum=_BATCH_NORM_DECAY, epsilon=_BATCH_NORM_EPSILON,
        scale=True, training=training)
```

Fixed Padding

- This operation pads a `inputs` according to the `paddings` required.

In [5]:
```python
def fixed_padding(inputs, kernel_size, data_format):
    pad_total = kernel_size - 1
    pad_start = pad_total // 2
    pad_end = pad_total - pad_start

    if data_format == 'channels_first':
        padded_inputs = tf.pad(inputs, [[0, 0], [0, 0],
                                        [pad_start, pad_end],
                                        [pad_start, pad_end]])
    else:
        padded_inputs = tf.pad(inputs, [[0, 0], [pad_start, pad_end],
                                        [pad_start, pad_end], [0, 0]])
    return padded_inputs
```

Convolution Layer with fixed padding

- Returns Convolution layer with fixed padding if strides are > 1 keeping the dimentions same.

In [6]:
```
def conv_padding(inputs, filters, kernel_size, data_format, strides=1):

    if strides > 1:
        inputs = fixed_padding(inputs, kernel_size, data_format)

    return tf.layers.conv2d(inputs=inputs, filters=filters, kernel_size=kernel_size,
                            strides=strides, padding=('SAME' if strides == 1 else 'VALID'),
                            use_bias=False, data_format=data_format)
```

Residual Block for DarkNet Architecture

In [7]:
```
def residual_block(inputs, filters, training, data_format, strides=1):

    skip_connection = inputs
    inputs = conv_padding(inputs, filters=filters, kernel_size=1, strides=strides,data_format=data_forma
t)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)
    inputs = conv_padding(inputs, filters=2 * filters, kernel_size=3, strides=strides,data_format=data_f
ormat)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)
    inputs += skip_connection

    return inputs
```

DarkNet Architecture

In [8]:
```
def dNetArch(inputs, training, data_format):
    inputs = conv_padding(inputs, filters=32, kernel_size=3,
                              data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)
    inputs = conv_padding(inputs, filters=64, kernel_size=3,
                              strides=2, data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    inputs = residual_block(inputs, filters=32, training=training,
                                  data_format=data_format)

    inputs = conv_padding(inputs, filters=128, kernel_size=3,
                              strides=2, data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    for _ in range(2):
        inputs = residual_block(inputs, filters=64,
                                      training=training,
                                      data_format=data_format)

    inputs = conv_padding(inputs, filters=256, kernel_size=3,
                              strides=2, data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    for _ in range(8):
        inputs = residual_block(inputs, filters=128,
                                      training=training,
                                      data_format=data_format)

    r1 = inputs

    inputs = conv_padding(inputs, filters=512, kernel_size=3,
                              strides=2, data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    for _ in range(8):
        inputs = residual_block(inputs, filters=256,
                                      training=training,
                                      data_format=data_format)

    r2 = inputs

    inputs = conv_padding(inputs, filters=1024, kernel_size=3,
                              strides=2, data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    for _ in range(4):
        inputs = residual_block(inputs, filters=512,
                                      training=training,
                                      data_format=data_format)

    return r1, r2, inputs
```

DarkNet Extension

In [9]:
```
def conv_ext(inputs, filters, training, data_format):
    inputs = conv_padding(inputs, filters=filters, kernel_size=1,
                          data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    inputs = conv_padding(inputs, filters=2 * filters, kernel_size=3,
                          data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    inputs = conv_padding(inputs, filters=filters, kernel_size=1,
                          data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    inputs = conv_padding(inputs, filters=2 * filters, kernel_size=3,
                          data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    inputs = conv_padding(inputs, filters=filters, kernel_size=1,
                          data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    r = inputs

    inputs = conv_padding(inputs, filters=2 * filters, kernel_size=3,
                          data_format=data_format)
    inputs = batch_norm(inputs, training=training, data_format=data_format)
    inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)

    return r, inputs
```

Mouth of NN

- This part is where our main Object Detection paradigm of YOLO (You Only Look Once) comes.
- Now let us build a mouth which will be our final detection layer for object detection
- The function will take following arguments:
  - inputs: Tensor input.
  - n_classes: Number of labels.
  - anchor_boxes: A list of anchor sizes.
  - Img_size: The input size of the model.
  - data_format: The input format.
- If you find difficulty in understanding, I would recommend looking Scholarly articles and videos on YOLO Algorithm

In [10]:
```python
def mouth(inputs, n_classes, anchor_boxes, Img_size, data_format):
    '''
    here we have number of anchor boxes
    '''
    n_anchor_boxes = len(anchor_boxes)

