Video Classifier

Hassan Imani Parashkooh

January 2023

This document explains how to modify an image classifier and turn it into a frame-by-frame video classifier. The classifier is implemented in the file `label_video.py`.

1 Loading the video file

The initial code contained a function named `read_tensor_from_image_file()` that reads the image file, decodes it based on its format to a tensor, then creates and runs the session to resize and normalize the image tensor. This function could be modified using `tfio.experimental.ffmpeg.decode_video()` to receive video inputs, however, this function was not implemented in the OS that I used. Therefore I changed the function so it reads the video using OpenCV's function `cv2.VideoCapture()`, the frames are then resized and normalized similar to `read_tensor_from_image_file()` function.

```
def read_tensor_from_video_file(file_name,
                                input_height=299,
                                input_width=299,
                                input_mean=0,
                                input_std=255):

    if file_name.endswith(".mp4"):
        cap = cv2.VideoCapture(file_name)
    else:
        raise Exception("The input file must be in mp4 format")
    sess = tf.compat.v1.Session()
    frames = []
    read = True
    while read:
      read, img = cap.read()
      if read:
        frame = tf.convert_to_tensor(img)
        float_caster = tf.cast(frame, tf.float32)
        dims_expander = tf.expand_dims(float_caster, 0)
        resized = tf.compat.v1.image.resize_bilinear(dims_expander, [input_height, input_width
    ])
        normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
        frames.append(sess.run(normalized))
    return frames
```

2 Loading the model and the labels

Using the same functions that were used in the original code, the model file (`load_graph()`) and the labels(`load_labels()`) were loaded. Just a few changes that are commented below, were applied to make them compatible with the TensorFlow version that I used (2.11.0).

```
def load_graph(model_file):
    graph = tf.Graph()
    graph_def = tf.compat.v1.GraphDef() # replaced graph_def = tf.GraphDef()
```

```
4
5       with open(model_file, "rb") as f:
6           graph_def.ParseFromString(f.read())
7       with graph.as_default():
8           tf.import_graph_def(graph_def)
9
10      return graph
```

```
1   def load_labels(label_file):
2       proto_as_ascii_lines = tf.io.gfile.GFile(label_file).readlines() # replaced tf.gfile.GFile(
        label_file).readlines()
3       return [l.rstrip() for l in proto_as_ascii_lines]
```

3 Labeling the frames

The first part of the original code after defining the function was assigning input variables including the input, model and label files in addition to the width and height of the input that the model's network receives. After those lines, parsing arguments were defined to get the defined values from the command line in which the code is running. These lines were kept the same, except for a minor change in the parsing argument related to the input file from `--image` to `--video`.

Next, the model and the video are loaded. As shown below `tf.compat.v1.disable_eager_execution()` is called to prevent the functions called in `read_tensor_from_video_file` to be executed immediately.

