In [1]:

```
import os
import scipy.io
import scipy.misc
import numpy as np
import pandas as pd
import PIL
import struct
import cv2
from numpy import expand_dims
import tensorflow as tf
from skimage.transform import resize
from keras import backend as K
from keras.layers import Input, Lambda, Conv2D, BatchNormalization, LeakyReLU, ZeroPadding2D, UpSampling2D
from keras.models import load_model, Model
from keras.layers import add, concatenate
from tensorflow.keras.utils import load_img
from tensorflow.keras.utils import img_to_array
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
from matplotlib.patches import Rectangle
%matplotlib inline
```

In [2]:

```
class WeightReader:
        def __init__(self, weight_file):
                with open(weight_file, 'rb') as w_f:
                        major,  = struct.unpack('i', w_f.read(4))
                        minor,  = struct.unpack('i', w_f.read(4))
                        revision, = struct.unpack('i', w_f.read(4))
                        if (major*10 + minor) >= 2 and major < 1000 and minor < 1000:
                                w_f.read(8)
                        else:
                                w_f.read(4)
                        transpose = (major > 1000) or (minor > 1000)
                        binary = w_f.read()
                self.offset = 0
                self.all_weights = np.frombuffer(binary, dtype='float32')

        def read_bytes(self, size):
                self.offset = self.offset + size
                return self.all_weights[self.offset-size:self.offset]
```

```
        def load_weights(self, model):
                for i in range(106):
                        try:
                                conv_layer = model.get_layer('conv_' + str(i))
                                print("loading weights of convolution #" + str(i))
                                if i not in [81, 93, 105]:
                                        norm_layer = model.get_layer('bnorm_' + str(i))
                                        size = np.prod(norm_layer.get_weights()[0].shape)
                                        beta  = self.read_bytes(size) # bias
                                        gamma = self.read_bytes(size) # scale
                                        mean  = self.read_bytes(size) # mean
                                        var   = self.read_bytes(size) # variance
                                        weights = norm_layer.set_weights([gamma, beta, mean, var])
                                if len(conv_layer.get_weights()) > 1:
                                        bias   = self.read_bytes(np.prod(conv_layer.get_weights()[1].shape))
                                        kernel = self.read_bytes(np.prod(conv_layer.get_weights()[0].shape))
                                        kernel = kernel.reshape(list(reversed(conv_layer.get_weights()[0].shape)))
                                        kernel = kernel.transpose([2,3,1,0])
                                        conv_layer.set_weights([kernel, bias])
                                else:
                                        kernel = self.read_bytes(np.prod(conv_layer.get_weights()[0].shape))
                                        kernel = kernel.reshape(list(reversed(conv_layer.get_weights()[0].shape)))
                                        kernel = kernel.transpose([2,3,1,0])
                                        conv_layer.set_weights([kernel])
                        except ValueError:
                                print("no convolution #" + str(i))

        def reset(self):
                self.offset = 0
```