    '''
    Make a convolution layer with no. of filters as no. of anchor boxes times (classes+5)
    You can find the theory on yolo research paper.
    '''

    inputs = tf.layers.conv2d(inputs, filters=n_anchor_boxes * (5 + n_classes),
                              kernel_size=1, strides=1, use_bias=True,
                              data_format=data_format)

    shape = inputs.get_shape().as_list()

    '''
    Now we decide to make a grid overlay for our image, So we will find the shape of grid as follows.
    '''

    grid_shape = shape[2:4] if data_format == 'channels_first' else shape[1:3]

    if data_format == 'channels_first':
        inputs = tf.transpose(inputs, [0, 2, 3, 1])

    '''
    Now I reformat the shape as follows would be a better practice and further more implementations
    according to research paper.
    '''

    inputs = tf.reshape(inputs, [-1, n_anchor_boxes * grid_shape[0] * grid_shape[1],
                                 5 + n_classes])

    strides = (Img_size[0] // grid_shape[0], Img_size[1] // grid_shape[1])

    box_c, box_shapes, confidence, classes = tf.split(inputs, [2, 2, 1, n_classes], axis=-1)

    '''
    Here I start building the mesh grid as per our algorithm.
    '''

    x = tf.range(grid_shape[0], dtype=tf.float32)
    y = tf.range(grid_shape[1], dtype=tf.float32)
    x_off, y_off = tf.meshgrid(x, y)
    x_off = tf.reshape(x_off, (-1, 1))
    y_off = tf.reshape(y_off, (-1, 1))
    x_y_off = tf.concat([x_off, y_off], axis=-1)
    x_y_off = tf.tile(x_y_off, [1, n_anchor_boxes])
    x_y_off = tf.reshape(x_y_off, [1, -1, 2])
    box_c = tf.nn.sigmoid(box_c)
    box_c = (box_c + x_y_off) * strides

    anchor_boxes = tf.tile(anchor_boxes, [grid_shape[0] * grid_shape[1], 1])
    box_shapes = tf.exp(box_shapes) * tf.to_float(anchor_boxes)

    confidence = tf.nn.sigmoid(confidence)

    classes = tf.nn.sigmoid(classes)

    inputs = tf.concat([box_c, box_shapes,
                        confidence, classes], axis=-1)

    return inputs
```

- This function will upsample the image using nearest neighbor interpolation.

In [11]:
```
def Upsample(inputs, outputShape, data_format):
    if data_format == 'channels_first':
        inputs = tf.transpose(inputs, [0, 2, 3, 1])
        H = outputShape[3]
        W = outputShape[2]
    else:
        H = outputShape[2]
        W = outputShape[1]

    inputs = tf.image.resize_nearest_neighbor(inputs, (H, W))

    if data_format == 'channels_first':
        inputs = tf.transpose(inputs, [0, 3, 1, 2])

    return inputs
```

- Computes top left and bottom right points of the boxes with given center of box , Height and Width.

In [12]:
```
def make_boxes(inputs):
    c_x, c_y, W, H, confidence, classes = \
        tf.split(inputs, [1, 1, 1, 1, 1, -1], axis=-1)

    TL_x = c_x - W / 2
    TL_y = c_y - H / 2
    BR_x = c_x + W / 2
    BR_y = c_y + H / 2

    boxes = tf.concat([TL_x, TL_y,
                       BR_x, BR_y,
                       confidence, classes], axis=-1)

    return boxes
```

Non Max Suppression

Since we have generated so many bounding boxes for a single images we would have to choose a single BEST bounding box for our object(s). So we calculate Intersection Over Union rator of boxes and take a confidence threshold and IOU threshold to suppress the ones with low. Hence called NON MAX SUP. This is key part of model which can reflect senstivity by adjusting thresholds.

In [13]:
```python
def NMS(inputs, n_classes, max_output_size, iou_threshold,
                        confidence_threshold):
    """

