Object Detection with YOLO-V2-TINY using Tensorflow graph construction

Anchit Tandon 20206033
Jinwoo Kim 20160171

April 2020

1 How we Implemented

1.1 Read and write of videos

From command line arguments specifying input/output videos, our implemented function `open_video_with_opencv()` uses `cv2.VideoCapture` to import input video to a `cv2.VideoWriter` object. Then, the function checks if the video opened successfully and retrieves width, height, fps, and number of frames for later use. Then, for output video, a `cv2.VideoWriter` object with mp4v encoding and same width, height and fps to input video is made. Then, the function returns input/output video objects and frame size (width, height, and number of frames) for later processing.

1.2 Tensorflow Graph Construction

In function `YOLOv2_TINY.build_graph()`, we first loaded the pretrained weight parameters from the pickled binary and extracted kernel, layer biases, moving mean, moving variance, and gamma with an auxiliary function `_w_to_tensor()`. Then, with `self.g` as the default graph by using the `as_default()` and `self.proc` passed as a command line argument as device for tensor computation, we defined each layers of the graph. We used `tf.nn.placeholder()` as input layer, and used `tf.nn.conv2d()`, `tf.nn.bias_add()`, `tf.nn.batch_normalization()`, `tf.nn.leaky_relu()`, and `tf.nn.max_pool2d()` as intermediate computation layers. We followed the network architecture of `YOLOv2tiny` given in onnx profile viewed with neutron.

1.3 Inference

Given input/output video objects and initialized model graph, we loop over the frames and carry out inference in `video_object_detection()`. At each iteration, we read an input video frame and resize it using `cv2.resize()` in function `resize_input()` to meet `YOLOv2_tiny` specification. Then, in `YOLOv2_TINY`

`.inference()`, we run the graph computation by calling `self.sess.run()` with resized input image and get computed tensor of last layer as model output.

Then, we run `postprocessing()` on the output tensor to obtain categorical bounding boxes. Because inference is performed on resized images, we performed additional adjustment on the parameters of the bounding boxes to map them to original image space. This can be achieved simply by multiplying the $x-$ and $y-$ box coordinates and sizes by $w_0/416$ and $h_0/416$ respectively, where (w_0, h_0) is original image size and $(416, 416)$ is model input size. Finally, we drew rectangles and corresponding labels on the input frame, and wrote the accumulated image to the output video file. After all iterations, we release the opened videos.

For performance measurement, we measured end-to-end inference time and only-inference time separately. For end-to-end measurement, we treated entry point for inference loop as first-end and loop termination as second-end, omitting procedures up to model graph initialization. For only-inference time, we summed up computation time of `YOLOv2_TINY.inference()`. For fps measurement, we simply divided the total number of processed frames (453) by total elapsed time.

2 A comparison of results from CPU and GPU

Device	End-to-end time (s)	Only-inference time (s)	End-to-end fps(frames/s)	Only-inference fps (frames/s)
GPU	22.19	6.97	20.42	64.95
CPU	42.04	24.51	10.77	18.48

3 Analysis of results from CPU and GPU

From the above table, we can see that processing on GPU reduces end-to-end inference time by 47% over CPU for this application. Especially, The only-inference time, reflecting graph computation which is primarily parallelized by GPU, is reduced by nearly 72% by using GPU as compared to CPU.

4 Conclusion

Due to the increased parallelism, hidden latency and high memory bandwidth offered by GPU architecture, we are able to compute CNN inference faster in GPU than CPU.