In [20]:

```
def _conv_block(inp, convs, skip=True):
        x = inp
        count = 0
        for conv in convs:
                if count == (len(convs) - 2) and skip:
                        skip_connection = x
                count += 1
                if conv['stride'] > 1: x = ZeroPadding2D(((1,0),(1,0)))(x) # peculiar padding as darknet prefer left and
                x = Conv2D(conv['filter'],
                                   conv['kernel'],
                                   strides=conv['stride'],
                                   padding='valid' if conv['stride'] > 1 else 'same', # peculiar padding as darknet prej
```

```
                                name='conv_' + str(conv['layer_idx']),
                                use_bias=False if conv['bnorm'] else True)(x)
            if conv['bnorm']: x = BatchNormalization(epsilon=0.001, name='bnorm_' + str(conv['layer_idx']))(x)
            if conv['leaky']: x = LeakyReLU(alpha=0.1, name='leaky_' + str(conv['layer_idx']))(x)
        return add([skip_connection, x]) if skip else x

def make_yolov3_model():
        input_image = Input(shape=(None, None, 3))
        # Layer  0 => 4
        x = _conv_block(input_image, [{'filter': 32, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx'
                                      {'filter': 64, 'kernel': 3, 'stride': 2, 'bnorm': True
                                      {'filter': 32, 'kernel': 1, 'stride': 1, 'bnorm': True
                                      {'filter': 64, 'kernel': 3, 'stride': 1, 'bnorm': True
        # Layer  5 => 8
        x = _conv_block(x, [{'filter': 128, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 5},
                            {'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                            {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
        # Layer  9 => 11
        x = _conv_block(x, [{'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 9},
                            {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
        # Layer 12 => 15
        x = _conv_block(x, [{'filter': 256, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 12},
                            {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                            {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
        # Layer 16 => 36
        for i in range(7):
            x = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx':
                                {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky'
        skip_36 = x
        # Layer 37 => 40
        x = _conv_block(x, [{'filter': 512, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 37},
                            {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                            {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
        # Layer 41 => 61
        for i in range(7):
            x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx':
                                {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky'
        skip_61 = x
        # Layer 62 => 65
        x = _conv_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer_idx': 62},
                            {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                            {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
        # Layer 66 => 74
        for i in range(3):
```

```
        x = _conv_block(x, [{'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx'
                                                {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky
    # Layer 75 => 79
    x = _conv_block(x, [{'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 75},
                                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
    # Layer 80 => 82
    yolo_82 = _conv_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx'
                                                {'filter':  255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'le
    # Layer 83 => 86
    x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 84}], s
    x = UpSampling2D(2)(x)
    x = concatenate([x, skip_61])
    # Layer 87 => 91
    x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 87},
                                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
    # Layer 92 => 94
    yolo_94 = _conv_block(x, [{'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx':
                                                {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'lea
    # Layer 95 => 98
    x = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,   'layer_idx': 96}],
    x = UpSampling2D(2)(x)
    x = concatenate([x, skip_36])
    # Layer 99 => 106
    yolo_106 = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx'
                                                {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'le
                                                {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'le
                                                {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'le
                                                {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'le
                                                {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'le
                                                {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'le
    model = Model(input_image, [yolo_82, yolo_94, yolo_106])
    return model
```

In [ ]:

```
# define the yolo v3 model
yolov3 = make_yolov3_model()

# load the weights
```

```
weight_reader = WeightReader('/content/yolov3.weights')

# set the weights
weight_reader.load_weights(yolov3)

# save the model to file
yolov3.save('ffd_model.h5')
```

In [22]:

```
class BoundBox:
  def __init__(self, xmin, ymin, xmax, ymax, objness = None, classes = None):
    self.xmin = xmin
    self.ymin = ymin
    self.xmax = xmax
    self.ymax = ymax
    self.objness = objness
    self.classes = classes
    self.label = -1
    self.score = -1

  def get_label(self):
    if self.label == -1:
      self.label = np.argmax(self.classes)

    return self.label

  def get_score(self):
    if self.score == -1:
      self.score = self.classes[self.get_label()]
    return self.get_score

def _sigmoid(x):
  return 1. /(1. + np.exp(-x))

def decode_netout(netout, anchors, obj_thresh, net_h, net_w):
        grid_h, grid_w = netout.shape[:2]
        nb_box = 3
        netout = netout.reshape((grid_h, grid_w, nb_box, -1))
        nb_class = netout.shape[-1] - 5
        boxes = []
        netout[..., :2]  = _sigmoid(netout[..., :2])
        netout[..., 4:]  = _sigmoid(netout[..., 4:])
        netout[..., 5:]  = netout[..., 4][..., np.newaxis] * netout[..., 5:]
        netout[..., 5:] *= netout[..., 5:] > obj_thresh
```

```
        for i in range(grid_h*grid_w):
                row = i / grid_w
                col = i % grid_w
                for b in range(nb_box):
                        # 4th element is objectness score
                        objectness = netout[int(row)][int(col)][b][4]
                        if(objectness.all() <= obj_thresh): continue
                        # first 4 elements are x, y, w, and h
                        x, y, w, h = netout[int(row)][int(col)][b][:4]
                        x = (col + x) / grid_w # center position, unit: image width
                        y = (row + y) / grid_h # center position, unit: image height
                        w = anchors[2 * b + 0] * np.exp(w) / net_w # unit: image width
                        h = anchors[2 * b + 1] * np.exp(h) / net_h # unit: image height
                        # last elements are class probabilities
                        classes = netout[int(row)][col][b][5:]
                        box = BoundBox(x-w/2, y-h/2, x+w/2, y+h/2, objectness, classes)
                        boxes.append(box)
        return boxes
```