    Args:
        inputs: Tensor input.
        n_classes: Number of classes.
        max_output_size: Max number of boxes to be selected for each class.
        iou_threshold: Threshold for the IOU.
        confidence_threshold: Threshold for the confidence score.
    Returns:
        A list containing class-to-boxes dictionaries
            for each sample in the batch.
    """


    batch = tf.unstack(inputs)
    boxes_dicts = []
    for boxes in batch:
        boxes = tf.boolean_mask(boxes, boxes[:, 4] > confidence_threshold)
        classes = tf.argmax(boxes[:, 5:], axis=-1)
        classes = tf.expand_dims(tf.to_float(classes), axis=-1)
        boxes = tf.concat([boxes[:, :5], classes], axis=-1)

        boxes_dict = dict()
        for cls in range(n_classes):
            mask = tf.equal(boxes[:, 5], cls)
            mask_shape = mask.get_shape()
            if mask_shape.ndims != 0:
                class_boxes = tf.boolean_mask(boxes, mask)
                boxes_coords, boxes_conf_scores, _ = tf.split(class_boxes,
                                                              [4, 1, -1],
                                                              axis=-1)
                boxes_conf_scores = tf.reshape(boxes_conf_scores, [-1])
                indices = tf.image.non_max_suppression(boxes_coords,
                                                       boxes_conf_scores,
                                                       max_output_size,
                                                       iou_threshold)
                class_boxes = tf.gather(class_boxes, indices)
                boxes_dict[cls] = class_boxes[:, :5]

        boxes_dicts.append(boxes_dict)

    return boxes_dicts
```

Now its time to contain whole model for producing Instances later.

In [32]:
```python
class Model:

    def __init__(self, n_classes, input_size, max_output_size, iou_threshold,
                 confidence_threshold, data_format=None):
        """
        Args:
            n_classes: Number of class labels.
            input_size: The input size of the model.
            max_output_size: Max number of boxes to be selected for each class.
            iou_threshold: Threshold for the IOU.
            confidence_threshold: Threshold for the confidence score.
            data_format: The input format.
        """
        if not data_format:
            if tf.test.is_built_with_cuda():
                data_format = 'channels_first'
            else:
                data_format = 'channels_last'

        self.n_classes = n_classes
        self.input_size = input_size
        self.max_output_size = max_output_size
        self.iou_threshold = iou_threshold
        self.confidence_threshold = confidence_threshold
        self.data_format = data_format

    def __call__(self, inputs, training):
        """
        Add operations to detect boxes for a batch of input images.

        Args:
            inputs: A Tensor representing a batch of input images.
```

```python
            training: A boolean, whether to use in training or inference mode.

        Returns:
            A list containing class-to-boxes dictionaries
                for each sample in the batch.
        """

        with tf.variable_scope('yolo_v3_model'):
            if self.data_format == 'channels_first':
                inputs = tf.transpose(inputs, [0, 3, 1, 2])

            inputs = inputs / 255

            r1, r2, inputs = dNetArch(inputs, training=training,
                                               data_format=self.data_format)

            r, inputs = conv_ext(
                inputs, filters=512, training=training,
                data_format=self.data_format)
            detect1 = mouth(inputs, n_classes=self.n_classes,
                                 anchor_boxes=_ANCHORS[6:9],
                                 Img_size=self.input_size,
                                 data_format=self.data_format)

            inputs = conv_padding(r, filters=256, kernel_size=1,
                                          data_format=self.data_format)
            inputs = batch_norm(inputs, training=training,
                                data_format=self.data_format)
            inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)
            Upsample_size = r2.get_shape().as_list()
            inputs = Upsample(inputs, outputShape=Upsample_size,
                              data_format=self.data_format)
            axis = 1 if self.data_format == 'channels_first' else 3
            inputs = tf.concat([inputs, r2], axis=axis)
            r, inputs = conv_ext(
                inputs, filters=256, training=training,
                data_format=self.data_format)
            detect2 = mouth(inputs, n_classes=self.n_classes,
                                 anchor_boxes=_ANCHORS[3:6],
                                 Img_size=self.input_size,
                                 data_format=self.data_format)