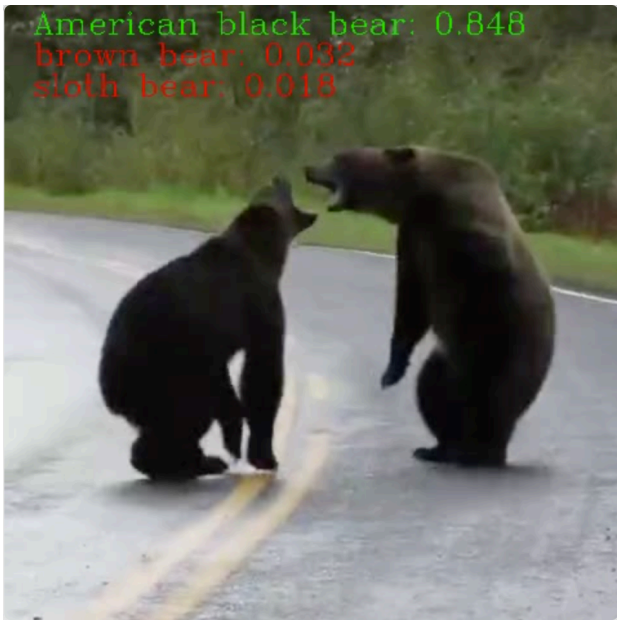
From line 15 to 29 of the code below, a loop goes through the loaded frames and passes each of them through the model. The top-3 predictions and their confidence values are stored in the list `objs`. In line 33, the list `objs` and the video file are passed to the initializer of class `play_vid` that I wrote to display the predictions on the video with the related color (green for confidence greater than 0.8, red otherwise)

```
1       graph = load_graph(model_file)
2       tf.compat.v1.disable_eager_execution()
3       ts = read_tensor_from_video_file(
4           file_name,
5           input_height=input_height,
6           input_width=input_width,
7           input_mean=input_mean,
8           input_std=input_std)
9
10      input_name = "import/" + input_layer
11      output_name = "import/" + output_layer
12      input_operation = graph.get_operation_by_name(input_name)
13      output_operation = graph.get_operation_by_name(output_name)
14      objs = []
15      for fn,t in enumerate(ts):
16          with tf.compat.v1.Session(graph=graph) as sess:
17              results = sess.run(output_operation.outputs[0], {
18                  input_operation.outputs[0]: t
19              })
20          results = np.squeeze(results)
21
22          top_k = results.argsort()[-3:][::-1]
23          labels = load_labels(label_file)
24          temp = []
25          for i in top_k:
26              temp.append([labels[i],results[i]])
27
28          objs.append(temp)
29
30  from vid_utils import play_vid
31
32  play_vid(file_name,objs)
```

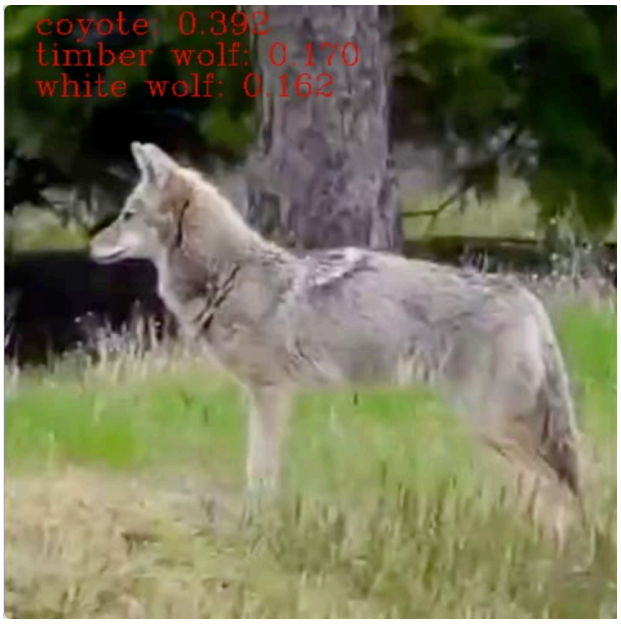
4 Results

Some sample videos were download, trimmed and cropped to test the code. The sample videos are the following: video1, video2, video3, video4, video5, video6, video7, video8. The directory `/test` contains the trimmed and cropped version of these videos. The bash script `driver.sh` feeds the videos to the classifier and stores the output videos in the directory `/outputs`. Some of the results are shown in figure 1

(a) From `hlz.mp4`. The correct class is predicted with the confidence above 0.8, displayed in green. The runner-ups with confidences below 0.8 are shown in red



(b) A frame from `krs.mp4`. The correct class (American black bear) is predicted with the confidence of 0.848 despite having multiple objects of the class in the frame.



(c) Taken from `kyt.mp4`. The correct class (Cayote) is predicted correctly but with 0.392 confidence, therefore it is shown in red

(d) A frame from `asa.mp4` in which the body of the object (hippopotamus) is under water, causing the network to label it incorrectly

Figure 1: Snapshots from the video results