Darknet Source Code Starter Guide

ECE382M.20, SoC Design

Dimitrios Gourounas

---

1) Important data structures

Darknet is exclusively written in C. Hence, instead of classes, structs are used for the network's various data structures. The most important of them can be found under **include/darknet.h:**

- **struct layer:** Also defined (in comments) in **src/layer.h** for convenience, this struct contains all the information required to encapsulate a darknet layer. It can be of many different types, such as convolutional, maxpool, dropout etc. This struct contains a lot of information, but the fields you will most likely need the most are the **weights** pointer, containing the layer's weights and the **outputs** pointer. Each layer type has different functions associated with it. The one that you should mostly look into is the **forward()** function, since this lab focuses on inference and not training. Each layer type's **forward()** function is declared in the associated C file. For example, the convolutional layer's **forward()** function is in **src/convolutional_layer.c**, **forward_convolutional_layer()**.
- **struct network:** Also defined (in comments) in **src/network.h**, this struct has, again, a lot of information, but the most important one is the **layers** pointer. Upon initialization of the network, each layer is properly allocated and each layer's weights are loaded from the trained weight values. Darknet generates the network architecture by parsing the configuration file provided in the command line. In our case, we will be using tiny Yolo, whose configuration file is under **cfg/tiny-yolo.cfg**.
- **struct network_state:** Struct that contains the **struct network** and its current state. The most important fields are the **input** and **workspace** pointers. For example, the **workspace** pointer will point to the array that holds each layer's input data, as we traverse the network during inference.

2) Inference flow in Darknet code

When running the darknet executable with the **'detector'** option enabled, the **run_detector()** function will get called. This function is located in **src/detector.c** and will eventually call the function **test_detector()** in the same file, when the **'test'** argument is given in the command line.

The **test_detector()** function will firstly load all the network weights (hint: this might be a good place to pre-load any modified weights you created, e.g. weights converted to fixed-point) from the input files (e.g. **yolov3-tiny.weights**) and then iterate over all the images and call the **network_predict()** function, which runs inference on each image across the network's layers. After inference completes, runtime is reported and the **draw_detections_v3()** function is called to draw the boundary boxes in the image.

The **network_predict()** function is located in **src/network.c**, and calls the **forward_network()** function, which in turn iterates over all the network's layers and calls each layer's associated **forward()** function.

As you will notice by your profiling, the most time consuming function is **gemm()**, which will get called by the convolutional layer's **forward()** function. Spend some time to understand the flow of the **forward_convolutional_layer()** function in **src/convolutional_layer.c**. Notice where the convolution matrices are referenced and how the **im2col_cpu_ext()** (located in **src/im2col.c**) function is called before **gemm()** is called, in order to transform the input images and to convert the convolution operation into a matrix-multiplication (as we saw in lecture and homework). The **gemm()** function is where the primary focus of your work will be, located in **src/gemm.c**.