            inputs = conv_padding(r, filters=128, kernel_size=1,
                                          data_format=self.data_format)
            inputs = batch_norm(inputs, training=training,
                                data_format=self.data_format)
            inputs = tf.nn.leaky_relu(inputs, alpha=_LEAKY_RELU)
            Upsample_size = r1.get_shape().as_list()
            inputs = Upsample(inputs, outputShape=Upsample_size,
                              data_format=self.data_format)
            inputs = tf.concat([inputs, r1], axis=axis)
            r, inputs = conv_ext(
                inputs, filters=128, training=training,
                data_format=self.data_format)
            detect3 = mouth(inputs, n_classes=self.n_classes,
                                 anchor_boxes=_ANCHORS[0:3],
                                 Img_size=self.input_size,
                                 data_format=self.data_format)

            inputs = tf.concat([detect1, detect2, detect3], axis=1)

            inputs = make_boxes(inputs)

            boxes_dicts = NMS(
                inputs, n_classes=self.n_classes,
                max_output_size=self.max_output_size,
                iou_threshold=self.iou_threshold,
                confidence_threshold=self.confidence_threshold)

            return boxes_dicts
```

Here I define a function to assign model weights stored in ./model_data/ directory

In [33]:
```
def load_model_weights(vars, file_name):

    with open(file_name, "rb") as f:
        # Skip first 5 values containing irrelevant info
        np.fromfile(f, dtype=np.int32, count=5)
        weights = np.fromfile(f, dtype=np.float32)

        assign_ops = []
        ptr = 0

        # Load weights for Darknet part.
        # Each convolution layer has batch normalization.
        for i in range(52):
            conv_var = vars[5 * i]
            gamma, beta, mean, variance = vars[5 * i + 1:5 * i + 5]
            batch_norm_vars = [beta, gamma, mean, variance]

            for var in batch_norm_vars:
                shape = var.shape.as_list()
                num_params = np.prod(shape)
                var_weights = weights[ptr:ptr + num_params].reshape(shape)
                ptr += num_params
                assign_ops.append(tf.assign(var, var_weights))

            shape = conv_var.shape.as_list()
            num_params = np.prod(shape)
            var_weights = weights[ptr:ptr + num_params].reshape(
                (shape[3], shape[2], shape[0], shape[1]))
            var_weights = np.transpose(var_weights, (2, 3, 1, 0))
            ptr += num_params
            assign_ops.append(tf.assign(conv_var, var_weights))

        # Loading weights for Yolo part.
        # 7th, 15th and 23rd convolution layer has biases and no batch norm.
        ranges = [range(0, 6), range(6, 13), range(13, 20)]
        unnormalized = [6, 13, 20]
        for j in range(3):
            for i in ranges[j]:
                current = 52 * 5 + 5 * i + j * 2
                conv_var = vars[current]
                gamma, beta, mean, variance =  \
                    vars[current + 1:current + 5]
                batch_norm_vars = [beta, gamma, mean, variance]

                for var in batch_norm_vars:
                    shape = var.shape.as_list()
                    num_params = np.prod(shape)
                    var_weights = weights[ptr:ptr + num_params].reshape(shape)
                    ptr += num_params
                    assign_ops.append(tf.assign(var, var_weights))

                shape = conv_var.shape.as_list()
                num_params = np.prod(shape)
                var_weights = weights[ptr:ptr + num_params].reshape(
                    (shape[3], shape[2], shape[0], shape[1]))
                var_weights = np.transpose(var_weights, (2, 3, 1, 0))
                ptr += num_params
                assign_ops.append(tf.assign(conv_var, var_weights))

            bias = vars[52 * 5 + unnormalized[j] * 5 + j * 2 + 1]
            shape = bias.shape.as_list()
            num_params = np.prod(shape)
            var_weights = weights[ptr:ptr + num_params].reshape(shape)
            ptr += num_params
            assign_ops.append(tf.assign(bias, var_weights))

            conv_var = vars[52 * 5 + unnormalized[j] * 5 + j * 2]
            shape = conv_var.shape.as_list()
            num_params = np.prod(shape)
            var_weights = weights[ptr:ptr + num_params].reshape(
                (shape[3], shape[2], shape[0], shape[1]))
            var_weights = np.transpose(var_weights, (2, 3, 1, 0))
            ptr += num_params
            assign_ops.append(tf.assign(conv_var, var_weights))

    return assign_ops
```

This function will Draw boxes on a Single Image

In [45]:
```python
def draw_boxes(image,result, class_names):
    result=result[0]
    print(image.shape)