In [23]:

```
def correct_yolo_boxes(boxes, image_h, image_w, net_h, net_w):
        new_w, new_h = net_w, net_h
        for i in range(len(boxes)):
                x_offset, x_scale = (net_w - new_w)/2./net_w, float(new_w)/net_w
                y_offset, y_scale = (net_h - new_h)/2./net_h, float(new_h)/net_h
                boxes[i].xmin = int((boxes[i].xmin - x_offset) / x_scale * image_w)
                boxes[i].xmax = int((boxes[i].xmax - x_offset) / x_scale * image_w)
                boxes[i].ymin = int((boxes[i].ymin - y_offset) / y_scale * image_h)
                boxes[i].ymax = int((boxes[i].ymax - y_offset) / y_scale * image_h)
```

In [24]:

```
def _interval_overlap(interval_a, interval_b):
        x1, x2 = interval_a
        x3, x4 = interval_b
        if x3 < x1:
                if x4 < x1:
                        return 0
                else:
                        return min(x2,x4) - x1
        else:
                if x2 < x3:
                         return 0
```

```
                else:
                        return min(x2,x4) - x3

def bbox_iou(box1, box2):
        intersect_w = _interval_overlap([box1.xmin, box1.xmax], [box2.xmin, box2.xmax])
        intersect_h = _interval_overlap([box1.ymin, box1.ymax], [box2.ymin, box2.ymax])
        intersect = intersect_w * intersect_h
        w1, h1 = box1.xmax-box1.xmin, box1.ymax-box1.ymin
        w2, h2 = box2.xmax-box2.xmin, box2.ymax-box2.ymin
        union = w1*h1 + w2*h2 - intersect
        return float(intersect) / union

def do_nms(boxes, nms_thresh):
        if len(boxes) > 0:
                nb_class = len(boxes[0].classes)
        else:
                return
        for c in range(nb_class):
                sorted_indices = np.argsort([-box.classes[c] for box in boxes])
                for i in range(len(sorted_indices)):
                        index_i = sorted_indices[i]
                        if boxes[index_i].classes[c] == 0: continue
                        for j in range(i+1, len(sorted_indices)):
                                index_j = sorted_indices[j]
                                if bbox_iou(boxes[index_i], boxes[index_j]) >= nms_thresh:
                                        boxes[index_j].classes[c] = 0

# get all of the results above a threshold
def get_boxes(boxes, labels, thresh):
        v_boxes, v_labels, v_scores = list(), list(), list()
        # enumerate all boxes
        for box in boxes:
                # enumerate all possible labels
                for i in range(len(labels)):
                        # check if the threshold for this label is high enough
                        if box.classes[i] > thresh:
                                v_boxes.append(box)
                                v_labels.append(labels[i])
                                v_scores.append(box.classes[i]*100)
                                # don't break, many labels may trigger for one box
        return v_boxes, v_labels, v_scores

# draw all results
def draw_boxes(filename, v_boxes, v_labels, v_scores):
```

```
        # load the image
        data = plt.imread(filename)
        # plot the image
        plt.imshow(data)
        # get the context for drawing boxes
        ax = plt.gca()
        # plot each box
        for i in range(len(v_boxes)):
                box = v_boxes[i]
                # get coordinates
                y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
                # calculate width and height of the box
                width, height = x2 - x1, y2 - y1
                # create the shape
                rect = Rectangle((x1, y1), width, height, fill=False, color='yellow', linewidth = '2')
                # draw the box
                ax.add_patch(rect)
                # draw text and score in top left corner
                label = "%s (%.3f)" % (v_labels[i], v_scores[i])
                plt.text(x1, y1, label, color='yellow')
        # show the plot
        plt.show()
```