    """This is our resizing factor which will resize the box dimentions for the original image"""
    rx= image.shape[1]/_MODEL_SIZE[1]
    ry= image.shape[0]/_MODEL_SIZE[0]


    for cls in result:
        color = np.random.randint(120,256,3)
        class_name= class_names[cls]
        color=(int(color[0]),int(color[1]),int(color[2]))
        cls_boxes= result[cls]

        for box in cls_boxes:

            crd= box[:4]
            confidence= str(int(box[4]*100))

            #(coordinates)

            x1=int(crd[0]*rx)
            y1=int(crd[1]*ry)
            x2=int(crd[2]*rx)
            y2=int(crd[3]*ry)

            '''print(x1,y1,x2,y2)
            print(cls)
            print(color)'''


            thickness= max(image.shape[0],image.shape[1])//300

            ## main rectangle
            image=cv2.rectangle(image,(x1,y1),(x2,y2),color,thickness=thickness)

            ## text with background rectangle
            font_weight=0.5
            h=int(y1-(font_weight*50))

            image=cv2.rectangle(image, (x1-thickness//2,h),(x2+thickness//2,y1),color,-1)

            text= class_name+" "+confidence+"%"



            image=cv2.putText(image,
                              text,
                              (x1+thickness,y1-thickness-5),
                              cv2.FONT_HERSHEY_SIMPLEX,font_weight,(0,0,0), thickness=2)
    ## This will show image in notebook as well as save in output directory.
    plt.figure(figsize=(20,10))
    plt.imshow(image)
    cv2.imwrite("./output/output.png", image)
    plt.show()
    plt.close()
```

Here I instantiate our model.

- Also we load class names present in ./model_data/
- This model will detect 80 types of Common Objects of Context

In [39]:
```
tf.reset_default_graph()
batch_size = 1
class_names = load_class_names('./model_data/coco_classes.txt')
n_classes = len(class_names)
"""Maximum objects to detect in image"""
max_output_size = 100
iou_threshold = 0.5
confidence_threshold = 0.3

model = Model(n_classes=n_classes, input_size=_MODEL_SIZE,
                max_output_size=max_output_size,
                iou_threshold=iou_threshold,
                confidence_threshold=confidence_threshold)

inputs = tf.placeholder(tf.float32, [batch_size, 416, 416, 3])

detections = model(inputs, training=False)

model_vars = tf.global_variables(scope='yolo_v3_model')
assign_ops = load_model_weights(model_vars, './model_data/yolov3.weights')
sess= tf.Session()
wts=sess.run(assign_ops)
```

Now we are ready to go for a Single Image

In [40]:
```
del wts ## to free memory
```

In [42]:
```
img=cv2.imread("./inputs/got.png")
testimg= cv2.resize(img, _MODEL_SIZE)

testimg=testimg.reshape(1,416,416,3)

detection_result = sess.run(detections, feed_dict={inputs: testimg})
```

The detection result will contain box dimensions with respect to scaled down imge

In [43]:
```
detection_result
```

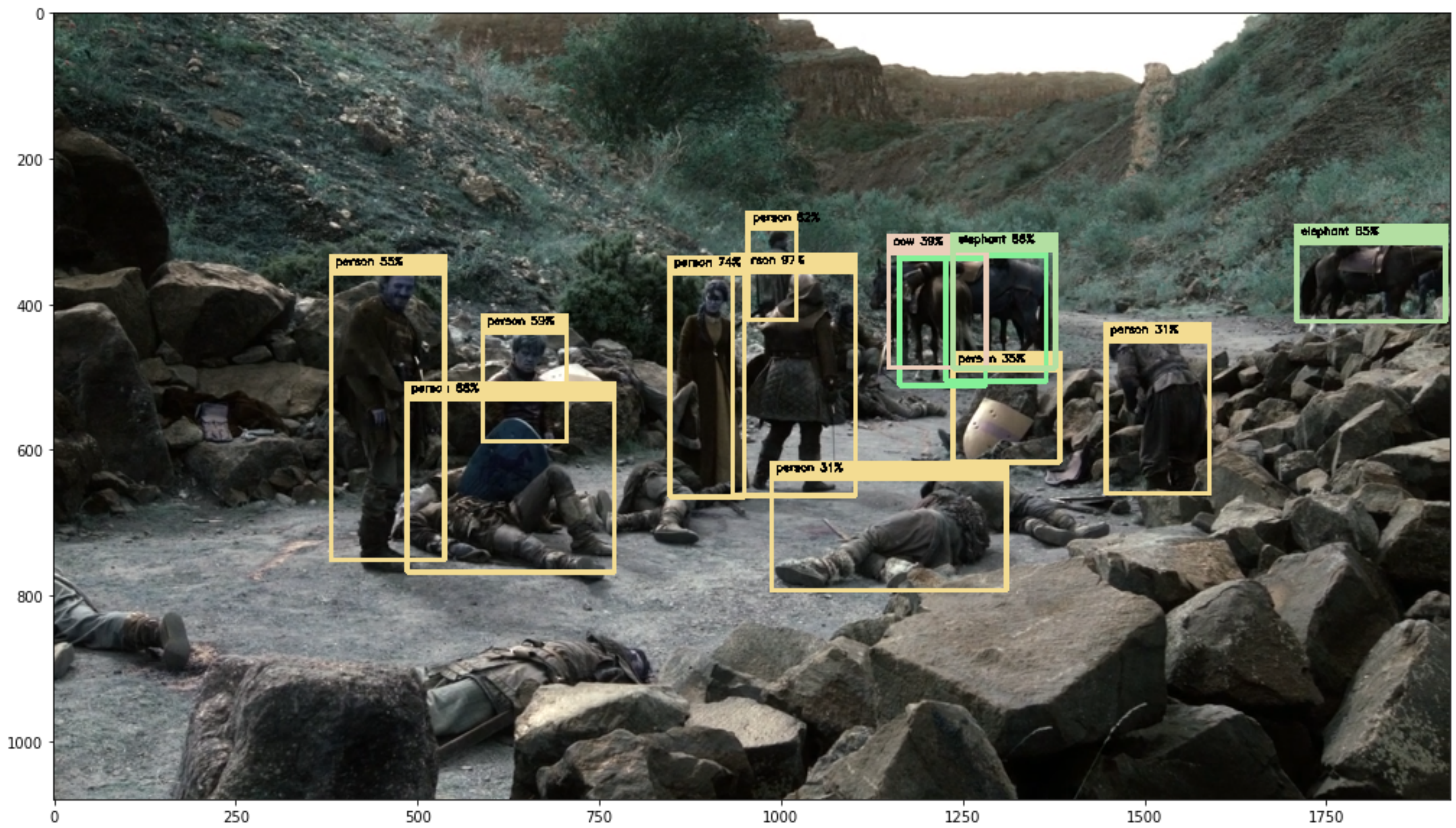
Out[43]:
```
[{0: array([[2.0237677e+02, 1.3685815e+02, 2.3877267e+02, 2.5540527e+02,
          9.7004318e-01],
         [1.0544134e+02, 2.0482413e+02, 1.6711565e+02, 2.9611429e+02,
          8.8535744e-01],
         [2.0698827e+02, 1.1416114e+02, 2.2121733e+02, 1.6231865e+02,
          8.2599533e-01],
         [1.8367609e+02, 1.3791714e+02, 2.0603320e+02, 2.5647916e+02,
          7.4818665e-01],
         [1.2804594e+02, 1.6885254e+02, 1.5276927e+02, 2.2672586e+02,
          5.9588128e-01],
         [8.2968224e+01, 1.3771484e+02, 1.1661645e+02, 2.8951187e+02,
          5.5656725e-01],
         [2.6803238e+02, 1.8876523e+02, 2.9969772e+02, 2.3779337e+02,
          3.5091272e-01],
         [2.1390263e+02, 2.4659766e+02, 2.8404340e+02, 3.0567346e+02,
          3.1334639e-01],
         [3.1359583e+02, 1.7356592e+02, 3.4428925e+02, 2.5431342e+02,
          3.1136131e-01]], dtype=float32),
  1: array([], shape=(0, 5), dtype=float32),
  2: array([], shape=(0, 5), dtype=float32),
  3: array([], shape=(0, 5), dtype=float32),
  4: array([], shape=(0, 5), dtype=float32),
  5: array([], shape=(0, 5), dtype=float32),
  6: array([], shape=(0, 5), dtype=float32),
  7: array([], shape=(0, 5), dtype=float32),
  8: array([], shape=(0, 5), dtype=float32),
  9: array([], shape=(0, 5), dtype=float32),
  10: array([], shape=(0, 5), dtype=float32),
  11: array([], shape=(0, 5), dtype=float32),
  12: array([], shape=(0, 5), dtype=float32),
  13: array([], shape=(0, 5), dtype=float32),
  14: array([], shape=(0, 5), dtype=float32),
  15: array([], shape=(0, 5), dtype=float32),
  16: array([], shape=(0, 5), dtype=float32),
  17: array([[252.05641   , 130.45464   , 277.72586   , 197.34694   ,
            0.7857561 ],
         [265.8794    , 128.59282   , 295.78735   , 195.15486   ,
            0.42884928]], dtype=float32),
```