In [25]:

```
# define the anchors
anchors = [[116,90, 156,198, 373,326], [30,61, 62,45, 59,119], [10,13, 16,30, 33,23]]

# define the probability threshold for detected objects
class_threshold = 0.6

# define the labels
labels = ["with fire","forest"]
```

In [26]:

```
def load_image_pixels(filename, shape):
  # load image to get its shape
  image = load_img(filename)
  width, height = image.size

  # load image with required size
  image = load_img(filename, target_size=shape)
  image = img_to_array(image)
```

```
  # grayscale image normalization
  image = image.astype('float32')
  image /= 255.0

  # add a dimension so that we have one sample
  image = expand_dims(image, 0)
  return image, width, height
```

In [27]:

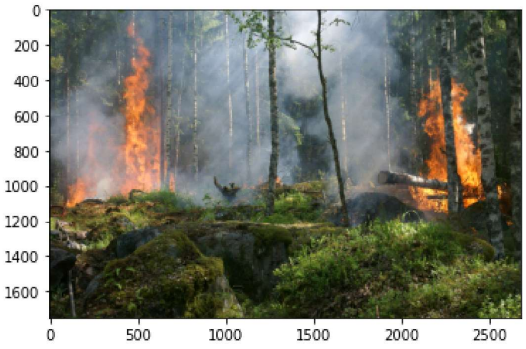
```
def predict():
  from google.colab import files
  upload = files.upload()

  for fn in upload.keys():
    photo_filename = '/content/' + fn

    # define the expected input shape for the model
    input_w, input_h = 416, 416

    image, image_w, image_h = load_image_pixels(photo_filename, (input_w, input_h))

    # make prediction
    yhat = yolov3.predict(image)
    # summarize the shape of the list of arrays
    print([a.shape for a in yhat])

    boxes = list()
    for i in range(len(yhat)):
      # decode the output of the network
      boxes += decode_netout(yhat[i][0], anchors[i], class_threshold, input_h, input_w)

    # correct the sizes of the bounding boxes for the shape of the image
    correct_yolo_boxes(boxes, image_h, image_w, input_h, input_w)

    # suppress non-maximal boxes
    do_nms(boxes, 0.5)

    # get the details of the detected objects
    v_boxes, v_labels, v_scores = get_boxes(boxes, labels, class_threshold)

    # summarize what we found
    for i in range(len(v_boxes)):
      print(v_labels[i], v_scores[i])
```

```
    # draw what we found
    draw_boxes(photo_filename, v_boxes, v_labels, v_scores)
```

In [28]:
```
predict()
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

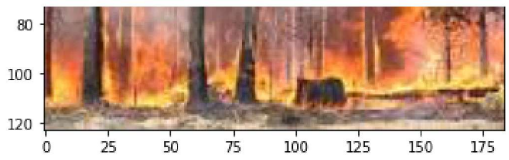
```
Saving output.jpg to output.jpg
1/1 [==============================] - 3s 3s/step
[(1, 13, 13, 255), (1, 26, 26, 255), (1, 52, 52, 255)]
```



In [29]:
```
predict()
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving wildfire.jpg to wildfire.jpg
1/1 [==============================] - 1s 1s/step
[(1, 13, 13, 255), (1, 26, 26, 255), (1, 52, 52, 255)]
```


80
100
120
0
25
50
75
100
125
150
175