```
18: array([], shape=(0, 5), dtype=float32),
19: array([[248.82092   , 126.76281   , 277.95135   , 187.779     ,
          0.39126468]], dtype=float32),
20: array([[268.02777  , 125.836624 , 298.43042  , 188.2632   ,   0.8614883],
       [370.37167  , 121.56887  , 414.82553  , 163.06174  ,   0.8529862]],
      dtype=float32),
21: array([], shape=(0, 5), dtype=float32),
22: array([], shape=(0, 5), dtype=float32),
23: array([], shape=(0, 5), dtype=float32),
24: array([], shape=(0, 5), dtype=float32),
25: array([], shape=(0, 5), dtype=float32),
26: array([], shape=(0, 5), dtype=float32),
27: array([], shape=(0, 5), dtype=float32),
28: array([], shape=(0, 5), dtype=float32),
29: array([], shape=(0, 5), dtype=float32),
30: array([], shape=(0, 5), dtype=float32),
31: array([], shape=(0, 5), dtype=float32),
32: array([], shape=(0, 5), dtype=float32),
33: array([], shape=(0, 5), dtype=float32),
34: array([], shape=(0, 5), dtype=float32),
35: array([], shape=(0, 5), dtype=float32),
36: array([], shape=(0, 5), dtype=float32),
37: array([], shape=(0, 5), dtype=float32),
38: array([], shape=(0, 5), dtype=float32),
39: array([], shape=(0, 5), dtype=float32),
40: array([], shape=(0, 5), dtype=float32),
41: array([], shape=(0, 5), dtype=float32),
42: array([], shape=(0, 5), dtype=float32),
43: array([], shape=(0, 5), dtype=float32),
44: array([], shape=(0, 5), dtype=float32),
45: array([], shape=(0, 5), dtype=float32),
46: array([], shape=(0, 5), dtype=float32),
47: array([], shape=(0, 5), dtype=float32),
48: array([], shape=(0, 5), dtype=float32),
49: array([], shape=(0, 5), dtype=float32),
50: array([], shape=(0, 5), dtype=float32),
51: array([], shape=(0, 5), dtype=float32),
52: array([], shape=(0, 5), dtype=float32),
53: array([], shape=(0, 5), dtype=float32),
54: array([], shape=(0, 5), dtype=float32),
55: array([], shape=(0, 5), dtype=float32),
56: array([], shape=(0, 5), dtype=float32),
57: array([], shape=(0, 5), dtype=float32),
58: array([], shape=(0, 5), dtype=float32),
59: array([], shape=(0, 5), dtype=float32),
60: array([], shape=(0, 5), dtype=float32),
61: array([], shape=(0, 5), dtype=float32),
62: array([], shape=(0, 5), dtype=float32),
63: array([], shape=(0, 5), dtype=float32),
64: array([], shape=(0, 5), dtype=float32),
65: array([], shape=(0, 5), dtype=float32),
66: array([], shape=(0, 5), dtype=float32),
67: array([], shape=(0, 5), dtype=float32),
68: array([], shape=(0, 5), dtype=float32),
69: array([], shape=(0, 5), dtype=float32),
70: array([], shape=(0, 5), dtype=float32),
71: array([], shape=(0, 5), dtype=float32),
72: array([], shape=(0, 5), dtype=float32),
73: array([], shape=(0, 5), dtype=float32),
74: array([], shape=(0, 5), dtype=float32),
75: array([], shape=(0, 5), dtype=float32),
76: array([], shape=(0, 5), dtype=float32),
77: array([], shape=(0, 5), dtype=float32),
78: array([], shape=(0, 5), dtype=float32),
79: array([], shape=(0, 5), dtype=float32)}]
```

- Now we will use draw_boxes function which will upscale these boxes and impose on the original image

```
In [46]: draw_boxes(img, detection_result, class_names)
```

```
(1080, 1920, 3)
```



Rendering a Video

- This function will render box in a single frame

In [47]:
```
def draw_boxes_video(image,result, class_names, colors):
    result=result[0]
    rx= image.shape[1]/_MODEL_SIZE[1]
    ry= image.shape[0]/_MODEL_SIZE[0]

    for cls in result:
        class_name= class_names[cls]
        color=colors[cls]
        cls_boxes= result[cls]

        for box in cls_boxes:

            crd= box[:4]
            confidence= str(int(box[4]*100))

            #(coordinates)

            x1=int(crd[0]*rx)
            y1=int(crd[1]*ry)
            x2=int(crd[2]*rx)
            y2=int(crd[3]*ry)

            '''print(x1,y1,x2,y2)
            print(cls)
            print(color)'''


            thickness= max(image.shape[0],image.shape[1])//500

            ## main rectangle
            image=cv2.rectangle(image,(x1,y1),(x2,y2),color,thickness=thickness)

            ## text
            font_weight=0.3
            h=int(y1-(font_weight*50))

            image=cv2.rectangle(image, (x1-thickness//2,h),(x2+thickness//2,y1),color,-1)

            text= class_name+" "+confidence+"%"



            image=cv2.putText(image,
                              text,
                              (x1+thickness,y1-thickness-5),
                              cv2.FONT_HERSHEY_SIMPLEX,font_weight,(0,0,0), thickness=2)


    return image
```

This function will render full video in output directory

In [49]:
```
def video(frames):
    total=frames.shape[0]
    class_color=dict()
    for i in range(80):

        color = np.random.randint(120,256,3)
        color=(int(color[0]),int(color[1]),int(color[2]))
        class_color[i]=color

    out= cv2.VideoWriter("./output/video.avi", cv2.VideoWriter_fourcc('M','J','P','G'),30,(854,470))
    i=0
    for frame in frames:
        i=i+1
        percentage=(i/total)*100

        if percentage
        print("Frame: ",i)
        testimg= cv2.resize(frame, _MODEL_SIZE)

        testimg=testimg.reshape(1,416,416,3)

        detection_result = sess.run(detections, feed_dict={inputs: testimg})

        frame= draw_boxes_video(frame,detection_result, class_names, class_color)

        out.write(frame)

    out.release()
```

- Capture the original video and store all frames in array. Note Make sure to deallocate the memory when we do not need them because they will stack up your RAM and you will get memory error.

In [52]:
```
cap= cv2.VideoCapture("./inputs/videoplayback.mp4")
frames= []
while cap.isOpened():
    ret, frame = cap.read()
    if ret==False:
        break
    else:
        frames.append(frame)
frames=np.array(frames)
```

In [53]:
```
frames.shape
```

Out[53]: (2340, 470, 854, 3)

- Run this to render video

In [ ]:
```
video(frames)
cap.release()

## V-Important Deallocate memory to free your RAM
del cap

del frames
```

Now take a look at the rendered video.

--- END ---

In [ ]:
```
```